

Универзитет у Београду  
Електротехнички факултет



**ДИЗАЈН ИОТ СИСТЕМА ЗА АКВИЗИЦИЈУ, НАДГЛЕДАЊЕ  
И ОБАВЕШТАВАЊЕ ПАМЕТНИХ УРЕЂАЈА**

Дипломски рад

Ментор:

Проф. др Милан Поњавић

Кандидат:

Владимир Сивчев 0155/2015

Београд, септембар 2019.

# Садржај

<b>1. Увод.....</b>	<b>1</b>
<b>2. Опис система и постојећа решења.....</b>	<b>2</b>
<b>3. Захтеви за реализацију система .....</b>	<b>4</b>
3.1. Комуникација између апликације и уређаја .....	4
3.1.1. Протоколи на којима се заснива комуникација .....	4
3.1.2. Платформски независан протокол вишег нивоа .....	6
3.2. Функционални аспекти система .....	7
3.3. Коришћене технологије и библиотеке .....	8
3.3.1. Развојна картица <i>Arduino MEGA 2560</i> .....	9
3.3.1.1. Архитектура <i>ATmega2560</i> .....	10
3.3.1.2. Напајање .....	11
3.3.1.3. Улази и излази .....	11
3.3.1.4. Комуникација.....	13
3.3.1.5. Програмирање.....	13
3.3.2. <i>Arduino Ethernet Shield</i> .....	14
3.3.2.1. <i>Wiznet W5100</i> .....	15
3.3.2.2. Пинови и конектори .....	16
3.3.3. Корисничка апликација .....	17
<b>4. Опис рада система.....</b>	<b>19</b>
<b>5. Реализација система .....</b>	<b>33</b>
5.1. Архитектура система .....	33
5.2. Хардвер уређаја .....	34
5.2.1. Главни процесорски блок (ГПБ) .....	34
5.2.1.1. ГПБ као <i>PLC</i> .....	35
5.2.1.2. ГПБ као сервер.....	35
5.2.2. Периферни хардвер.....	36
5.3. Софтвер система.....	39
5.3.1. Софтвер на уређају .....	39
5.3.1.1. <i>Concepts</i> .....	39
5.3.1.2. <i>Core</i> .....	45
5.3.1.3. <i>Device</i> .....	48
5.3.1.4. <i>Suit &amp; Tests</i> .....	59
5.3.2. Детаљи протокола у комуникацији .....	61
5.3.3. Софтвер клијентске апликације.....	64
<b>6. Закључак.....</b>	<b>67</b>

# 1. УВОД

Овај документ представља дипломски рад на тему „Дизајн IoT система за аквизицију, надгледање и обавештавање паметних уређаја“. Као и други IoT (енгл. *Internet of Things*) системи, који представљају међуумрежавање физичких објеката (што се односи и на „повезане уређаје“ и на „паметне уређаје“), возила, зграда и других ствари са уграђеном електроником, софтвером, сензорима и конективношћу путем постојеће мрежне инфраструктуре која омогућава објектима да размењују податке са произвођачем, оператером и/или другим повезаним уређајима, тако и идеја овог пројекта се занима на могућности даљинског контролисања уређаја, праћења околине кроз аквизицију података, стварање шансе за директну интеграцију физичког света и рачунарских система што резултује повећање ефикасности, тачности и економске користи, уз смањење потребе за људском интервенцијом.

Уређај, који је продукт овог пројекта, је само инстанца општије класе сајбер-физичких система, који се може употребити у разноврсним околностима и технологијама, попут паметних мрежа, паметних кућа, регулације физичких појава у индустријским коморама, интеграције паметних делова за самовозеће аутомобиле и слично.

За реализацију пројекта неопходно је постојање две стране учесника. Наиме, једна страна учесника је управо поменути уређај чија је улога да опслужује све захтеве друге стране, а то су корисници (клијенти), којима је комплетан сет функционалности доступан кроз апликацију која се налази било на персоналном рачунару, или на паметном телефону (енгл. *smart phone*), подржавајући што више популарних платформи, а то су:

- *UWP (Universal Windows Platform)* – позната *Windows* платформа која укључује како *Desktop PC* уређаје, тако и *Windows Phone*, искључиво верзије 10, развијена од стране *Microsoft Corporation*;
- *Android* – платформа развијена од *Google Inc.* која је дизајнирана примарно за уређаје осетљиве на додир (енгл. *touchscreen*) као што су паметни телефони и таблет уређаји;
- *iOS* – платформа веома слична *Android* платформи, развијена од стране *Apple Inc.* искључиво за њихов хардвер, која тренутно покреће многе паметне телефоне и сличне уређаје, а то су *iPhone*, *iPad* и *iPod Touch*.

За реализацију уређаја коришћена је *Arduino* платформа која је *open source* и базирана на програмском језику *C++*, заједно са периферним хардвером неопходним за аквизицију, надгледање и обавештавање. Што се корисничке стране и апликације тиче, употребљена је *Microsoft .NET (C#)* технологија, конкретно *Cross-platform Xamarin Forms* како би се постигао ефекат да развијена апликација може да се покреће на свим поменутим платформама.

У поглављу 2 овог документа биће детаљније описан реализован систем. У поглављу 3 биће дефинисани основни захтеви и представљене функционалности система, као и објашњене технике и библиотеке које су коришћене у реализацији. У поглављу 4 биће детаљније објашњене смернице за коришћење система. У поглављу 5 биће приказана конкретна реализација система, као и то да ће бити приказана архитектура система и комплетна имплементација. У поглављу 6 биће дат закључак до сада урађеног система и биће приказане могућности за надоградњу и унапређење.

## 2. ОПИС СИСТЕМА И ПОСТОЈЕЋА РЕШЕЊА

Као што је до сада речено, уређај служи за аквизицију (прикупљање) података у виду физичких величина из околине у којој се налази, као и управљање над разним електронским, односно електричним елементима и другим уређајима. Предвиђен је да се користи за регулацију температуре, нивоа осветљености у просторији, да се обезбеди алармни систем, да се омогући праћење и надзирање свих параметара и активности у његовој непосредној околини, као и надоградња многих других регулационих система по жељи.

Приступ уређају врши се путем интернета кроз корисничку апликацију кроз коју је могућа комплетна контрола над уређајем. Овим се постиже надзирање читаве просторије у којој се сам уређај налази из било ког дела света где је приступ интернету омогућен. Под надзирањем, односно контролом, сматра се:

- Праћење стања свих активних сензора (као што је сензор температуре, покрета, осветљења, влажности ваздуха и слично);
- Ручна, тј. мануелна контрола над тренутно активним елементима (као што су звучници, сијалице, грејачи, вентилатори и слично – такође могу бити и комплекснији електронски уређаји који су преко неког интерфејса спрегнути са самим уређајем);
- Увођење аутоматске контроле помоћу подсистема за „догађаје“, односно увођење логике када и шта уређај треба да уради у одсуству човека (увођење аутоматског система за температурну регулацију, система за дворишно осветљење и слично).

