Универзитет у Београду

Електротехнички факултет

Катедра за сигнале и системе

Милош Милошевић 83/2012

Дипломски рад

**Управљање системом осветљења у паметним зградама**

Ментор: Проф. др Жељко Ђуровић

# Захвалност

Захваљујем се руководству научно-истраживачког института „RT-RK“ на указаној прилици да се боље упознам са начином рада у инжењерском окружењу и будем укључен у процес развоја нових програмских решења. Захваљујем се Милету Давидовићу, Мићи Ћетковићу и Ђорђу Пешићу.

Желео бих да се захвалим и професору Жељку Ђуровићу на неисцрпном извору знања, идеја и мотивације које несебично преноси на студенте.

На крају, највећу захвалност дугујем својој породици.

# Сажетак

Циљ овог рада био је развој С++ корисничке апликације за управљање осветљењем у паметној згради. Развијена апликација се покреће на *Raspberry Pi* платформи и подржава графичко корисничко окружење. Рад даје и опис елемената паметних зграда, преглед модерних технологија које се користе у системима кућне аутоматизације и осврт на архитектуру *Oblo* система као једног од конкретних решења кућне аутоматизације.

# Кључне речи

*Home automation, Lighting control, Oblo,* *Raspberry Pi*, *cross-compile*, *gtkmm*

# Садржај

[1. Увод 6](#_Toc461493641)

[2. Преглед елемената и архитектуре паметних зграда 7](#_Toc461493642)

[2.1. Системи кућне аутоматизације 8](#_Toc461493643)

[2.2. *Z-Wave* и *ZigBee* протоколи 10](#_Toc461493644)

[2.3. MQTT протокол 12](#_Toc461493645)

[2.4. *Internet of things* 13](#_Toc461493646)

[2.5. *Oblo* систем кућне аутоматизације 14](#_Toc461493647)

[3. *Raspberry Pi* платформа и унакрсно превођење 17](#_Toc461493648)

[4. Предлог решења 21](#_Toc461493649)

[5. Имплементација решења 24](#_Toc461493650)

[5.1. Модул за комуникацију са гејтвејом 24](#_Toc461493651)

[5.2. Модул за складиштење података 28](#_Toc461493652)

[5.3. Модул за управљање 30](#_Toc461493653)

[5.4. Модул за рад са екраном 33](#_Toc461493654)

[5.5. Главни модул 37](#_Toc461493655)

[6. Закључак 38](#_Toc461493656)

[Литература 39](#_Toc461493657)

[Прилог А: *Makefile* за превођење апликације 40](#_Toc461493658)

# Листа симбола

ARM *Advanced RISC Machinе -* тип процесорске архитектуре

g++ *GNU C++ Compile -* стандардни *Linux* преводилац за С++

GUI *Graphical User Interface -* графичко корисничко окружење

gtkmm *GTK minus minus -* GTK омотач за С++

GTK *GNU Image Manipulation Toolkit -* скуп библиотека за рад са графиком

JSON *JavaScript Object Notation -* формат размене података

MQTT *MQ Telemetry Transport* - протокол размене података

OHM *Oblo Home Manager -* централна јединица *Oblo* система

POCO *Portable Components -* библиотека за развој наменских система

SSH *Secure Shell* - протокол за удаљени приступ рачунару

UML *Unified Modeling Language* - језик за представљање објектног модела

# Увод

Сведоци смо наглог развоја технологије и чињенице да модерни уређаји, све бржи рачунари и аутоматизовани системи налазе примену у многим сферама људског деловања те неповратно утичу на промену стила живота. Управо су развој технологије и жеља за лагоднијим животом резултовали појавом паметних кућа, система контроле над уређајима у стамбеном објекту.

Овај рад предлаже једно занимљиво решење за регулацију и управљање расветом у паметној кући. Решење се ослања на *Oblo* систем, идејно решење   
научно-истраживачког института „RT-RK“ у области кућне аутоматизације. Такође, рад укључује и примену *Raspberry Pi* рачунара, рачунара опште намене и малих габарита чија област примене сеже до граница људске креативности. Срж рада односи се на методе аутоматског управљања примењене у оквиру система за регулацију осветљености.

Структура рада састоји се од седам целина. Након уводног поглавља представљени су елементи и архитектура паметних зграда. У другом поглављу описане су и технологије релевантне за развој решења описаног овим радом. Треће поглавље односи се на *Raspberry Pi* платформу за коју је решење развијено. Додатно, треће поглавље говори и о техници унакрсног превођења која омогућава развој софтвера на персоналном рачунару и његово превођење у облику који је погодан за извршавање на некој од других платформи. Четврто поглавље описује архитектуру предложеног решења. Док је конкретна имплементација предложене архитектурe дата у петом поглављу. Претпоследње поглавље представља осврт на успешност реализације и правце даљег развоја решења, a последње поглавље садржи списак литературе која је коришћена приликом реализације рада.

На самом крају, у прилогу A дат је *Makefile* фајл којим је вршено превођење апликације. Коришћење овог фајла омогућава превођење апликације и за извршавање на рачунару и на *Raspberry Pi* платформи.

# Преглед елемената и архитектуре паметних зграда

## Системи кућне аутоматизације

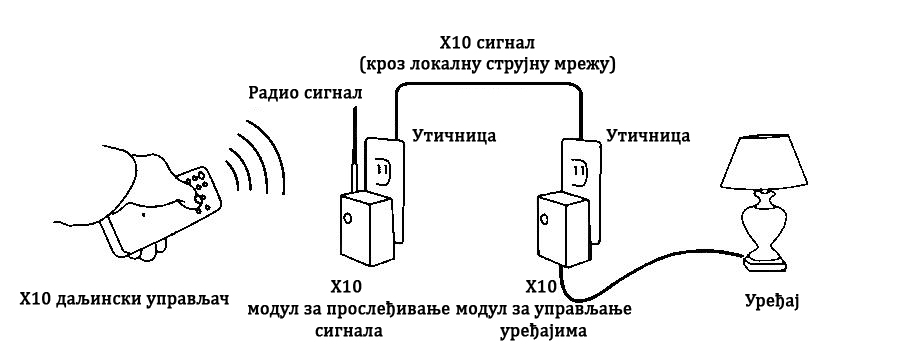
Појам кућне аутоматизације уведен је још средином прошлога века. Тада, односио се на електричне уређаје (шпорете, фрижидере, расхладне уређаје итд.) који су радили делимично самостално и имали могућност програмирања. Данас, кућна аутоматизација подразумева интегрисане системе сачињене од више десетина уређаја чијом се спрегом реализује висок степен аутономности у области безбедности и надгледања, управљања расветом, управљања грејањем и хлађењем, потрошње електричне енергије итд. Поменути системи најчешће укључују велики број различитих технологија и поред саме аутоматизације омогућују кориснику и удаљени приступ уређајима у кући користећи мобилни телефон или други уређај који има могућност повезивања на интернет. Зграде које су опремљене системима кућне аутоматизације називамо паметним зградама.

Примери погодности које паметне зграде доносе су небројени. Више нема дилеме јесу ли врата од стана остала окључана, хоће ли заборављени шпорет изазвати пожар или да ли је неко провалио у зграду. Решење тих проблема налази се на Вашем мобилном телефону. Неки од произвођача отишли су и корак даље и своја решења наоружали методама вештачке интелигенције. Тако многи системи кућне аутоматизације имају могућност самообучавања на основу навика корисника. Паметна кућа може да „зна“ да се њени укућани најчешће буде у осам часова изјутра и да тада воле да доручкују у топлијој соби, да се уморни враћају са посла око шест часова поподне или да воле да спавају у хладнијој постељи. У складу са тиме систем самостално управља грејањем и хлаћањем како би постигао жељену температуру.

Системи за кућну аутоматизацију су у пионирском добу своју примену углавном налазили у бинарном управљању електичним уређајима. Тако је 1975. године шкотска фирма *Pico Electronics* представила своје решење *X10* које и после више од четрдесет година налази своју примену широм света. Поставка *Х10* система се иницијално састојала од конзоле за управљање са 16 дугмади и више *Х10* модула за управљање уређајима (енгл. *X10 Appliance module*). Конзола је жичаном везом била повезана са свим модулима за управљање уређајима, а модули су у зависности од стања дугмади укључивали или искуљчивали уређаје прикључене на њих. За пренос података *Х10* систем користио је постојеће струјне инсталације унутар куће на начин да је конзола, у тренуцима када је напон локалне мреже приближно нула   
(100 пута у секунди) слала инстукције у виду поворке импулса учестаности 120KHz у дужини од 1ms. Модули за управљање уређајима ослушкивали су локалну мрежу и детектовали информације које конзола шаље те укључивали или искључивали уређаје прикључене на њих.

Наредни корак у еволуцији *Х10* система била је појава даљинског управљача. Слично конзоли, и даљински управљач на себи је носио 16 дугмади, али је поред тога могао да комуницира и радио сигналом. Појава бежичног начина комуникације захтевала је модул који би могао да разуме емитовани радио сигнал и проследи га постојећој инсталацији кроз жичану везу по принципу описаном у претходном поглављу. Решење је дошло у облику *Х10* модула за прослеђивање сигнала (енгл. *Х10 Transceiver Module*). Архитектура *Х10* система приказана је на слици 2.1.

Слика 2.1: Принцип рада *X10* система за кућну аутоматизацију



Недостаци оваквог система су бројни, а углавном су везани за начин преноса информација путем локалне струјне мреже. Несумњиво, највећи проблем односи се на брзину преноса информација. Наиме, поруке којима *Х10* систем комуницира састоје се од 36 бита (4 старт бита, 16 бита адресе *Х10* уређаја којем је порука упућена и 16 бита кодиране команде). Како је при сваком проласку напона локалне мреже кроз нулу могуће пренети информацију о само једном биту и уз то се све поруке шаљу двапут узастопно долазимо до закључка да је за пренос једне поруке потребно око 0,7 секунде. Додатни проблеми односе се на чињеницу да у систему нема ниједног нивоа заштите и енкрипције порука.

Свесни проблема које је *Х10* концепт носио, инжењери су се у последњој деценији двадесетог века окренули развоју бољих решења жичане комуникације уређаја. Као епилог рада појавило се више комуникационих протокола. Брзина преноса података је устостручена, смањени су губици и уведене су нове технике заштите података. Међутим, оно што је кочило даљи развој система кућне аутоматизације јесте чињеница да су рачунари у том добу били јако скупи и гломазни те је сваки од тадашњих система још увек захтевао непосредно учешће човека у доношењу одлука.