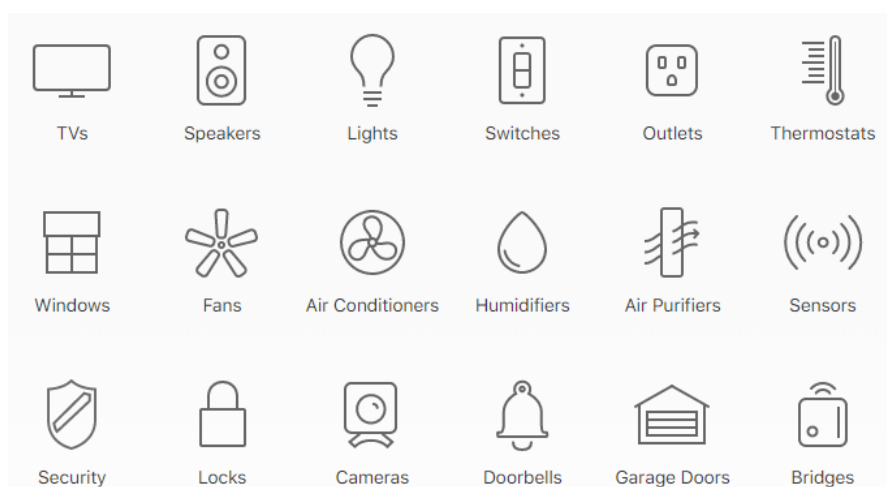
До сада поменута корисничка апликација за клијенте је само пример могуће (а у овом раду остварене) реализације друге стране система (клијентска страна). Сам систем је од почетка предвиђен тако да сама апликација буде у потпуности независна од уређаја и да може да се реализује у било којој жељеној технологији, као и да је могуће развијати не само једну, већ произвољно много апликација које би се интегрисале са уређајем. Дакле, уређај је центар овог система (и то не само по овом параметру), а клијентске апликације представљају само спрегу корисницима да на лак и једноставан начин приступају уређају и остварују жељене ефекте у виду контроле и надгледања. То такође доприноси могућност да други програмери, који би радили на систему, имају могућност да развију сопствени софтверски пакет, односно да развију сопствену апликацију коју желе да интегришу у систем.

У овом раду, као што је већ поменуто, апликација која је развијена јесте апликација коју је могуће инсталирати или на персоналним рачинарима, или на паметним телефонима. То у најосновнијем облику подразумева покретање инсталационог фајла (нпр. на *Windows* платформи *.appx*, на *Android* платформи *.apk* и слично), а такође дате апликације могуће је поставити и на продавнице апликација попут *Microsoft Store* или *Google Play* како би се корисницима омогућио још лакши начин да дођу до саме апликације.

Систем функционише попут пара „паметан сат – паметан телефон“ (енгл. *Smart Watch – Smart Phone*). Наиме, систем са паметним сатом и телефоном функционише тако да сам сат, који поред мерења времена, има уграђене и друге (хардверске) сензоре који прикупљају податке а затим путем *Bluetooth* комуникационог канала размењује податке са апликацијом која се налази на паметном телефону и на тај начин приказује кориснику жељене податке (брзина откуцаја срца, пулс, калорије и слично). Сам корисник такође може сатом да управља преко апликације (наравно уколико је сат таквог типа да у себи има подешавања и контролабилне, било хардверске или софтверске компоненте). Систем у овом пројекту састоји се од веома сличних карактеристика, али намена самог уређаја може бити разноврснија (али често предефинисана), најчешће да се пасивно налази на једном месту (за разлику од сата који се практично налази тамо где и сам корисник који га носи) и ту пружа услуге за које је предвиђен. Такође, комуникациони канал који преноси податке између две стране стога из практичних разлога не може бити *Bluetooth*, већ интернет, тако да је комуникација могућа не само на кратким растојањима, већ и на веома удаљеним местима.

Практичан пример где корисник може употребити систем је следећи: уколико се уређај налази у кући, за кога су повезани температурни сензор, а такође и управљач за клима уређај (или грејач или други неки регулатор температуре), а корисник се налази на удаљеном месту и враћа се назад, могуће је да кроз апликацију на свом телефону приступи уређају и прегледа температуру. Уколико жели већу/мању температуру до тренутка када стигне, могуће је да уређај преко апликације подеси да, коришћењем поменутог хардвера, изрегулише температуру на жељену вредност. Поред овога, постоје и многи други примери који илуструју употребу уређаја.

Постоје многи други уређаји, идеје и пројекти који функционишу на сличан начин. Као што је већ поменуто, сам принцип било каквог ИоТ система функционише на управо тако. Корпорација *Apple Inc.* је развила идеју о паметној кући која функционише на слично, а сврха пројекта је била управо удаљено управљање паметних справа, уређаја и прибора који се налазе у кући помоћу удаљеног *Apple TV*, *iPhone*, или *iPad*. На слици 1 се могу видети разни аспекти које је корпорација *Apple Inc.* успела да покрије својим пројектом, а слична идеја је била основна мотивација за креирање овог система.



Слика 1 – аспекти покривени од *Apple*-овог система за паметну кућу

### 3. ЗАХТЕВИ ЗА РЕАЛИЗАЦИЈУ СИСТЕМА

У овом поглављу биће детаљније описани захтеви за реализацију читавог система. Најпре ће бити дефинисан начин раздвајања корисничке апликације од уређаја, где се суштина заснива на њиховој комуникацији, потом минимални функционални аспекти који представљају минималан сет радњи и функција неопходних за испуњавање сврхе и циља пројекта, а потом ће бити разматране технологије и листа коришћених библиотека приликом имплементације система.

#### 3.1. КОМУНИКАЦИЈА ИЗМЕЂУ АПЛИКАЦИЈЕ И УРЕЂАЈА

Као што је већ поменуто, систем као комуникациони медиј између корисничке апликације и уређаја користи интернет. Да би се постигао раније поменути ефекат, да је могуће развити било какву апликацију која може да се лако интегрише у читав систем (прецизније са уређајем), неопходно је дефинисати протокол по ком би клијентска апликација морала да комуницира са уређајем. То мора бити такав протокол да је независан од платформе, а такође мора поседовати неки метод или интерфејс који омогућава излазак на интернет. Веома је битно напоменути да се овде ради о стандардној клијент-сервер архитектури, која је заправо заснована на *http* протоколу.

##### 3.1.1. ПРОТОКОЛИ НА КОЈИМА СЕ ЗАСНИВА КОМУНИКАЦИЈА

С обзиром да се ради о комуникацији преко интернета, као и многе друге архитектуре које су на исти начин базиране, уређај представља сервер који пружа услуге клијентима, односно корисничким апликацијама. Како се кроз апликацију захтева одређена радња од уређаја (нпр. кликом на одређен тастер), ту информацију (где она представља сам захтев) је неопходно пренети до уређаја. У основи, користи се већ поменути *http* протокол.

Наиме, *http* протокол је протокол апликационог слоја, који дозвољава да две апликације комуницирају преко мреже. Међутим, његова спецификација ништа не говори како информације заиста теку кроз мрежу и достижу до сервера (уређаја) – то је посао слојева који се налазе испод њега. Ове информације (у наставку ће се користити термин поруке који је прикладнији у овом контексту) морају да прођу кроз низ слојева по слању а затим при пријему такође морају да прођу кроз исте слојеве, али у обрнутом редоследу.