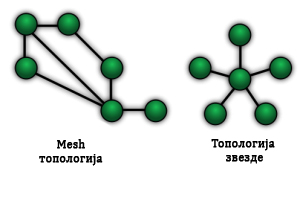
Улазак у трећи миленијум доноси нагли развој електронике. Рачунарске компоненте постају све мањих димензија, јефтиније и захтевају мање енергије за свој рад. Новитети су се појавили и на пољу рачунарских комуникација, а највећи утицај на паметне куће имао је изум *Z-Wave* и *ZigBee* мрежа.

## *Z-Wave* и *ZigBee* протоколи

*Z-Wave* и *ZigBee* представљају бежичне комуникационе протоколе развијене наменски за системе кућне аутоматизације. Свој рад заснивају на радио комуникацији и одликују се високим нивоом економичности. Чипови који се уграђују у уређаје и омогућавају *Z-Wave* и *ZigBee* комуникацију садрже велики број дигиталних кола, а аналогне компоненте садрже само на местима где је то неопходно. Захваљујући оваквој конструкцији, такви чипови су малих димензија и мали потрошачи па је радни век уређаја између две промене батерија и до 10 година. Један *ZigBee* чип приказан је на слици 2.2.

Слика 2.2: *ZigBee* хардверски модул

Кључна разлика између ова два протокола јесте у фреквенцији на којој остварују комуникацију. *ZigBee* комуницира на фреквенцији од 2,4GHz, а *Z-Wave* на 908MHz. Већа фреквенција *ZigBee* уређајима омогућава размену података брзином од   
250kbit/s, а *Z-Wave* уређајима брзином до свега 100kbit/s. Домет уређаја је такође различит. *ZigBee* уређаји најчешће остварују око 10m домета у затвореној просторији, а *Z-Wave* уређаји могу да комуницирају и на 30m при истим условима. Овако мали домет не може да обезбеди комуникацију унутар целе куће и учинио би   
*Z-Wave* и *ZigBee* уређаје бескорисним да оба протокола не подржавају *mesh* комуникацију. *Mesh* концепт подразумева да сваки уређај може да пошаље нову поруку, али и да проследи поруку од другог уређаја која није намењена њему. На овај начин омогућено је да комуникацију могу да остваре и уређаји који нису у директном међусобном домету. Разлика између стандрадног начина комуникације и *mesh* комуникације ислустрована је на слици џџ.

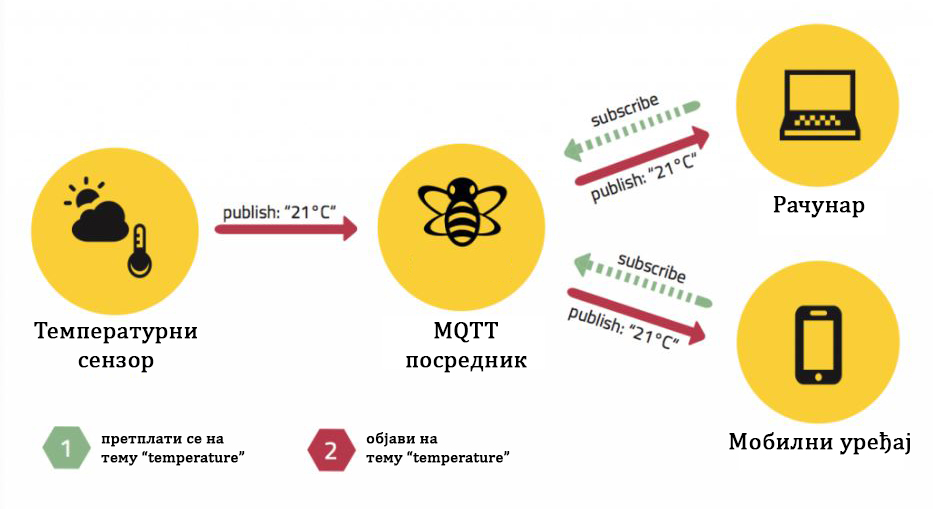
Слика џџ: Разлика између *mesh* топологије и топологије звезде

Дакле, закључили смо да је могуће повезати више уређаја и поруке слободно добацивати од једног до другог све док не стигну до свог циља. Ново питање је колико уређаја може повезати једна *Z-Wave*, односно *ZigBee* мрежа? Одговор поново зависи од одабира протокола. У случају *Z-Wave* мрежа један уређај може бити повезан са још 232 *Z-Wave* уређаја, а уколико је одабир *ZigBee* мрежа, један уређај може бити повезан са још 240 *ZigBee* уређаја. Оваква лимитација последица је чињенице да су сваки уређај Не само да подржавају овако велики број уређаја него је и додавање нових урећаја у мрежу веома једноставно. Закључујемо да су *Z-Wave* и *ZigBee* мреже

Развој *Z-Wave* и *ZigBee* комуникационих протокола значајно је олакшао и унапредио комуникацију уређаја. Захваљујући чињеници да су модерни комуникациони чипови мали потрошачи и малих димензија, јако брзо су нашли место у многим уређајима. Инфорамције су постале све доступније, а следећи корак био је осмислити ефикасан и универзалан језик на ком би уређаји комуницирали. Радио се MQTT прокол.

## MQTT протокол

MQTTпротокол представља скуп правила за пренос података између два уређаја. Одликује се јако простим и малим „потписом“ што га чини погодниим у ситацијама када уређаји имају ограничене ресурсе у погледу мрежног протока или утрошка енергије. Велики број развијених програмских библиотека и лака интеграција у већини програмских језика чини MQTTпротокол пожељним комуникационим решењем. Развијен је почетком двадесетпрвог века са идејом да буде основни концепт комуникације веома свеже и популарне технолошке области која је у енглеској терминологији позната као *Internet of Things (IoT)*. *Internet of Things* се заснива на мрежи великог броја уређаја који обављају просте улоге, али имају и могућност међусобне комуникације. Спрегом великог броја уређаја реализују се знатно сложеније фунцкије.

MQTTкомуникациона мрежа састоји се из два типа ентитета, клијената   
(енгл. *MQTT Clients*) и посредника(енгл. *MQTT Broker*)*.* Клијенти немају могућност директне комуникације већ размена података увек иде преко посредника. Адресирање порука врши се коришћењем тема (енгл. *topic*)и занива се на принципу објављивања и претплате (енгл. *publish*/*subscribe*). Сваки од клијената може се претплатити на тему или послати поруку на тему. Улога посредника је да пристиглу поруку проследи свим клијентима који су претплаћени тему на коју је порука послата (слика 2.4). Садржина поруке је најчешће произвољна што омогућава лаку комуникацију између два клијента уколико су обе стране упознате са форматом поруке. У случајевима када порука има сервисни карактер, на пример пријављивање клијента посреднику или претплаћивање на нову тему, тада је формат поруке прописан од стране посредника.

Слика 2.4: Принцип рада MQTT протокола

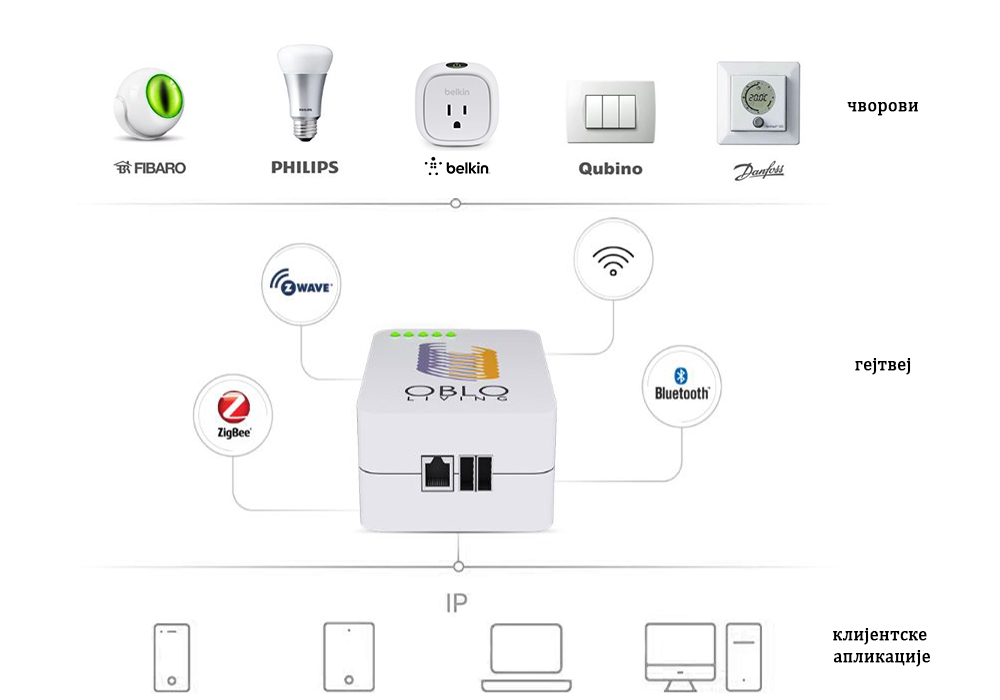
## *Internet of things*

## *Oblo* систем кућне аутоматизације

Узмимо поређење паметне куће и људског организма као илустративан пример на ком можемо објаснити концепт модерних система кућне аутоматизације. Људски орагнизам је способан да осматра околину и бележи промене у њој, доноси одлуке како да реагује или како да се адаптира на те промене и на крају спроводи те одлуке у дело. Јако слично, и системи кућне аутоматизације се састоје од великог броја сензора који им омогућују да осматрају околину и бележе промене у њој. За потребе доношења одлука, односно обраду информација из спољашње средине, системи кућне аутоматизације користе централни рачунар. Његова улога може лако да подсети на људски мозак. На самом крају, системи кућне аутоматизације поседују и велики број актуатора који им омогућују да адекванто реагују на велики број догаћаја. Дакле информација путује од сензора до централног рачунара, ту се обрађује и доноси се одлука о акцији, а одлука о акцији се прослеђује актуаторима на извршавање.

Практично, улога централног рачунара јесте да управља великим бројем уређаја. Он мора да познаје начине како да им се обрати и како да тумачи њихове поруке, чак и када сви ти уређаји долазе од различитих произвођача и подржавају различите стандарде. Задатак централног рачунара јесте и да зна како да се обрати човеку и како да представи прикупљене информације и донешене одлуке.

Анализу модерног система кућне аутоматизације, са техничке тачке гледишта, можемо дати на примеру *Oblo* ситема. *Oblo* систем кућне аутоматизације представља хардверско-софтверско решење развијено од стране научно-истраживачког института за наменске рачунаре „RT-RK“. У реализацији оваквог система користи се велики број различитих технологија и улога програмера је пројектовање таквог система у којима ће комуникација ентитета који користе различите технологије бити несметана. Са техничког становишта, читав систем се састоји од три типа ентитета (слика 2.3):

* Гејтвеј (енгл. *gateway*) представља централни рачунар. На њему се извршава софтвер и улога му је доношење одлука и комуникација са корисником, сензорима и актуаторима. За комуникацију са корисником користи се локална мрежа или клауд (енгл. *cloud*), а за комуникацију и управљање сензорима и актуаторима користе се *Z-Wave* и *ZigBee* бежичне мреже. Софтвер који се извршава на гејтвеју развијен је у духу програмирања за наменске системе (енгл. *embedded programming*), покреће се на *Linux* оперативном систему, писан је на С++ програмском језику и извршава се у реалном времену. На гејтвеју се покрећу само они сервиси који су неопходни за рад система кућне аутоматизације. На тај начин извршена је уштеда ресурса.
* Чворови (енгл. *nodes*) представљају сензоре и актуаторе. Они имају могућност остваривања комуникације по *Z-Wave* и *ZigBee* протоколима и њиховим коришћењем реализују интеракцију са гејтвејом. Ови уређаји пројективани су тако да троше што мање енергије и захтевају што мање ресурса па се и развијају на оскуднијем хардверу. Неки од чворова су: сензор дима и ватре, сензор осветљености, паметна утичница, паметна сијалица...
* Корисничке апликације имају улогу интеракције са корисником и у складу са тиме подржавају богат графички интерфејс. Њиховим коришћењем корисник диктира понашање читавог система. Ове апликације извршавају се на уређајима који немају подршку за *Z-Wave* *и ZigBee* комуникационе протоколе па интеракцију са сензорима и актуаторима остварују посредством гејтвеја. Технологије по којима су корисничке апликације развијне су разноврсне и зависе од платформе на којој се покрећу. Свим апликацијама је заједничко да се извршавају на рачунарима опште намене, дакле, на рачунарима на којима се поред ових апликација извршава и прегршт других сервиса. Корисничке апликације се не извршавају у реалном времену.

Слика 2.3: Структура *Oblo* система

Пошто смо се упознали са хардверском архитектуром *Oblo* система, усресредимо пажњу на пар софтверских конструкција које имају велики утицај на концепт решења који овај рад предлаже. Целокупан софтвер који се на гејтвеју покреће састоји се из 3 модула који међусобно сарађују. Тек рад свих модула заједно даје праву улогу гејтвеју. Модули унутар *Oblo* система су:

* *Oblo System Manager* (OSM) задужен да обезбеди осталим модулима неопходне услове за рад, представља посредника изеђу оперативног система гејтвеја и *Оblo* софтвера, управља везама и нуди сервисне услуге осталим модулима.
* *Oblo Message Broker* (OMB) задужен за контролу међупроцесне комуникације и прослеђивање порука. Сва комуникација, унутар *Oblo* система, али и са спољашњим ентитетима, реализује се захваљујући   
  OMB-у. Како је за комуникацију унутар *Oblo* система изабран MQTT комуникациони протокол, овај модул уједно представља и MQTT посредника.
* *Oblo Home Manager* (OHM) суштински представља језгро система. Овај модул задужен је за управљање чворовима и доношење свих одлука унутар система. OHM складишти податке и извршава логику коју је корисник задао путем корисничких апликација.

Модул са којим ће корисничка апликација интензивно сарађивати јесте OHM. Практично, OHM ће бити наш прозор у свет. Све информације о стању сензора и сва управљања актуаторима биће реализована посредством OHM-а. Наравно, не можемо заобићи OMB модул, јер све поруке морају проћи кроз њега, али како је OHM наш крајњи циљ, сматраћемо да се обраћамо право њему.

На овом месту морамо поменути и JSON (енгл. *JavaScript Object Notation*) формат. Све поруке *Oblo* система спаковане су у овај формат чија је нотација лако читљива човеку, а уједно омогућава и лако паковање и издвајање података коришћењем рачунара. JSON формат подразумева упаривање величине са њеном вредношћу у формату “величина” : “вредност”. Ради бољег разумевања следи пример једне JSON поруке унутар *Oblo* система:

{

"msg\_type":"command",

"command":"set\_service\_property\_value",

"service\_name": "DimService",

"property\_name" : "Level",

"property\_value" : 30,

"id" : 27

}

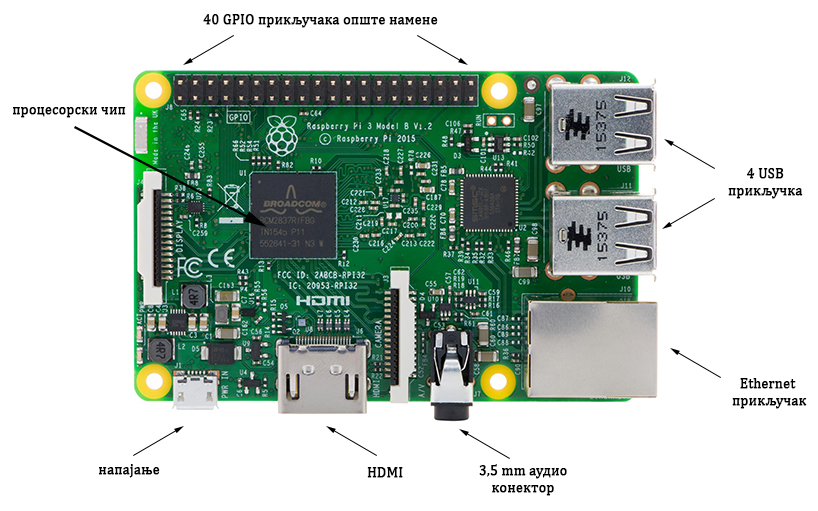
# *Raspberry Pi* платформа и унакрсно превођење

*Raspberry Pi* представља мали и приступачни рачунар (слика 3.1) развијен са идејом да буде школска платформа за изучавање информатике, рачунарства и програмирања. Идеја се јавила 2006. године на Кембриџ универзитету као последица чињенице да је знање студената рачунарских наука постајало све мање и сводило се на познавање једне уске области програмирања. За реализацију је било потребно шест година, и у фебруару 2012. године на тржишту се појавио први модел овог рачунара. До данас развијено је шест различитих модела у три генерације. Модели се међусобно разликују по брзини процесора, количини меморије којом располажу и број улазно-излазних прикључака. Долазак *Raspberry Pi* рачунара имао је позитиван утицај на обуку младих програмера, али је привукао и велику пажњу ентузијаста. Јако брзо на интернету се појавио велики број радова из обласи рачунарске технике, који описују занимљиве практичне примене *Raspberry Pi* платформе.

Слика 3.1: *Raspberry Pi* рачунари прве генерације (лево) и друге генерације (десно)

Сви *Raspberry Pi* модели користе ARM процесорске чипове произведене од стране фирме *Broadcom*. Такт чипова је у распону од 700 MHz на платформи прве генерације до 1.2 GHz на најмодернијој платформи треће генерације. И количина RAM меморије зависи од генерације и креће се у распону од свега 256 MB на плочи прве генерације до моћних 1 GB на плочама друге и трече генерације. Овакви хардверски ресусри омогућавају *Raspberry Pi* рачунарима лако извршавање већине програмерских задатака.

Све плоче долазе са уграђеним USB портовима, на које је могуће прикључити тастатуру, миша или неку другу периферију. Прикључак за *Ethernet* мрежу је такође подразумеван па свим *Raspberry Pi* рачунари имају могућност повезивања на интернет. Од аудио и видео прикључака, плоче су опремљене HDMI излазом што им омогућује лако повезивање на стандардне рачунарске мониторе, док стандрадни   
3,5 mm аудио прикључак гарантује повезивање са великим бројем аудио уређаја.

Поред описаних прикључака, *Raspberry Pi* рачунари поседују и одређен број дигитални прикључака опште намене (енгл. *General purpose input-output,* GPIO). Постојање GPIO пинова омогућава експериментисање и прикључивање нестандардних периферија ниског нивоа (релеја, потенциометара, транзистора, диода, дугмића итд.). За напајање плоча користи се стандарни *micro USB* прикључак напона 5V. Распоред свих прикључака приказан је на слици 3.2.

Слика 3.2: Распоред прикључака на *Raspberry Pi 3* рачунарској плочи

За смештање и чување података *Raspberry Pi* рачунари користе *microSD* меморијске картице. На картици се поред корисничких података чува и комплетан оперативни систем. Чињеница да се сви подаци смештају на меморијску картицу омогућава лако прављење резервне копије (енгл. *backup*). Преписивањем постојеће копије на картицу могуће је враћање било ког запамћеног стања целокупног софтвера.

Избор оперативних система које *Raspberry Pi* рачунари подржавају је велик: *Raspbian Windows 10, Ubuntu MATE, FreeBSD, Arch Linux ARM* итд. Међутим, препоручен оперативни систем је *Raspbian,* верзија *Linux* оперативног система оптимизована за *Raspberry Pi* рачунаре која са собом носи неопходне библиотеке за лагодан рад и развој софтвера. *Raspbian* подржава како рад у графичком окружењу тако и рад коришћењем терминала. Рад путем терминала нуди знатно већу контролу, а уз помоћ *SSH* протокола *(*енгл. *Secure Shell protocol)* могуће је и удаљено приступање рачунару.

Иако *Raspberry Pi* представља рачунар опште намене са графичким окружењем, програмерска пракса каже да се развој софтвера ипак врши на персоналном рачунару, а на платформи само извршавају преведени фајлови. Оваква техника назива се унакрсно превођење (енгл. *cross-compile*), а разлози за њену примену су бројни. Пре свега, персонални рачунари су бржи, раде на процесорима са више језгара и имају знатно више меморије. Потом, највећи број интегрисаних развојних окружења (енгл. *Integrated Development Environment*, IDE) и развојних алата намењен је персоналним рачунарима. Додатни мотив представља чињеница да је познавањем техника унакрсног превођења могућ развој софтвера и за платформе чији оперативни ситеми немају могућност покретања у графичком режиму.