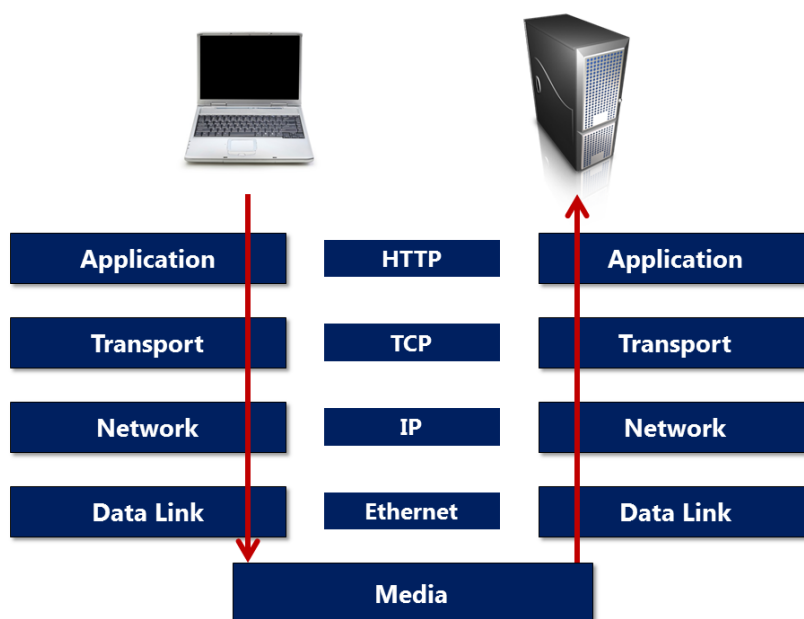
Слој који се налази испод апликационог је транспортни слој. У многим реализацијама на овом слоју налази се *TCP* протокол (енгл. *Transmission Control Protocol*), па тако и у реализацији овог пројекта (мада *http* не захтева директно њега). При слању било каквог захтева, клијентска апликација отвара *TCP socket* специфицирајући адресу уређаја и порт сервера, а онда шаље поруке (тј. податке) у сокет. *TCP* прихвата ове податке и обезбеђује да подаци стигну до сервера тако да се не изгубе ни дуплицирају. Овај протокол ће такође поново послати податке који су можда изгубљени у преносу. Стога, апликациони слој не мора да брине о губитку података, отуда је и *TCP* познат као поуздан протокол. Такође, овај протокол обезбеђује и детекцију грешака и *flow control* (осигурава да пошљалац не шаље пребрзо поруке за примаоца или за мрежу да процеси податке).

Укратко, *TCP* даје много значајних сервиса за успешну испоруку *http* порука. Као и многе друге апликације, ни клијентска апликација ни сам уређај не морају да воде много рачуна како о овом протоколу и слоју, тако и нижим. Како се говори о томе да клијентска апликација може бити израђена у било којој технологији, сама технологија диктира начин управљања и слања захтева уређају кроз *API* који пружа, док је на страни уређаја ситуација ипак другачија, о чему ће бити више речи касније.

Транспортни слој није једини слој испод апликационог. Наредни у низу је мрежни слој, где се налази интернет протокол (*IP*). Док је *TCP* одговоран за детекцију грешке, поуздан пренос, контролу тока (*flow control*), *IP* је одговоран да преузима делове порука и преноси их преко разних свичева, рутера, гејтвеја, рипитера и других уређаја који преносе поруке из једне мреже у другу кроз читав свет уколико је то потребно. Овај протокол не гарантује испоруку порука (као што је већ поменуто, то је задатак *TCP*-а). За пренос су неопходне *IP* адресе, и тако сам уређај има своју адресу (наравно, могуће га је спојити и са доменом, о чему ће бити више речи касније). Такође, протокол на овом слоју је такође одговоран за разлагање података у пакете (често по имену *datagram*), а понекад и за фрагментацију и реасемблирање тих пакета тако да су оптимизовани за одређени мрежни сегмент.

До сада, ови слојеви су слојеви који се налазе унутар самог једног уређаја на мрежи (било телефона, рачунара или било којег уређаја повезаног на мрежу, често именован као чвор), али у једном тренутку, *IP* пакети морају да буду пренети преко проводника, оптичког кабла, бежичне мреже или сателитске везе. За ово је одговоран слој везе (енгл. *data link layer*). Чест избор технологије у овом тренутку је *Ethernet*. *Ethernet* пакети постају оквири (енгл. *frames*), а протоколи на ниском нивоу попут њега су углавном фокусирани на 0, 1 и електричне сигнале.

На слици 2 може се видети приказ до сада поменутих слојева и принцип транспорта порука и комуникације између корисничке апликације и уређаја.



Слика 2 – слојеви и протоколи у комуникацији преко интернета



### 3.1.2. ПЛАТФОРМСКИ НЕЗАВИСАН ПРОТОКОЛ ВИШЕГ НИВОА

До сада је више пута поменуто да је неопходно постојање протокола између клијентске апликације и уређаја (који мора бити платформски независан) како би се омогућило да клијентска апликација буде потпуно независна од уређаја (у суштини и обрнуто), самим тим и добила могућност да се праве друге, нове апликације које се могу интегрисати у систем. У тачки 3.1.1. описани су протоколи и слојеви на којима се заснива комуникација, али то и даље није довољно да се оствари управо поменути захтев. Стандардни *http* обично се користи за веб претраживаче који свој садржај приказују у зависности од примљеног *html* садржаја који је пристигао кроз одговор сервера у протоколу. При томе, у овој реализацији, захтеви серверу немају много флексибилан сет порука које шаљу у телу захтева. Овакав начин комуникације би ограничавао многе аспекте у преносу порука, самим тим ограничавао и функционалности које се пружају крајњем кориснику (клијенту кроз апликацију), а притом такође и отежавао програмеру да се прилагоди самом протоколу при креирању нове апликације.

Ово су главни разлози због чега се инхерентно и долази до потребе за креирањем новог протокола који ће бити специјализован за овај систем, али такав да пружа велику флексибилност, како за текуће функционалности и захтеве који су присутни, тако и за будуће, нове захтеве, који би се накнадно додавали у систем као унапређења. Овакве карактеристике флексибилности и проширивости поседује *XML (eXtensible Markup Language)*. Овај *markup* језик је независан од платформе и апсолутно проширив, тако да се може правити било каква произвољна структура података која се може и накнадно лако мењати и надограђивати. Постоје већ слични протоколи који су засновани на оваквој архитектури, један од њих је и *SOAP* протокол.

Сваки захтев од стране клијентске апликације се може серијализовати у низ карактера који упућују сервер да клијент управо овакву операцију захтева, али није сваки избор било какве серијализације флексибилан, проширив, неутралан и независан. На сличан начин и функционише стандардни *http*, прецизније одговор сервера у *http*-у, где се кроз тело одговора шаље већ поменути *html*, који представља садржај, односно страницу коју веб претраживач треба да прикаже. Свакако, *html* је по природи исто *markup* језик као и *XML*, али за разлику од њега, има предефинисан сет структура порука које може да шаље, што смањује флексибилност, што је већ било поменуто у овој тачки. Међутим, начин слања страница (*html*-а) је од већег интереса. Уместо њега, на истом месту, сервер (уређај) клијенту (апликацији) може да шаље *XML*, што може да представља произвољан одговор са његове стране (конкретне поруке ће бити детаљно касније објашњене). Слично тако, као што је већ до сада поменуто, по стандардном *http*-у, тела захтева (који се шаљу од клијента ка серверу) често су или празна или садрже мање флексибилне информације, где се већина корисних информација од захтева заправо налазе у заглављу (енгл. *header*), па ни не постоји потреба за већим концептом у телу захтева. Међутим, само тело по *http*-у није предефинисано и може бити произвољног облика, јер у сваком случају, сервер на крају, по пријему захтева, добија и овај део поруке, а зарим га парсира и из њега извлачи корисне информације уколико је то и неопходно. Управо на овом месту се поново може употребити *XML*, у ком ће се налазити читав корисни садржај захтева, док би се у заглављу захтева ставили само они основни параметри и информације неопходне да се сам *http* испоштује.