С++ изворни код се, на *Linux* оперативном систему, најчешће преводи коришћењем стандардног *g++* програмског преводиоца. Позив за превођење се упућује из командне линије и његова синтакса је следећа:

g++ [**-I***dir*...] [**-L***dir*...] [**-o** *outfile*] *infiles*

У оквиру [**-I***dir*...] блока наводи се путања до свих фолдера који садрже фајлове заглавља (енгл. *header files*) нпр. -I./include, -I/home/me/randomplace/include… У оквиру [**-L***dir*...] блока наводи се путања до свих фолдера који садрже дељене библиотеке од којих корисничка апликација зависи нпр. -L./lib   
-L/home/me/someplace/libs… Блок [**-o** *outfile*] носи информације од излазном извршивом фајлу и његовој адреси. На самом крају наредбе, у оквиру *infiles* блока наводи се списак путања до свих фајлова који садрже изворни код. Додатно, на крају *infiles* блока наводе се и имена свих библиотека које апликација користи у облику   
-**l***libname*. Све путање које се наводе у позиву *g++* програмског преводиоца могу бити и у апсолутном и у релативном формату.

Додатну погодност приликом превођења нуди *make* алат. Његова улога се своди на позивање описане *g++* наредбе. Међутим, предност се огледа у томе што се наредба памти у текст фајлу и није је потребно увек изнова куцати. Уколико се током развоја апликације појави нови фајл са изворним кодом потребно га је једноставно дописати у постојећи текст фајл. Кључна погодност коју *make* алат нуди исказује се при превођењу већих апликација са великим бројем фајлова, реда величине хиљада. Тада, уколико је дошло до мале промене на једном фајлу неће се вршити поновно превођење читаве апликације пошто то може трајати сатима већ ће *make* алат поновно превести само део на који је промена утицала. Поред описаних погодности, приликом писања фајлова за *make* алат могуће је дефинисати променљиве које ће касније бити коришћене у том фајлу. На овај начин постиже се боља прегледност. Како би покренули *make* алат потребно је да фајл који садржи сет инструкција назовемо *Makefile* и потом у командној линији позовемо наредбу *make*. Пример једног *Makefile* фајла дат је у наставку, а у Прилогу Б на крају рада могуће је наћи *Makefile* који је коришћен за превођење решења овог рада.

INCDIR = -I./include -I./include/messenger -I./include/Poco

LIBDIR = -L./lib

LIBS = -loblomessenger -lPocoCrypto -lPocoJSON -lPocoFoundation

TARGET = ./bin/exe

SRC = ./src/main.cpp ./src/MqttWrapper.cpp ./src/GTWStorage.cpp

CC = g++

**$(**TARGET**)**: **$(**SRC**)**

**$(**CC**)** **$(**INCDIR**)** **$(**LIBDIR**)** -o **$(**TARGET**)** **$(**SRC**)** **$(**LIBS**)**

Како сада знамо принцип превођења апликација на *Linux* оперативном систему није тешко проширити причу за случај унакрсног превођења. И у овом случају *make* алат ће нам бити од велике користи. Једино у чему се унакрсно превођење разликује од нативног превођења јесте програмски преводилац. Поново користимо *g++*, међутим, овога пута он мора да зна како да преводи за *Raspberry Pi* платформу. Дакле, потребан нам је идентичан *g++* програмски преводилац оном који би се иначе покретао на *Raspberry Pi* рачунару, а да је притом могуће покренути га и на персоналном рачунару. Програми, који имају описану могућност, се у енглеској литератури називају *cross-compile toolchain* програми. Овакви програми преводе изворни код у сет инструкција које су јасне циљној платформи, у нашем случају *Raspberry Pi* рачунару. Потпуно јасно, излазни фајл који се добија превођењем није извршив на персоналном рачунару и могуће га је покретати само на *Raspberry Pi* платформи. *Toolchain* коришћен за унакрсно превођење у овом раду је   
*arm-linux-gnueabi-g++.*

# Предлог решења

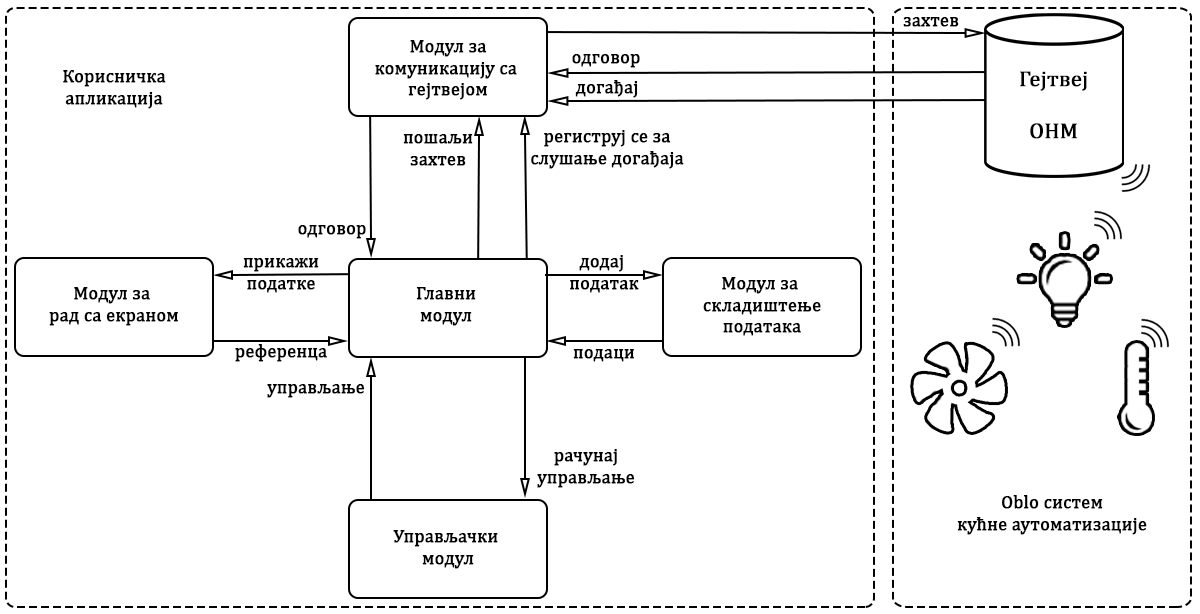
Развијена апликација се покреће на *Raspberry Pi* рачунару и са *Oblo* системом комуницира по MQTT протоколу користећи *Ethernet* мрежу. Писана је у С++ програмском језику и за интеракцију са корисником поседује графичко окружење. Апликација треба да подржи рад у два режима, ручном и аутоматском. У ручном режиму рада корисник може самостално да задаје вредности којима управља димерима осветљености. У аутоматском режиму рада корисник може да задаје само жељену референцу осветљености, а управљања димерима аутономно одређује апликација. У сваком тренутку, на екрану се исписују тренутна управљања димерима и тренутна вредност са сензора осветљености.

Главни проблем при реализацији апликације односи се на управљање димерима. Међутим, како ће се апликација покретати као независни софтвер на засебном хардверу, у пратеће проблеме можемо сврстати комуникацију са гејтвејом и управљање графичким интерфејсом. Архитектура предложеног решења састоји се из 5 модула. Сваки од модула има енкапсулирану логику и јединствену улогу, а са осталим модулима комуницира преко позивних функција и параметара. Комуникација модула диктирана је променама у спољашњој средини. Спрегом свих модула реализује се жељена регулација.

Модули од којих се састоји апликација су:

* + - Модул за комуникацију са гејтвејом
    - Модул за складиштење података
    - Модул за управљање
    - Модул за рад са екраном
    - Главни модул

Апликација се извршава у вишенитном окружењу (енгл. *multithreading*) и треба оптимално да користи доступне ресурсе. У случају приступања дељеним ресурсима неопходна је синхронизација нити. На слици 4.1 дат је графички приказ односа модула од којих се састоји предложено решење.



Слика 4.1: Архитектура предложеног решења

Модул за комуникацију са гејтвејом

Улога модула за комуникацију је посредство између OHM*-*аи корисничке апликације. То је једини модул који има могућност комуникације са OHM*-*ом и сви остали модули обраћање OHM*-*у обављају уз помоћ овог модула. Комуникација је двосмерна и то тако да се при слању сваког захтева добија одговарајући одговор. Како не би дошло до мешања одговора, онемогућено је слање новог захтева у интервалу између слања претходног захтева и пристизања одговора на претходни захтев. Улога овог модула је и регистовање догађаја (енгл. *events*), порука којима OHM обавештава о променама које се дешавају у спољашњем систему. Модул за комуникацију врши обраду садржаја догађаја и информације прослеђује објектима који су се за то пријавили.

Сва комуникација одвија се по MQTTпротоколу. Садржај порука диктиран је синтаксом позивних функција OHM-а и форматиран је по JSON формату. Овај модул свој рад заснива на *oblomessenger* програмској библиотеци са којом је уско повезан.

Модул за складиштење података

Улога модула за складиштење података јесте пуко меморисање. Овај модул памти јединствене идентификационе бројеве сензора и димера као и све промене управљања, референце и остварених вредности осветљености током рада апликације. Промене поменутих величина памте се упарене са временом које је протекло од покретања апликације до њихове промене. Складиштење података реализује се додавањем у уланчану листу. На захтев осталих модула, модул за складиштење података може доставити меморисане податке. По жељи корисника, овај модул омогућава и уписивање пристиглих података у фајл на диску.

Модул за управљање

Улога модула за управљање јесте рачунање управљања које се шаље димерима. Модул за управљање реагује на сваку промену референтне или мерене вредности осветљености и врши прерачунавање управљања. О промени референтне вредности обавештава га модул за рад са екраном, а о промени мерене вредности обавештава га модул за комуникацију. За израчунавање управљања овај модул поседује засебну нит. У периоду ручног режима рада апликације, ова нит је блокирана како не би трошила процесорско време. При преласку у аутоматски режим рада, промени референтне или мерене вредности, нит се одблокира и наставља свој рад.

Модул за рад са екраном

Улога модула за рад са екраном је интеракција са корисником. Овај модул контролише графичко корисничко окружење, GUI (*Graphical user interface*). Од корисника прима информације попут одабира режима рада, жељене референце или жељеног управљања и те информације прослеђује одговарајућим модулима, а на екрану приказује остварене вредности и тренутна управљања. Реализација GUI-а заснована је на *gtkmm* програмској библиотеци.

Главни модул

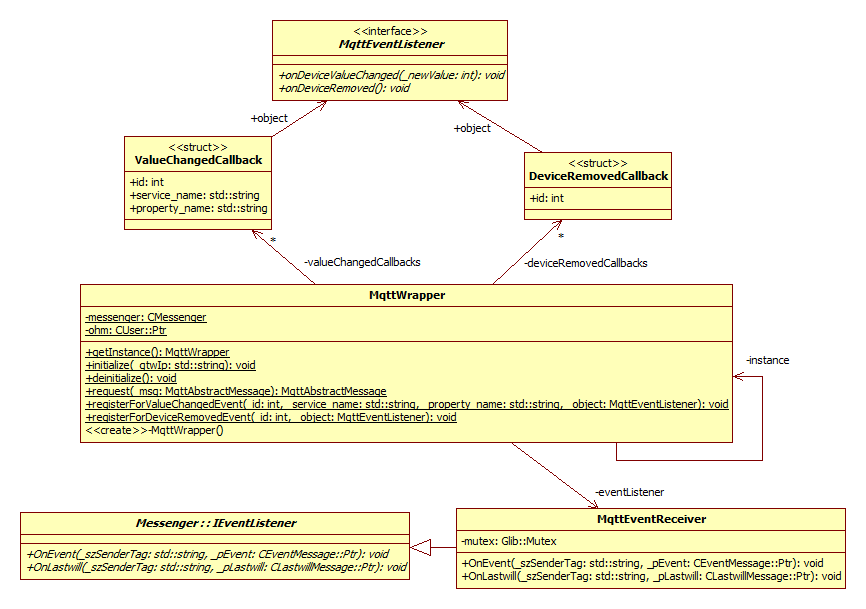
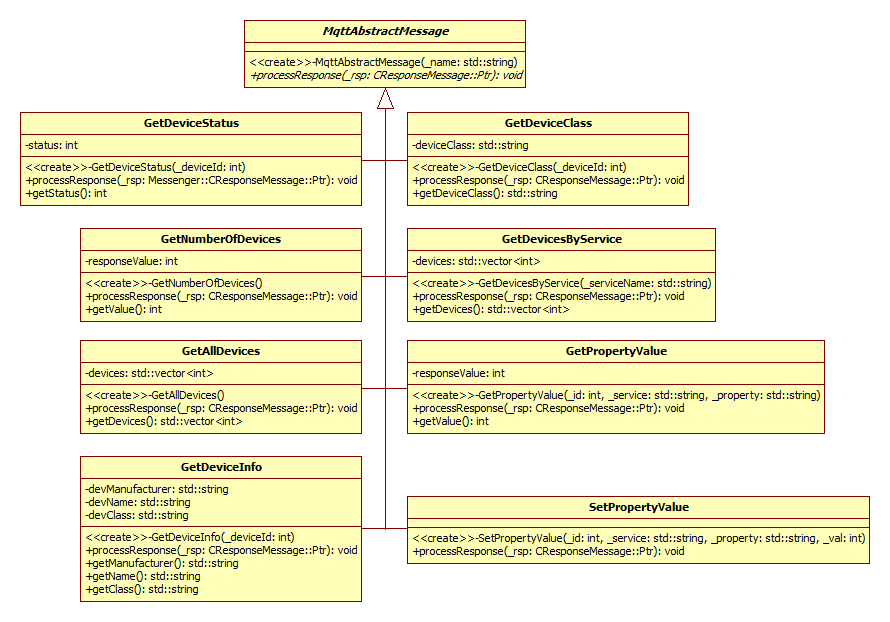
Примарна улога главног модула је контрола рада других модула. Он је први модул који се покреће и на самом почетку рада апликације његов задатак је иницијализација осталих модула. Током извршавања апликације улога главног модула је детекција и опоравак од евентуалних неправилности.

# Имплементација решења

Сваки од модула поменутих у претходном поглављу састоји се из више класа. Однос класа је такав да једна, главна, класа сваког модула представља сервис тог модула и имплементирана је по *Singleton* пројектном обрасцу. На овај начин обезбедили смо да може постојати само једна инстанца сваке од сервисних класа и омогућили знатно бољу контролу иницијализације тих класа. Додатна погодност овакве имплементације је лак приступ сервисним класама из осталих делова изворног кода. Остале класе можемо сматрати помоћним. Њихове улоге су разнолике, али им је заједничко да од њиховог стања зависи понашање система. Дакле сервисне класе познају процедуре функционисања система и оперирају над помоћним класама, док помоћне класе садрже информације о стању система и диктирају динамику рада.

## Модул за комуникацију са гејтвејом

Апликација свој рад уско везује за процесе који се одвијају на гејтвеју и сви модули за свој рад захтевају повезаност са OHM-ом. Зато, први модул који треба развити је управо модул за комуникацију са гејтвејом. Услуге овог модула доступне су упоребом *MqttWrapper* сервисне класе. За свој рад, модул дефинише још неколико типова. *MqttEventListener* представља интерфејс који мора имплементирати свака класа чији рад зависи од догађаја које обрађује модул за комуникацију са гејтвејом. За складиштење података о објектима који су се пријавили за праћење догађаја користе се структуре *DeviceRemovedCallback* и *ValueChangedCallback*. Даље, за обрађивање података о догађајима, модул користи и *MqttEventReceiver* класу која имплементира *Messenger::IEventListener* интерфејс. Овај интерфејс декларисан је унутар *oblomessenger* програмске библиотеке, која је развијена од стране „RT-RK“ и представља скуп објеката за комуникацију са OHM-ом. Све поруке које се могу послати гејтвеју упаковане су у посебне објекте, а формат објеката прописује *oblomessenger* библиотека. *MqttAbstractMessage* представља апстрактну класу која садржи генерализоване особине свих порука. Конкренте поруке проширују *MqttAbstractMessage.* За потребе промене вредности неке променљиве унутар OHM-а или управљање понашањем уређаја користе се објекти класе *SetPropertyValue.* Објекти класе *GetPropertyValue* служе за захтевање информације о тренутном стању неке променљиве у OHM-у. *GetAllDevices* омоћуава захтевање информације о свим уређајима који су тренутно повезани са *Oblo* системом. Поред поменутих, имплементиране су и друге класе конкрених порука.Ради лакшег разумевања односа класа, на слици 5.1 приказан је UML модел модула за комуникацију.

Слика 5.1: UML модел модула за комуникацију са гејтвејом

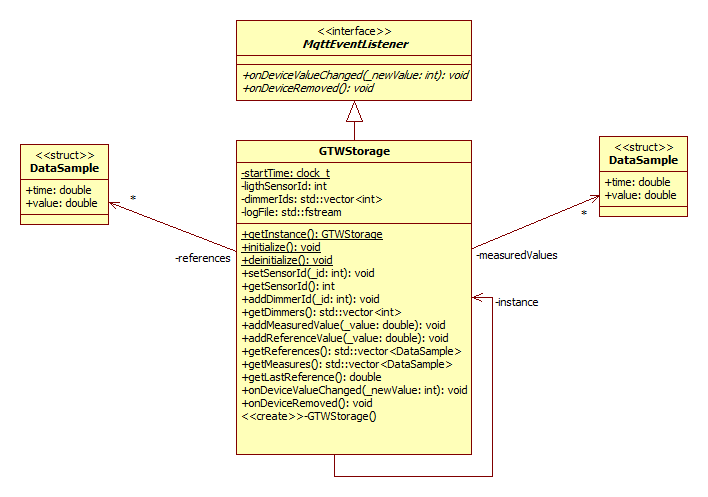
Рад модула за комуникацију са гејтвејом почиње поступком пријављивања корисничке апликације OHM-у. Поступак пријављивања као и позиви који се за пријављивање користе описани су у *oblomessenger* библиотеци. Пријављивање се одвија у *MqttWrapper::initialize()* статичкој методи и неопходно је пре слања иједне поруке OHM-у. Уколико корисник ипак покуша да пошаље поруку без претходне иницијализације, апликација пријављује грешку и завршава рад. Поступак се састоји из више фаза. Прво, *MqttWrapper* покушава да се пријави са постојећим корисничким именом и лозинком, уколико је у питању прво пријављивање корисничко име и лозинка су насумичне вредности. Ако то не успе, апликација се пријављује OHM-у са подразумеваним налогом и захтева нове креденцијале. По успешном добијању нових података, одјављује се и поновно пријављује са новим корисничким именом и лозинком које додатно уписује у фајл. Након пријављивања, остварена је веза између корисничке апликације и OHM-а која ће бити отворена до краја рада апликације. По поновном покретању апликације, понавља се поступак пријављивања.

За потребе слања порука користе се *MqttAbstractMessagе* класа, класе конкретних порука и *MqttWrapper::request()* статичка метода. Пре него опишемо поступак слања порука и обраде одговора, позабавимо се структуром поменутих класа. *MqttAbstractMessagе* јесте апстрактна класа, што значи да не може постојати ниједан објекат те класе већ само објекти класа које су изведене из ње. Из ње се изводе класе конкретних порука: *SetPropertyValue, GetPropertyValue, GetAllDevices, GetDeviceStatus, GetDeviceClass…* Ове класе предвиђају меморијски простор за смештање података и о одлазној поруци и о одговору на ту поруку. На овај начин омогућили смо лако упаривање захтева и одговора, али и то да је свака класа задужена за обраду свог одговора. *MqttAbstractMessagе* класа врши генерализацију и прописује *processResponse()*виртуелну методу, а све класе конкретних порука дају своје дефиниције ове методе у складу са форматом поруке која им стиже као одговор. Сада долази на ред *MqttWrapper::request()* метода. Ова метода као аргумент прима објекат класе *MqttAbstractMessage.* По принципима објектно-орјентисаног програмирања, на месту родитељске класе може се појавити објекат класе која је изведена из ње па ова метода уме да пошаље било коју конкретну поруку. Задатак *MqttWrappe::request()* јесте да објекат поруке, који јој је стигао као аргумент, пошаље OHM-у. Потом, сачека одговор и проследи одговор истом оном објекту поруке који је претходно послала OHM-у. Прослеђивање се врши позивом *processResponse()* методе*.* У овој методи врши се издвајање информација које је OHM послао и смештање тих информација у меморијски простор намењен за одговор на поруку. Информације о одговору се могу дохватити јавним методама објеката конкретних порука. Дакле, класа која жели да пошаље поруку треба да направи нови објекат те поруке и напуни део меморије намењен за одлазне информације. Тако напуњен објекат треба да проследи *MqttWrappe::request()* методи која, на основу информација о одлазној поруци, пошаље захтев OHM-у и одговор врати објекту поруке. Објекат поруке обради одговор и напуни део меморије намењен за информације о одговору. Тако напуњен објекат поруке враћа се класи која је и започела процес комуникације чиме су јој доступни и одговори на њен захтев.