Наравно, постоје и други протоколи који на сличан начин функционишу, где кроз тела прослеђују или *XML* (већ поменути *SOAP*), или други који на овом месту шаљу нпр. *json* формат података. Модерни *javascript* фрејмворци, као што су *Angular* и *React* са *node*-ом на серверу, па чак и технологије у *ASP.NET* или *Java* такође у својим неким деловима користе овакав принцип, али нису у потпуности засновани на овом концепту. Конкретне поруке у протоколу, принципи, зачења одређених делова и постојећи параметри ће касније бити детаљно размотрени.

### 3.2. ФУНКЦИОНАЛНИ АСПЕКТИ СИСТЕМА

Претходно је обрађен начин комуникације између корисничке апликације и уређаја, и по захтевима отуда дефинисана могућа (и искоришћена) реализација потпуног раздвајања корисничке апликације од уређаја како би она била независна. Тиме, корисничка апликација може не само пружати опције и услуге које пружа сам уређај, већ и на свој начин (по жељи) може да их комбинује и тако кориснику пружи много већи сет функционалности и кориснику прилагоди рад његовим потребама. У овом документу биће само обрађене основне функционалности, које су реализоване преко већ поменуте *cross-platform* апликације, која служи искључиво као пример да илуструје начин рада система. У наставку су дате основне опције и функције које се могу користити кроз систем:

- Успостављање конекције са уређајем – клијентска апликација мора да има могућност да унесе адресу уређаја на који жели да се конектује; у овом тренутку треба напоменути да је неопходно да се апликација и уређај на интернету „виде“, односно, неопходно је да су међусобно достижни (енгл. *reachable*);
- Приказивање листе компонената и њихових детаља – кориснику је неопходно приказати све компоненте са којима уређај располаже и са којима је могуће обављати жељене радње;
- Аквизиција података у реалном времену – приказивање стања свих компонената које су директно или индиректно повезане са уређајем (из листинга), а које спадају у његов периферни хардвер; ово могу бити стања сензора, стања дигиталних елемената, тренутно време које мери уређај или било који други параметар који је од интереса за корисника;
- Ручна контрола система – кориснику се мора обезбедити начин да целокупан периферни хардвер, па чак и сам уређај уколико поседује подешљиве или контролабилне (софтверске или хардверске) компоненте, може да управља на жењени налин, као и да те промене види у реалном времену кроз апликацију;
- Рад са аутоматским система – до сада је поменуто да сам уређај има и могућност извршавања аутоматске логике; у овом контексту, користиће се термин „догађај“, који ће касније бити детаљније објашњен, а сам концепт који систем (тачније уређај) поседује је флексибилан начин за креирање, мењање, чување и брисање произвољних, нових или већ постојећих аутоматски система који у одсуству човека могу да ураде жељени посао под одређеним условима.

### 3.3. КОРИШЋЕНЕ ТЕХНОЛОГИЈЕ И БИБЛИОТЕКЕ

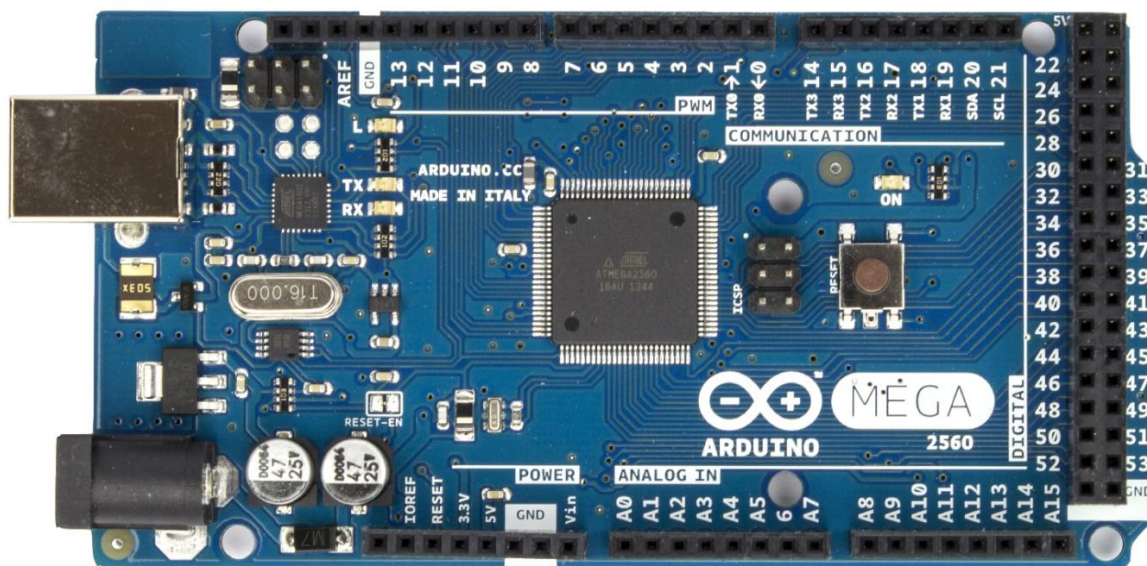
Већ је напоменуто да је уређај заснован на *Arduino* платформи. У овом контексту, то је и технологија која је коришћена за израду овог дела пројекта. *Arduino* постоји већ готово 15 година, и веома је популаран због своје једноставности. Не само да га могу користити почетници у свету микроелектронике, већ и искусни корисници за озбиљније пројекте. Користе се *Atmel*-ови микроконтролери који су веома јефтини. *Arduino* програмски језик је језик веома сличан познатом и веома коришћеном језику *C*, тачније *C++*, што утиче на његову брзу експанзију. Такође, ради се о једној *open-source* платформи, што значи да је целокупна софтверска подршка бесплатна и доступна свима, и може се преузети са сајта <https://www.arduino.cc>. Постоје бројни форуми где корисници размењују своја искуства, а сваки програмер може дати свој допринос развоју *Arduino* апликација и библиотеке.

За развој клијентске апликације коришћена је технологија *Xamarin.Forms*, која излаже комплетну гарнитуру *cross-platform UI* алата за *.NET* програмере. *Xamarin.Forms* је само део *Xamarin* технологије, популарне за развој мобилних апликација, *framework* који проширује *.NET* развојну платформу са алатима за развијање *native* мобилних апликација. *Xamarin.Forms* није само *cross-platform UI* библиотека, већ и потпуни апликациони *framework* који укључује све што је неопходно за развој мобилних апликација. Такође се ради о једном *open-source framework*-у, а платформе које укључује јесу већ поменуте *UWP*, *Android* и *iOS*.

У наставку ће детаљније бити обрађене коришћене технологије и библиотеке за уређај, а потом и за развој клијентске апликације.

### 3.3.1. РАЗВОЈНА КАРТИЦА *ARDUINO MEGA 2560*

Конкретна развојна картица која је коришћена у пројекту за реализацију уређаја се назива *Arduino MEGA 2560*. Развојна картица *Arduino MEGA 2560* је микроконтролер са *Atmel*-овим процесором *ATmega2560*. Састоји се из 54 улазно/излазних пинова (од којих се 15 могу користити као *PWM* излази), 16 аналогних улаза, 4 хардверска серијска порта, 16 MHz осцилатора, *USB* конекције, напонског прикључка, *ICPS header*-а и *reset* тастера. Изглед овог микроконтролера се може видети на слици 3.



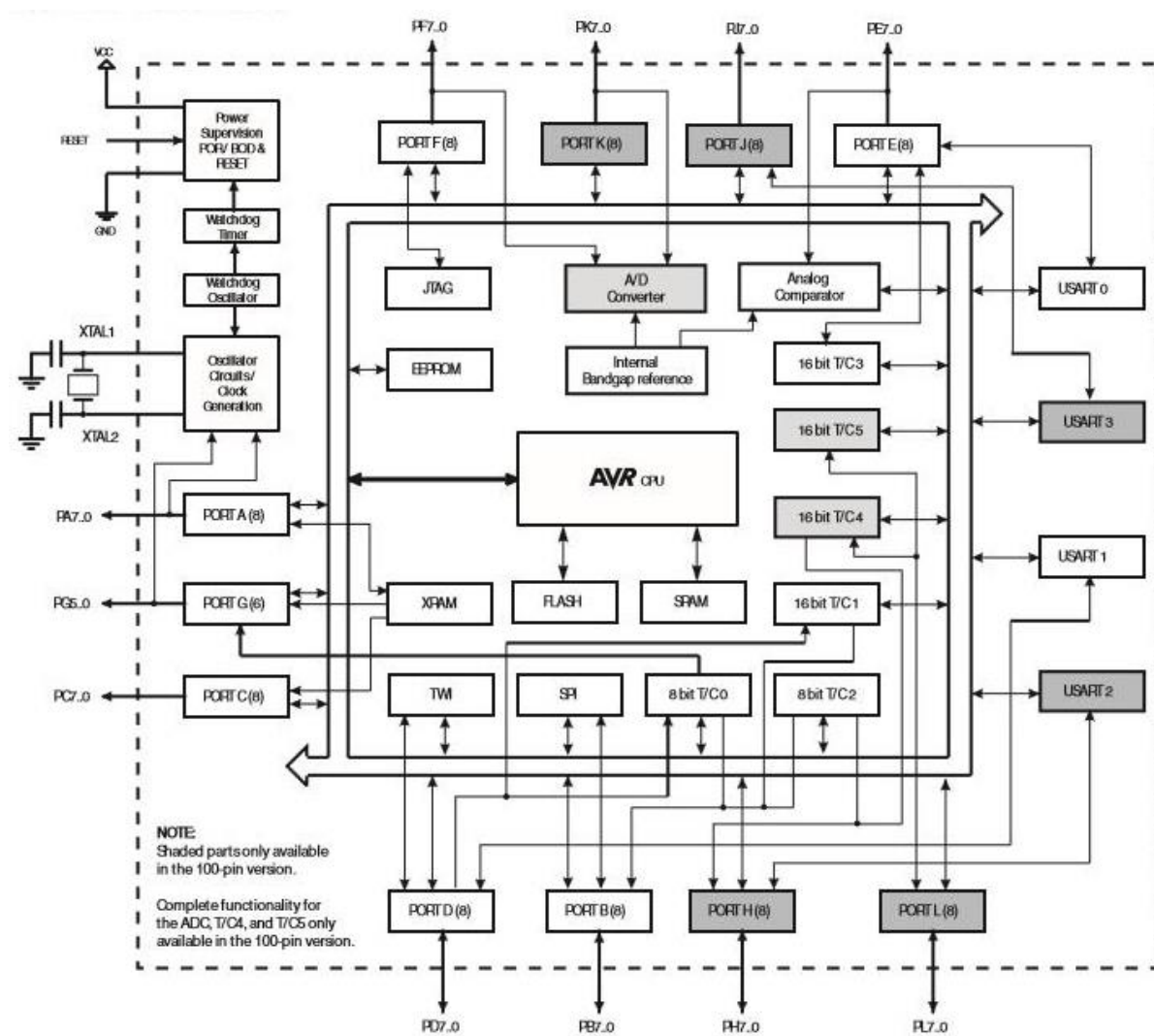
Слика 3 – изглед *Arduino MEGA 2560* картице

Основне карактеристике развојне картице:

Микроконтролер	ATmega2560
Радни напон	5 V
Улазни напон ( <i>препоручени</i> )	7-12 V
Улазни напон ( <i>гранични</i> )	6-20 V
Дигитални I/O пинови	54
Аналогни улазни пинови	16
Макс. струја I/O пина	40 mA
Макс. струја на 3.3 V	50 mA
Flash меморија	256 KB од којих се 8KB користи за <i>bootloader</i>
SRAM	8KB
EEPROM	4KB
Такт процесора	16 MHz

### 3.3.1.1. АРХИТЕКТУРА ATMEGA2560

Користи се микроконтролер високих перформанси, мале потрошње, базиран на CMOS осмобитном AVR микроконтролеру и са RISC архитектуром. Састоји се из: ISP флеш меморије капацитета 256 KB, 8 KB SRAM-а, 4 KB EEPROM-а, 86 I/O линија опште намене, 32 радна регистра опште намене, бројача у реалном времену, 6 тајмера/бројача са модом поређења, PWM излаза, двожичног серијског интерфејса, 16 10-битних A/D конвертора и JTAG интерфејса за *on-chip debugging*. Уређај постиже 16 MIPS-а на 16 MHz и ради са напоном од 4.5 V до 5.5 V. На слици 4 је приказана блок шема ATmega2560.



Слика 4 – архитектура ATmega2560

### 3.3.1.2. НАПАЈАЊЕ

*Arduino MEGA 2560* се може напајати путем *USB* конекције или преко екстерног напајања (извор напајања се аутоматски бира).

Екстерни извор напајања може бити *AC/DC* адаптер или батерија. Адаптер се повезује преко конектора од 2.1 mm, док се проводници са батерије воде на пинове *Gnd* и *Vin*. Картица функционише са напонима од 6 V до 20 V. Ако се напаја са напоном мањим од 7 V могуће је да ће напони на излазним пиновима картице бити мањи од 5 V. У случају напона већег од 12 V регулатор напона ће се појачано грејати. Самим тим, постоји могућност оштећења картице. Препоручени опсег је од 7 V до 12 V.

Пинови за напајање су следећи:

- **VIN** – улазни напон са екстерног напајања или *USB*-а,
- **5V** – излазни пин са регулисаним напоном од 5 V,
- **3.3V** – излазни пин са регулисаним напоном од 3.3 V,
- **GND** - маса
- **IOREF** – напонска референца за *I/O* пинове.