Поред слања порука и обраде одговора, модул за комуникацију обавља и ослушкивање и обраду догађаја које шаље гејтвеј. Информација о догађају долази у истом облику као и регуларна порука, дакле, путем MQTT протокола и форматирана у JSON формату. Улога *MqttEventReceiver* класе јесте да изврши издвајање података из сваке поруке о догађају и обавести евентуалне објекте који су се пријавили за слушање информација о том типу догађаја. Постоји више класа догађаја у *Oblo* систему, али су нама од интереса једино догађаји који су везани за уређаје. То су *EventDevicePropertyValueChanged* и *EventDeviceRemoved.* У случају пристизања поруке о догађају другог типа, ништа се не дешава. За слушање догађаја може да се пријави било који обејкат чија класа имплементира *MqttEventListener* интерфејс позивом *MqttWrapper::registerForValueChangedEvent()*. Ова метода као параметре прима идентификациони број уређаја, назив величине чија се промена слуша као и референцу на објекат који се пријављује за слушање догађаја. При сваком позиву ове методе ствара се нова инстанца *ValueChangedCallback* структуре, попуњава се прослеђеним аргументима и тако попуњена структура додаје се у низ *valueChangedCallbacks.* Овај низ памти сва пријављивања за праћење догађаја. При пристизању поруке новом о догађају окида се *onEvent()* метода *MqttEventReceiver* класе у којој се врши издвајање инфорамција о догађају. Наредни задатак ове методе је да прође кроз низ *valueChangedCallbacks* и упореди издвојене информације о догађају са упамћеним пријавама. Уколико постоји поклапање, користи се упамћена референца на објекат и модул за комуникацију позива *onDeviceValueChanged()* методу тог објекта, а нову вредност прослеђује као параметар. У овој методи, објекат који се пријавио за праћење догађаја, дефинише скуп операција које се извршавају при сваком појављивању догађаја.

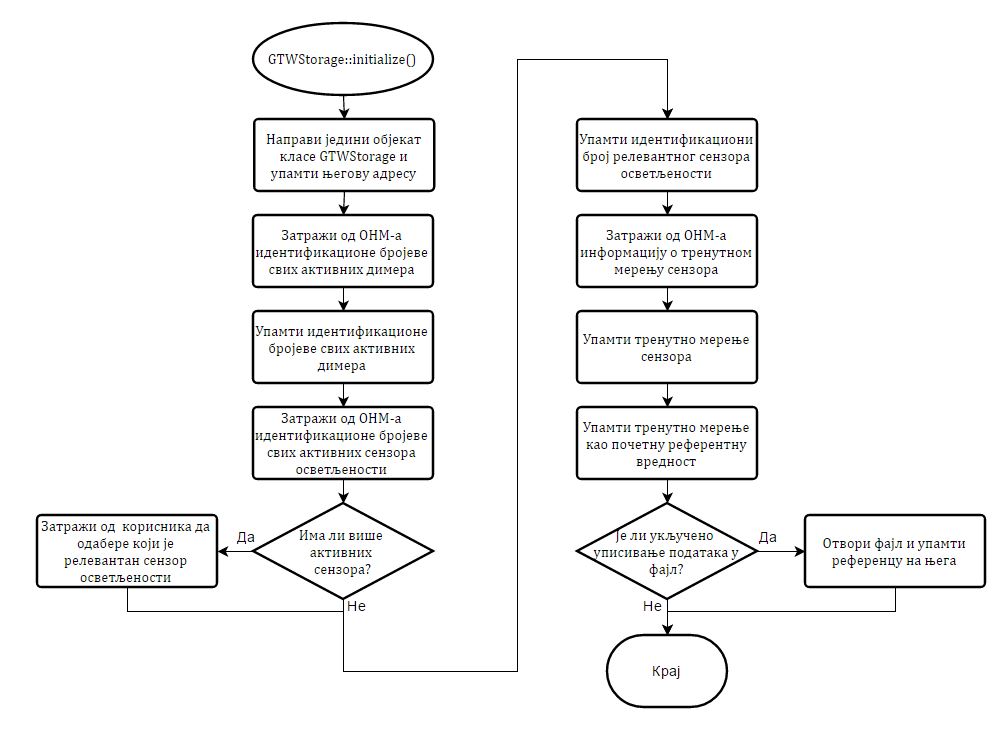
Од преосталих метода *MqttWrapper* сервисне класе, важна је још и *deinitialize()* статичка метода. Ова метода се позива на крају рада апликације и њена улога је одјављивање корисничке апликације и прекидање везе са OHM-ом. Ова метода врши и ослобађање заузете меморије.

## Модул за складиштење података

Модул за складиштење података свој рад остварује кроз *GTWStorage* сервисну класу. Како је промена мерене вредности осветљености од кључног интереса за овај модул, *GTWStorage* сервисна класа имплементира *MqttEventListener* интерфејс. Поред поменуте сервисне класе, модул за складиштење података прописује и стуктуру *DataSample* којом је декларисана форма података које модул складишти. На слици 5.2 приказан је UML модел модула за складиштење података.

Слика 5.2: UML модел модула за складиштење података

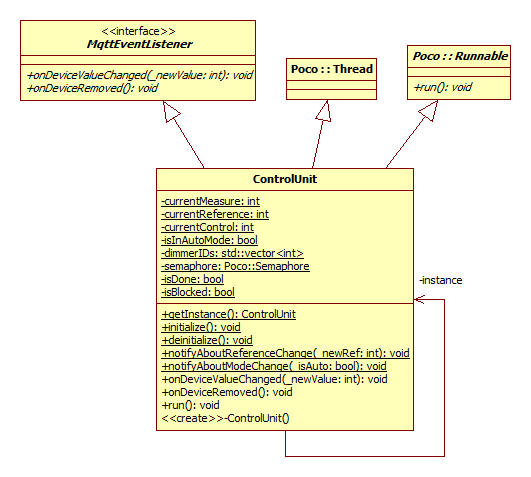
Пре покушаја смештања ма каквих података у модул, неопходна је иницијализација *GTWStorage* сервисне класе. Иницијализација се може обавити позивом *initialize()* статичке методе којом се ствара једини објекат ове класе. Улога *initialize()* методе је и добављање информација о тренутно доступним димерима и сензорима осетљености унутар *Oblo* система. Идентификациони бројеви свих димера се одмах памте. У случају постојања више сензора метода ће од корисника захтевати да одабере један чије ће се мерење сматрати релевантним и чији ће се идентификациони број упамтити. Како рад осталих модула од самог старта захтева вредност мерења сензора, наредна операција коју метода извршава је добављање и памћење почетне вредности мерења. Да на почетку рада апликације не би било скока у управљању, почетна мерена вредност проглашава се за тренутну референтну вредност те је модул и памти. Уколико је корисник укључио опцију уписивања података у фајл, на самом крају *initialize()* методе се отвара фајл и памти његова референца. На слици 5.3 приказан је дијаграм тока *initialize()* методе.

Слика 5.3: Дијаграм тока *initialize()* методе

За потребе додавања нових вредности у складиште, класа *GTWStorage* прописује методе *addMeasuredValue*() односно *addReferenceValue()*. Позивом ових метода ствара се нови примерак *DataSample* структуре, у њега се уписује време промене, мерено од почетка рада апликације, и вредност промене. Тако попуњена структура убацује се у вектор упамћених вредности. *GTWStorage* класа декларише и методе које се користе за дохватање података. Доступне су методе које враћају: вектор свих референтних вредности, вектор свих мерених вредности, вектор свих идентификационих бројева доступних димера и идентификациони број регистрованог сензора осветљености. За уништавање модула користи се *deinitialize()* статичка метода која затвара евентуално отворени фајл и ослобађа меморију.

## Модул за управљање

Модул за управљање реализован је *ControlUnit* сервисном класом. Како архитектура решења предвиђа да се управљање рачуна у засебној нити, ова класа проширује *Poco::Thread* класу и имплементира *Poco::Runnable* интерфејс. Додатно, рад *ControlUnit* класе зависи и од тренутне мерене вредности осветљености па класа имплементира и *MqttEventListener* интерфејс. На слици 5.4 дат је UML модел модула за управљање.

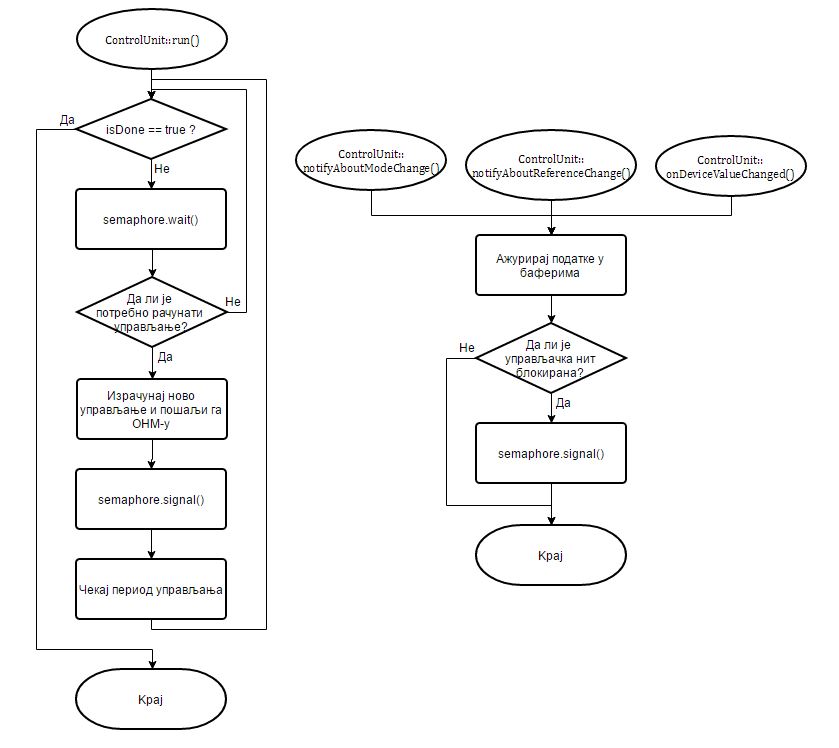


Слика 5.4: UML модел модула за управљање

За иницијализацију *ControlUnit* сервисне класе, као у и случају осталих сервисних класа, доступна је *initialize()* статичка метода. Позивање ове методе неопходно је пре првог коришћења *ControlUnit* класе*.* Улога *initialize()* методе је заузимање потребне меморије за рад сервиса, захтевање почетне мерене и референтне вредности од модула за складиштење података те попуњавање бафера њима. У овој методи врши се и пријављивање *ControlUnit* класе за слушање догађаја о сензору осветљености. На самом крају методе, покреће се нит у којој ће се рачунати управљање.