### 3.3.1.3. УЛАЗИ И ИЗЛАЗИ

Сви дигитални пинови (54) на *MEGA* картици се могу конфигурисати као улазни или излазни пинови коришћењем функција: *pinMode()*, *digitalWrite()*, и *digitalRead()*. Радни напон им је 5 V. Такође, сваки пин може да произведе, а такође и да прими максималну струју од 40 mA и садржи интерни *pull-up* отпорник (који по почетној конфигурацији није прикључен) од 20 k $\Omega$  до 50 k $\Omega$ . Неки пинови имају вишеструку функцију:

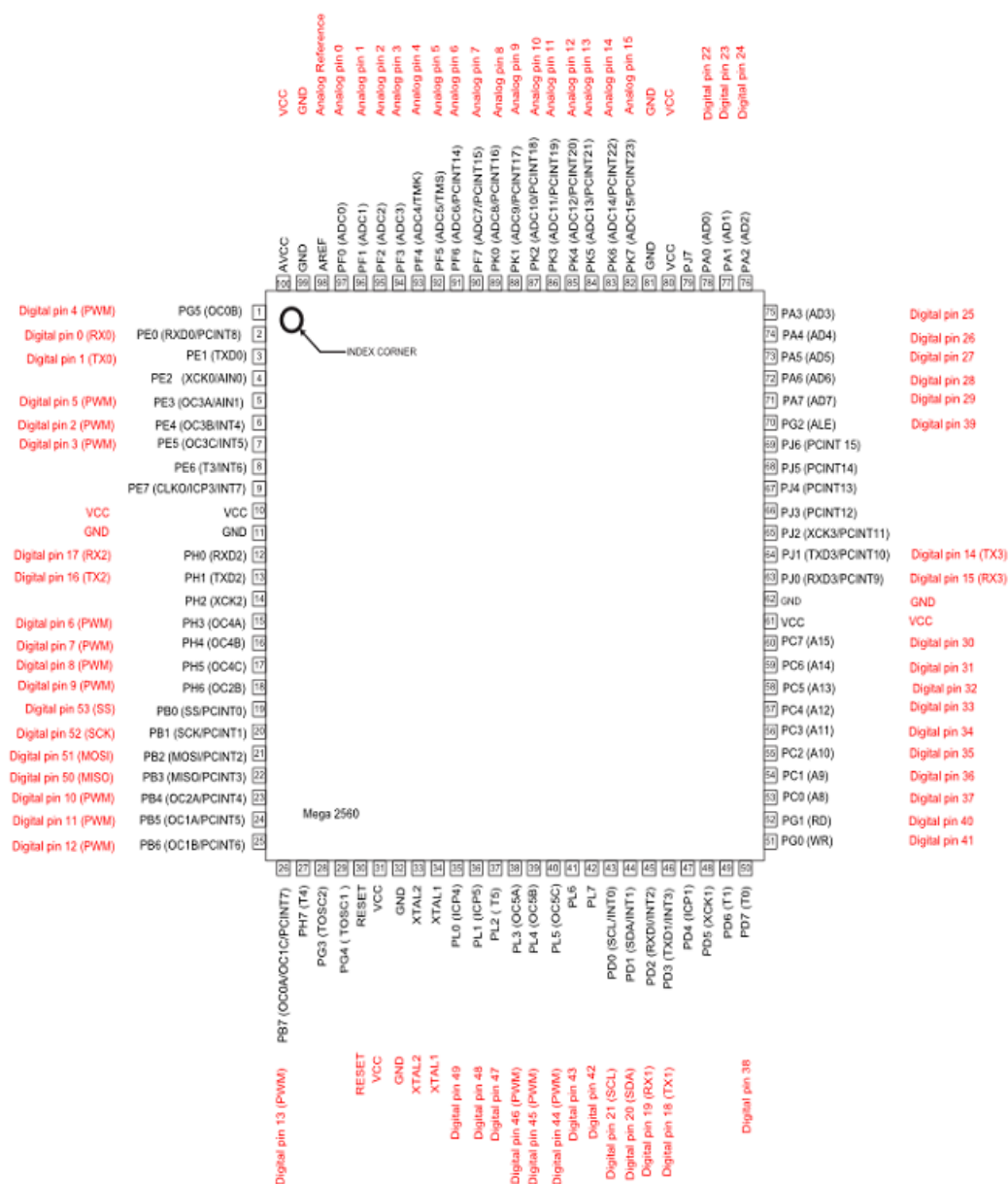
- **Serial 0: 0 (RX) i 1 (TX); Serial 1: 19 (RX) i 18 (TX); Serial 3: 17 (RX) i 16 (TX); Serial 3: 15 (RX) i 14 (TX)** – користе за примање (RX) и слање (TX) података; **TTL serial** – Vcc је логичка „1“, док је Gnd логичка „0“; **USB комуникација** – пинови 0 и 1.
- **Екстерни интерапти: пин 2 (интерапт 0), пин 3 (интерапт 1), пин 18 (интерапт 5), пин 19 (интерапт 4), пин 20 (интерапт 3) и пин 21 (интерапт 2)** – ови пинови могу бити конфигурисани да буду активни на низак ниво, растућу или опадајућу ивицу, или на промену одређене вредности.
- **PWM: пинови од 2 да 13 и од 44 до 46** – обезбеђују 8-битни *PWM* излаз преко функције *analogWrite()*.
- **SPI: пин 50 (MISO), пин 51 (MOSI), пин 52 (SCK), пин 53 (SS)** – ови пинови подржавају *SPI* комуникацију користећи *SPI* библиотеку.
- **LED: пин 13 – LED** је конектован на пин 13.
- **TWI: пин 20 (SDA) и 21 (SCL)** – подржавају *TWI* комуникацију користећи библиотеку *Wire*



*Mega 2560* такође поседује 16 аналогних пинова 10-битне резолуције, што значи да може да региструје 1024 различитих вредности при  $A/D$  конверзији. Ове вредности, у основном облику, крећу се од 0 V до 5 V, али се горња граница може мењати користећи *AREF* пин и *analogReference()* функцију. Поред досад наведених пинова, на картици постоје и други:

- ***AREF*** – даје могућност промене горњег граничног напона при  $A/D$  конверзији доводећи дати референтни напон на овај пин. Користи се са функцијом *analogReference()*
- ***Reset*** – хардверски ресет картице доводећи логичку „0“ на овај пин.

На слици 5 представљен је распоред свих пинова овог контролера.



Слика 5 – распоред пинова *Arduino Mega 2560*

#### 3.3.1.4. КОМУНИКАЦИЈА

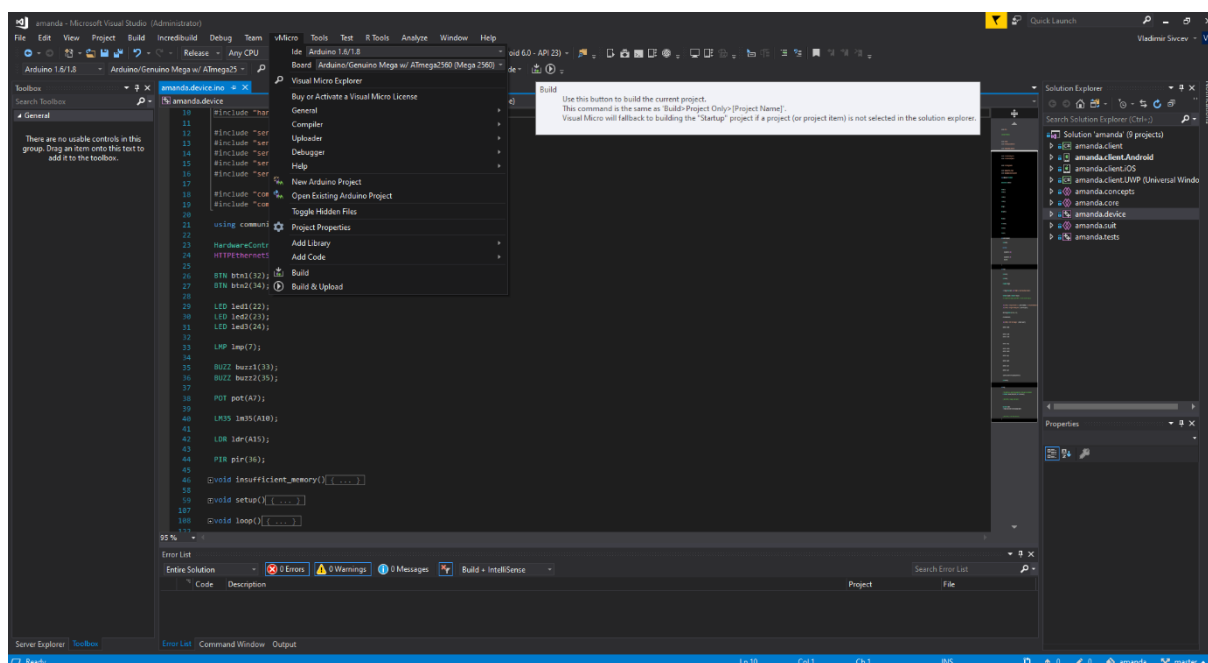
*Arduino MEGA 2560* садржи бројне могућности за комуникацију са рачунаром, *Arduino* или другим микроконтролерима. *ATmega2560* обезбеђује 4 *UART* хардверска порта за *TTL* серијску комуникацију. *Arduino* софтвер обезбеђује *Serial monitor* који омогућује једноставну комуникацију између рачунара и *Arduino* картице у оба смера размењујући податке у текстуалном облику. Постоје *RX* и *TX LED*-овке које трепере приликом преноса података између картице и рачунара преко *USB* конекције.

*Atmega2560* такође обезbeđuje *TWI* и *SPI* комуникацију. *Arduino* софтвер садржи библиотеке које поједностављују ове две врсте комуникације, а то су библиотеке *Wire* (за *TWI* комуникацију) и *SPI* (за *SPI* комуникацију).

### 3.3.1.5. ПРОГРАМИРАЊЕ

*ATmega2560* на *Arduino MEGA* картици долази са унапред уграђеним *bootloader*-ом који омогућава програмирање без коришћења екстерног хардверског програматора. Комуницира преко *STK500* протокола. Такође, након програмирања, *ATmega2560* има могућност софтверског *reset*-а, не обавезујући корисника да ручно (хардверски) ресетује контролер преко *Reset* тастера.

*Arduino* поред свог основног софтверског окружења, *Arduino IDE*, може да се програмира и помоћу познатог *Microsoft Visual Studio 2017* окружења, али за то је неопходно поседovati екстензију *Visual Micro*. На слици 6 се може видети изглед софтверског окружења са *Arduino* опцијама.

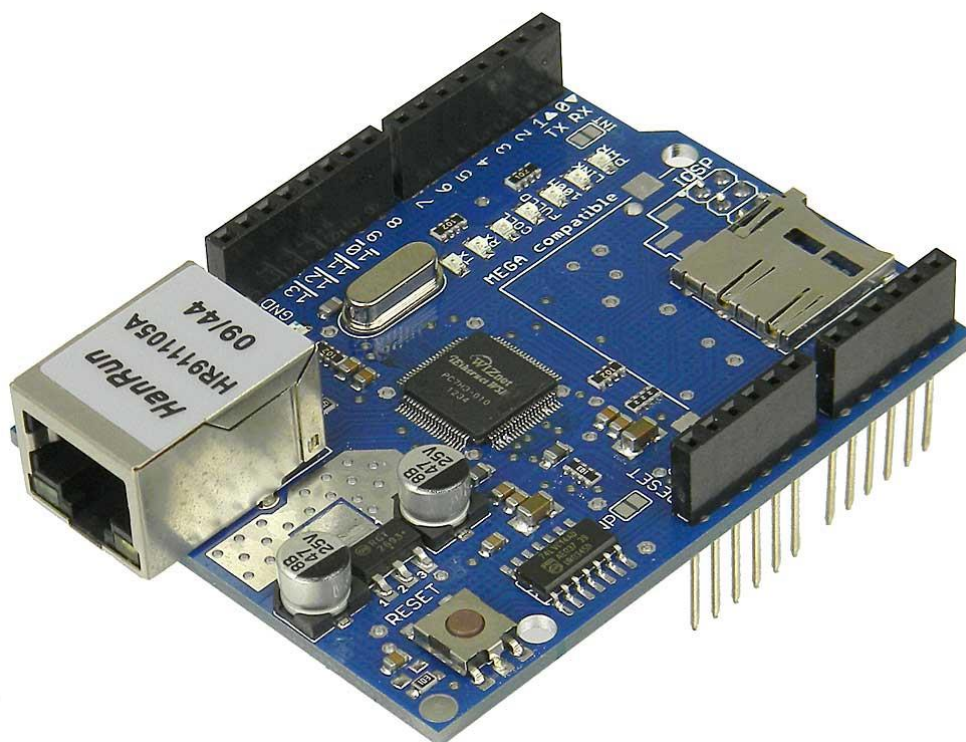


**Слика 6** – изглед окружења *Microsoft Visual Studio* са *Visual Micro* екстензијом



### 3.3.2. ARDUINO ETHERNET SHIELD

*Arduino Ethernet Shield* је додатна картица која омогућава да се *Arduino* микроконтролер конектује на интернет. Базиран је на *Wiznet W5100 ethernet* чипу. *Wiznet W5100* обезбеђује оба протокола преноса података, *TCP* и *UDP*. Такође, подржава и до 4 истовремене *socket* конекције. Користи се са *Ethernet* библиотеком приликом писања програма који захтевају интернет конекцију. Као и друге додатне картице (*shield*-ови), и *Ethernet Shield* се поставља на врх другог *Arduino* микроконтролера за остваривање своје функције. Садржи стандардну *RJ-45* конекцију. Слика 7 приказује изглед *Arduino Ethernet Shield*-а.