Израчунавање управљања врши се у *run()* виртуелној методи. Ова метода прописана је *Poco::Runnable* интерфејсом и извршава се све време рада модула за управљање. Да бисмо обезбедили да се рад ове методе понавља у неповрат, читава логика уоквирена је *while* петљом, а ако услов за излаз из ње користимо *isDone* логичку променљиву која ће сигнализирати да је потребно завршити рад модула. Блокирање нити обавља се помоћу бинарног семафора. Свака итерација *while* петље састоји се из пар етапа. У почетку, позива се *wait()* метода семафора за блокирање. Потом, врши се провера је ли је потребно рачунати ново упраљање. Проверава се да ли је апликација у ручном режиму рада, јесмо ли постигли референтну вредност у претходној итерацији или јесу ли управљања већ на максималној или минималној вредности те нема смисла рачунати ново управљање. Уколико је иједан од ова два услова искуњен, одмах се прелази на извршавање следеће итерације *while* петље. Поново ће се позвати *wait()* метода семафора за блокирање и, како је у питању бинарни семафор, нит ће се овог пута блокирати. Нит остаје блокирана до тренутка када неко позове *signal()* методу семафора за блокирање. Уколико извршавање прође проверу и донесе се одлука да је потребно рачунати ново управљање, прелази се на рачунање новог управљања. Прерачунато управљање се шаље OHM-у на извршавање. Наредни корак јесте позивање *signal()* методе семафора за блокирање чиме смо обезбедили да се нит неће блокирати при следећем позиву *wait()* методе тј. на првом кораку следеће итерације. На крају алгоритма управљачка нит се успављује на период управљања. Важну улогу у целом алгоритму имају и *notifyAboutReferenceChange(), notifyAboutModeChange()* и *onDeviceValueChanged()* методе *ControlUnit* сервисне класе. Ове методе позивају остали модули и на тај начин сигнализирају да је дошло до неке промене на систему и да је потребно ново прерачунавање управљања. Задатак им је да попуне бафере новим вредностима и одблокирају управљачку нит уколико је она у том тренутку блокирана позивањем *signal()* методе семафора за блокирање. Када управљачка нит уђе у следећу итерацију, доносиће одлуке о управљању на основу најновијих вредности које су јој уписане у бафере. На слици 5.5 дат је дијаграм тока алгоритма за управљање.

Осврнимо се сада на рачунање управљања. Прави инжењерски приступ би био да покушамо да смимимо преносну функцију система од управљања димерима до релизованог осветљења. На основу ње пројектујемо оптимални контролер и затворимо повратну спрегу. Међутим, отежавајућа околност у овом приступу је чињеница да би се преносна карактериситка значајно мењала у зависности од привременог распореда људи и објеката у просторији. Додатни проблем прави чињеница да је читав систем окренут уштеди енергије, па сензор осветљености има јако дугу периоду одабирања, чак 2 секунде. Да ствар буде још и гора, сензори осветљености раде са мртвом зоном па о променама које су мање од 200 Lux и не обавештавају *Oblo* систем. Због описаних проблема, одлука је била да се управљање реализује што простије и робусније могуће, а да цена за то буде брзина система. Реализован је јако прост алгоритам који врши пуко упоређивање тренутне мерене вредности и тренутне референце. Уколико је њихова разлика мања од 200 Lux не рачуна се ново управљање и нит се блокира. Уколико је референца већа од тренутног мерења за више од 200 Lux управљање се повећева за 2%, а уколико је референца више од 200 Lux мања од тренутног мерења управљање димерима се смањује за 2%. Затим, систем чека 2,5s како би сензор осветљености доставио нову вредност и поступак се понавља. Додатно, алгоритам је заштићен од ситације у којој би јединична промена управљања мењала излаз за више од 200 Lux и тада би систем у свакој итерацији скакао изнад те испод референтне вредности до бесконачности. У тим ситуацијама бира се управљање које остварује вредност која је ближа референтној. Овако реализовано управљање омогућава систему да споре промене, на пример излазак и залазак Сунца, компензује у једној до две итерације. У случају скоковитих промена референце, време одзива система је најдуже 125 секунди.

Слика 5.5: Дијаграм тока алгоритма за управљање

На крају рада апликације, нужно је позивање *deinitialize()* статичке методе. У овој методи се поред ослобађања меморије врши и сигнализирање управљачкој нити да заврши рад постављањем *isDone* логичке променљиве на истиниту вредност. Уколико бисмо покушали да завршимо рад апликације без позивања ове методе, управљачка нит би остала да се покреће у позадини и исход је непредвидив.

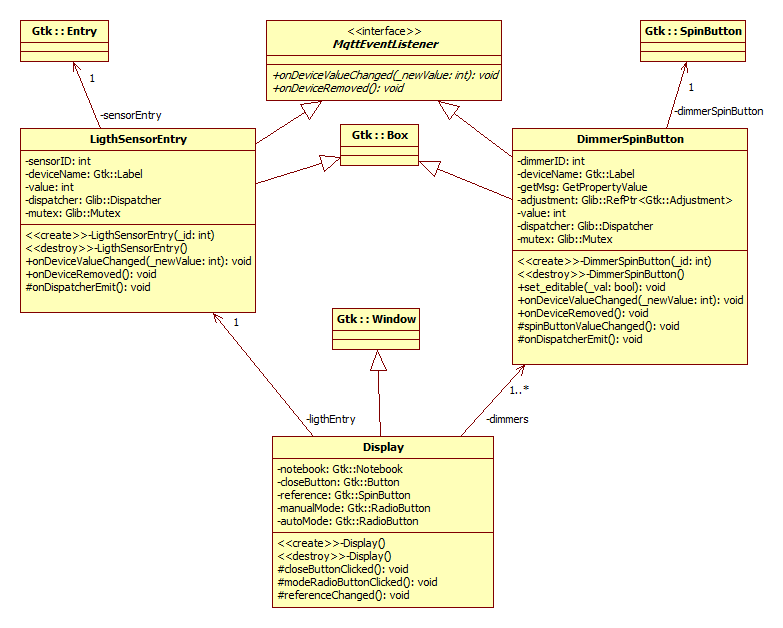
## Модул за рад са екраном

Управљање графичким корисничким окружењем, модул за рад са екраном реализује помоћу три класе. Главна класа овог модула именована је *Display*. За разлику од претходних модула, главна класа сада није импленетирана по *Singleton* пројектном обраску и нећемо је сматрати сервисном. Улога *Display* класе је да управља прозором апликације и држи остале елементе на окупу што она оставрује проширивањем *GTK::Window* класе. Како се модул за рад са екраном ослања на GTK и *gtkmm* библиотеке за рад са графиком, све класе проширују и обилују њиховим елемeнтима.

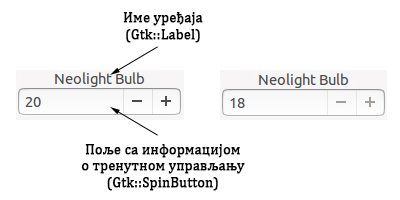
Помоћне класе овог модула су *DimmerSpinButton* и *LigthSensorEntry*. Oне су задужене за приказивање информација од димерима светлости и сензору осветљености. Како само име сугерише, прва класа користи функционалности *Gtk::SpinButton* класе, а друга *Gtk::Entry* класе. Да би поред елемента за приказивање вредности држале и име уређаја на окупу помоћне класе наслеђују *Gtk::Box* класу, а међу својим атрибутима садрже објекте поменутих графичких класа. Додатно, све помоћне класе имплементирају *MqttEventListener* интерфејс како би могле да се региструју за слушање догађаја у вези са уређајима за које су везане. UML модел модула за рад са екраном приказан је на слици 5.5.

Пре него што пређемо на имплементацију самих класа модула, било би корисно да се позабавимо и објаснимо концепт догађаја и сигнала којим се већина GUI класа води. Сигнал представља информацију да се неки од догађаја десио. Догађаји су најчешће изазвани акцијом корисника на пример: корисник је кликнуо на дугме, уписао нову вредност у поље, спустио прозор... Свака класа *gtkmm* библиотеке декларише скуп сигнала које може да пошаље као и догађаје који условљавају слање сваког сигнала. На овај начин апликација добија информацију о асинхроној промени из спољашње средине. Он што је програмеру на располагању је да повеже неки од сигнала са функцијом која ће се извршити када се сигнал емитује. Ове функције се уобичајено називају *callback* функцијама и најчешће се имплементирају као методе објекта који емитује сигнал.

Корисно је напоменути и да се за потребе исцртавања графичких елемената на екрану ствара посебан ток извршаваља програма, дакле, посебна процесорска нит која се назива GUI нит. Њен задатак јесте да при свакој промени изврши освежавање графичког окружења и поновно исцртавање сви графичких елемената. Добра програмерска пракса је да GUI нит не буде задужена нити за какаве друге операције тј. да се у контексту GUI нити не врши дужа обрада података. Такође пракса је да се методе које су задужене за ажурирање графичког садржаја позивају искључиво у контексту GUI нити.

Слика 5.5: UML модел модула за рад са екраном

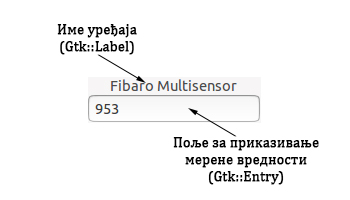
Опис класа модула за рад са екраном почећемо од помоћних класа. *DimmerSpinButton* класа писана је са идејом да представља графички елемент контроле на димером. Приликом учитавања графичког окружења, за сваки од детектованих димера у *Oblo* систему ствара се по једна инстанца класе *DimmerSpinButton.* Увезивање се обавља у контруктору. Тада се памти идентификациони број димера за који је класа везана. Потом, од *Oblo* система се захтевају подаци о димеру, издваја име димера и прави графичка лабела. Лабела се додаје објекту класе *DimmerSpinButton* који се управо прави. Наредни корак јесте захтевање од *Oblo* система информације о тренутном управљању димером. Та вредност се, као почетна, уписује у *dimmerSpinButton* поље и оно се такође додаје објекту класе *DimmerSpinButton* чији конструктор се извршава. Оно што још остаје да се уради у конструктору је регистрација за слушање догађаја у вези са повезаним димером. Како *DimmerSpinButton* имплементира *MqttEventListener* интерфејс регистрација се обавља позивом методе *MqttWrapper::registerForValueChangedEvent()*. Графички приказ објекта ове класе приказан је на слици 5.6.

Слика 5.6: Графички приказ објекта *DimmerSpinButton* класе у ручном режиму (лево) и у аутоматском режиму (десно)

За потребе одазивања на промену уписане вредности, *DimmerSpinButton* класа има дефинисану *spinButtonValueChanged()* методу која је повезана са *signal\_value\_changed* сигналом. При свакој промени уписане вредности шаље се поменути сигнал и извршава *callback* метода. Задатак ове методе јесте да узме вредност коју је корисник уписао, узме идентификациони број димера за који је објекат везан, направи и попуни објекат класе *SetPropertyValue* и проследи га *MqttWrapper::request()* методи. Тиме је новоуписана вредност достављена OHM-у на извршавање. Ажурирање приказане вредности у случају када промена управљања димером долази изван корисничке апликације користи се *onDeviceValueChanged()* метода.

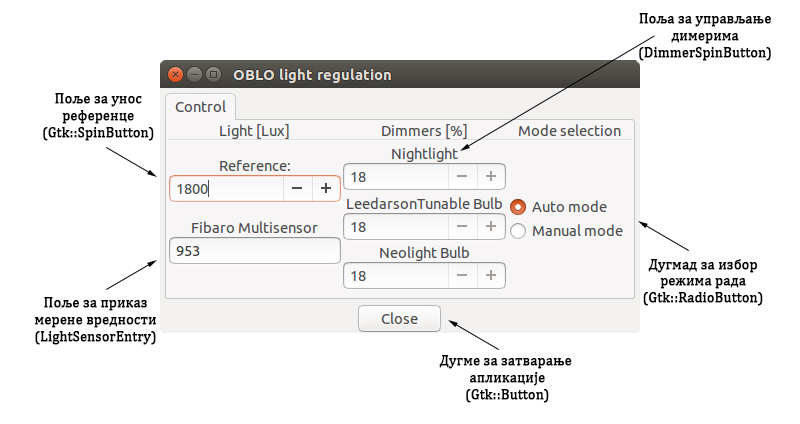
*DimmerSpinButton* класа своје понашање мења у зависности да ли се апликација налази у аутоматском или ручном режиму. У ручном режиму, уписивање вредности у поље је дозвољено. У аутоматском режиму, није омогућено уписивање вредности у поље већ се та вредност програмски ажурира у складу са променом управљања. Разлика између изгледа графичког објекта у ручном и аутоматском режиму приказана је на слици 5.6.

Имплементација *LigthSensorEntry* класе је нешто једноставнија. Објекат ове класе само приказује вредност која стиже од сензора осветљености. Слично *DimmerSpinButton* класи, у конструктору се врши повезивање графичког елемента са сензором осветљености. Памти се идентификацион број сензора, добављају информације, издваја име и ствара нова лабела која се додаје објекту за који је конструктор позван. У конструктору се врши и регистрација за слушање догађаја који су везани за сензор. Графички приказ објекта ове класе дат је на слици 5.7.

Слика 5.7: Графички приказ објекта класе *LigthSensorEntry*

Графички објекат ове класе не мења понашање у зависности од одабира режима рада. Унос вредности у поље никада није дозвољен, а ажурирање садржаја се врши искључиво програмски, из методе *onDeviceValueChanged().*

Пошто смо се упознали са свим елементима који учествују у раду *Display* класе, можемо да пређемо и на њену имплементацију. Ова класа проширује *Gtk::Window* класу и садржи прегршт графичких елемената, држи елементе на окупу и управља њиховим понашањем. У конструктору ове класе прво се ствара дугме за затварање апликације, *closeButton,* које је типа *Gtk::Button*. Сигнал *signal\_clicked* дугмета за затварање апликације се одмах повезује са *closeButtonClicked() callback* методом чија је једина улога да сакрије прозор и заврши извршавање GUI нити. Следи иницијализација *DimmerSpinButton* објеката за сваки од димера у систему и *LigthSensorEntry* објекта за одабрани сензор осветљености. Следећи корак јесте прављење поља у које ће моћи да се унесе жељена референца. Поље је типа *Gtk::SpinButton* и сигнал *signal\_value\_changed* овог поља се одмах увезује са *referenceChanged()* методом. Улога ове методе је да при свакој промени референце обавести модул за управљање, позивом *ControlUnit::notifyAboutReferenceChange()* методе, и да промену референце проследи модулу за складиштење података. У конструктору се још врши и стварање графичког објекта за одабир режима рада. Овај објекат је типа *Gtk::RadioButton* и састоји се од два дугмета: *autoMode* за одабир аутоматског режима рада и *manualMode* за одабир ручног режима рада. Класа *Gtk::RadioButton* прописује да само један од два режима може бити изабран. Сигнал *signal\_clicked* објекта за одабир режима рада се повезује са *modeRadioButtonClicked* методом чија је улога да промени понашање одговарајућих објеката, као што је већ описано, и да обавести модул за управљање да је дошло до промене режима позивањем методе *ControlUnit::notifyAboutModeChange().* На слици 5.8 приказан је изглед графичког објекта класе *Display*. То је уједно и главни прозор из којег се управља радом читаве апликације.

Слика 5.8: Изглед главног прозора апликације

## Главни модул

Главни модул реализован је као уобичајена *main* функција. Таквом реализацијом обезбедили смо да извршавање апликације увек почиње овим модулом. Главни модул прво врши иницијализацију свих сервинсних класа *(MqttWrapper, GTWStorage, ContolUnit)* позивањем *initialize()* методе за сваку од њих. У случају неуспеле иницијализације макар једне од класа, модул исписује обавештење о разлогу неуспеха и завршава рад апликације. Након тога, главни модул ствара објекат класе *Display* и прослеђује га статичкој методи *Gtk::Main::run().* Ова метода као аргумент прима објекат класе *Gtk::Window*, а класа *Display* проширује класу *Gtk::Window*, и њен задатак јесте да покрене нову нит која ће бити задужена за исцртавање графичког окружења и његових елемената. GUI нит извршаваће се све до тренутка када корисник затвори главни прозор. Од позива *Gtk::Main::run()* методе надаље извршавање главног модула је блокирано и наставља се по завршетку рада GUI нити. Тада је завршен рад апликације и улога главног модула је деиницијализација и гашење сервисних класа позивањем *deinitialize()* методе за сваку од њих.

# Закључак

# Литература

[1] *Oblo living*

Нови Сад: RT-RK Institute for Computer Based Systems

<http://www.obloliving.com>

[2] *MQTT Essentials*

Landshut: HiveMQ

<http://www.hivemq.com/blog/mqtt-essentials>

[3] *УВОД - Raspberry Pi рачунар*

Нови Сад: Факултет техничких наука

<http://www.rt-rk.uns.ac.rs/sites/default/files/materijali/lab/uvod_0.pdf>

[4] *Poco C++ libraries Documentation*

Maria Elend: Applied Informatics Software Engineering GmbH

<http://pocoproject.org/documentation>

[5] *Programming with gtkmm 3*

GNOME

<https://developer.gnome.org/gtkmm-tutorial/stable/index.html.en>

[6] *How* *X10 works*

SmartHomeUSA

<http://www.smarthomeusa.com/how-x10-works>

[7] *A Simple Makefile Tutorial*

Waterville: Colby College

<http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor>

[8] *Home Automation buying guide*

CNET

<http://www.cnet.com/news/smart-home-buying-guide-home-automation>

[9] *About Z-Wave Technology*

Z-Wave Alliance

<http://z-wavealliance.org/about_z-wave_technology>

[10] *What’s The Difference Between ZigBee And Z-Wave?*

Electronic Design

<http://goo.gl/oKpY7c>

# Прилог А: *Makefile* за превођење апликације

Помоћу овог *Makefile* фајла могуће је превођење апликације и за извршавање на персоналном рачунару и за извршавање на *Raspberry Pi* рачунару. У случају превођења за персонални рачунар користи се *g++* преводилац, а у случају унакрсног превођења *arm-linux-gnueabi-g++* унакрсни преводилац.

Превођење за персонални рачунар реализује позивањем наредбе make LightRegulationPC из терминала рачунара. А у случају превођења за *Raspberry Pi* потребно је позвати make LightRegulationPi. У оба случаја неопходно је да радни фолдер терминала буде фолдер у коме се налази изворни код апликације. Извршив фајл смешта се у *bin* фолдер.

INCDIR = -I./include -I./include/messenger -I./include/paho -I./include/Poco

LIBDIR = -L./lib

TARGET = ./bin/LightRegulationPC

CC = g++

INCDIR\_PI = -I./includePi -I./includePi/messenger -I./includePi/paho -I./includePi/Poco

LIBDIR\_PI = -L./libPi

TARGET\_PI = ./bin/LightRegulationPi

CROSS\_CC = arm-linux-gnueabi-g++

LIBS = -loblomessenger -lPocoCrypto -lPocoJSON -lPocoFoundation

SRC = ./src/main.cpp ./src/UserException.cpp ./src/MqttWrapper.cpp ./src/MqttAbstractMessage.cpp ./src/GTWStorage.cpp ./src/Display.cpp ./src/GUIHandlers.cpp

GTKMM\_TAG = `pkg-config gtkmm-3.0 --cflags --libs`

LightRegulationPC: **$(**SRC**)**

**$(**CC**)** **$(**INCDIR**)** **$(**LIBDIR**)** -o **$(**TARGET**)** **$(**SRC**)** **$(**LIBS**)** **$(**GTKMM\_TAG**)**

LightRegulationPi: **$(**SRC**)**

**$(**CROSS\_CC**)** **$(**INCDIR\_PI**)** **$(**LIBDIR\_PI**)** -o **$(**TARGET\_PI**)** **$(**SRC**)** **$(**LIBS**)** **$(**GTKMM\_TAG**)**