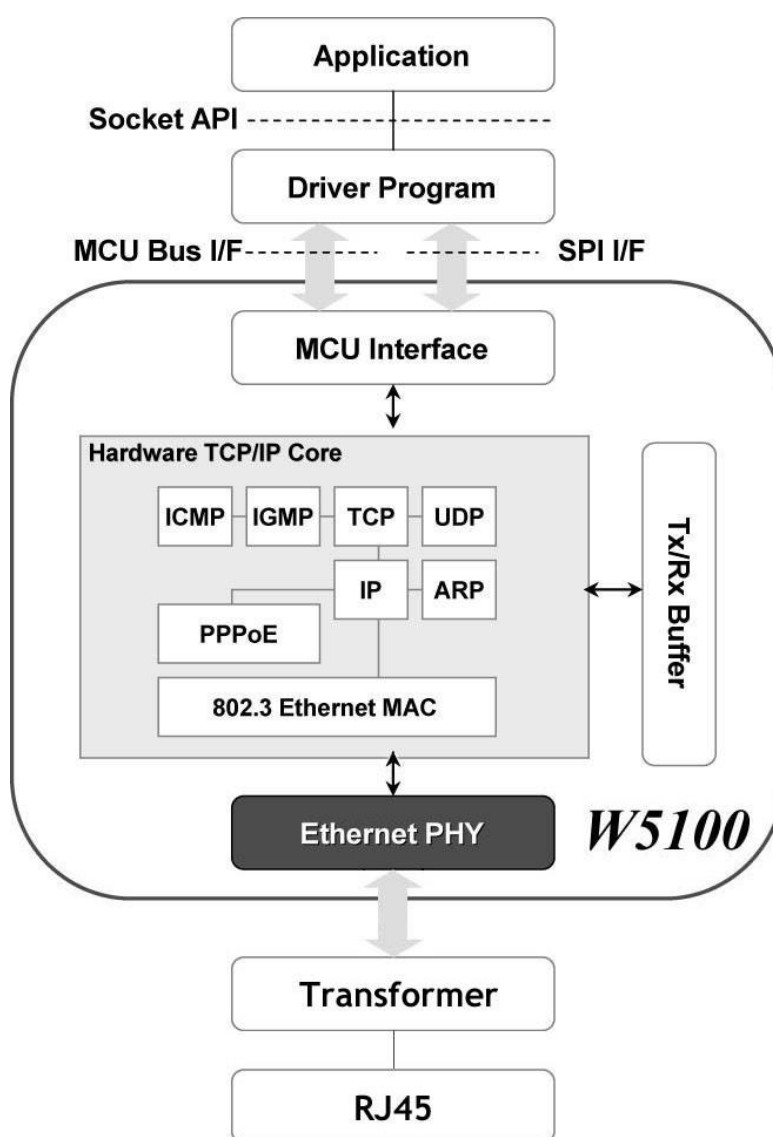


Слика 7 – изглед *Arduino Ethernet Shield*-а

### 3.3.2.1. WIZNET W5100

*Wiznet W5100* је потпуно опремљен, једно-чипни 10/100 *Ethernet* контролер направљен за уграђене системе где је неопходна једноставна имплементација, добра стабилност и добре перформансе. Највише се користи када је неопходна уградња интернет конекције без оперативног система. Може се користити са стандардним *IEEE 802.3 10BASE-T* и *802.3u 100BASE-TX*.

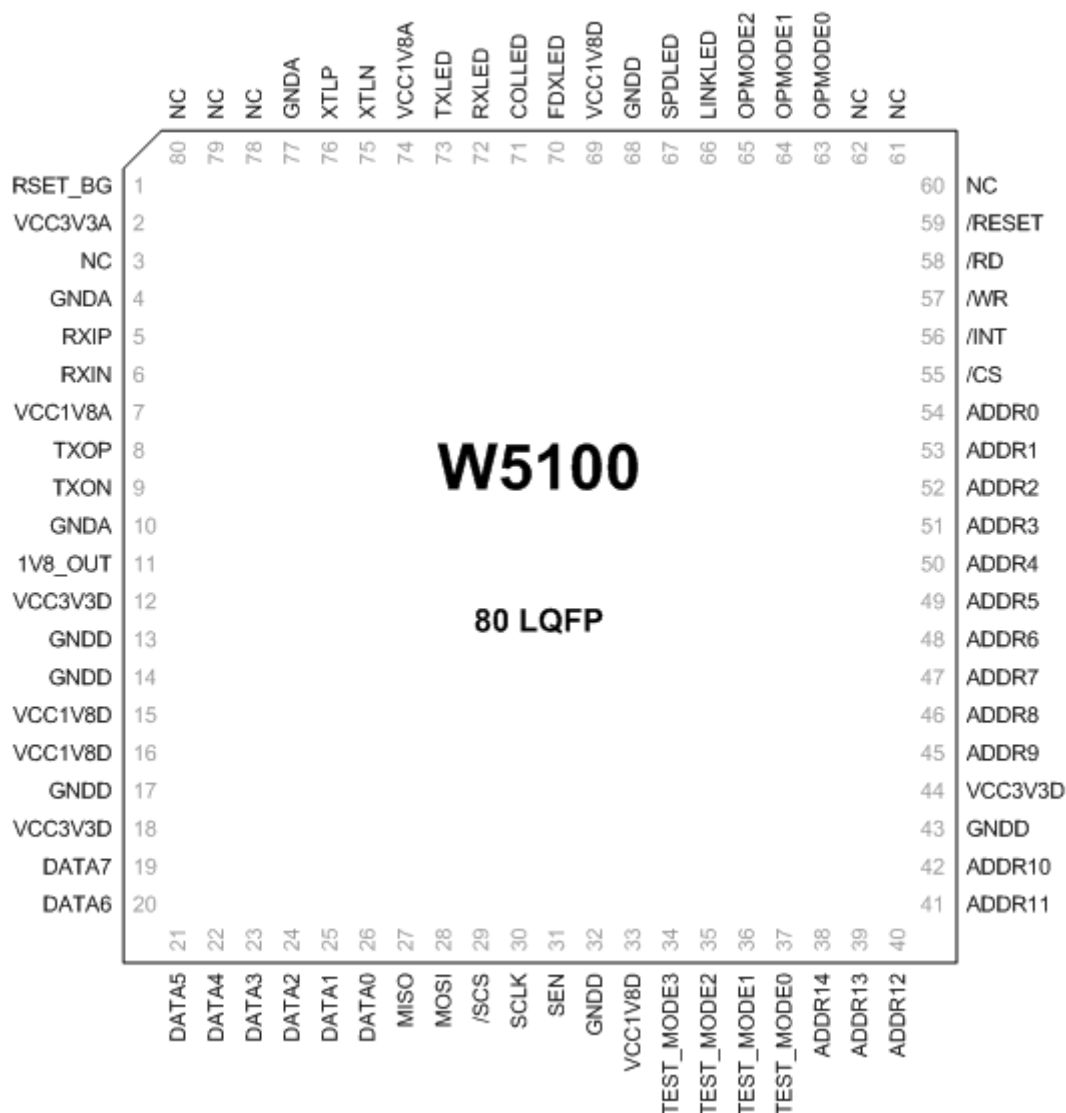
*W5100* садржи потпуно непроменљиво *TCP/IP* складиште и интегрисани *Ethernet MAC & PHY*. Складиште подржава *TCP*, *UDP*, *IPv4*, *ICMP*, *ARP*, *IGMP*, и *PPPoE* протоколе који се користе у разним апликацијама већ много година. Такође, уграђен је и интерни бафер од 16 KB за трансмисију података. Блок шема *Wiznet W5100* чипа се може видети на слици 8.



Слика 8 – блок шема *Wiznet W5100*

### 3.3.2.2. ПИНОВИ И КОНЕКТОРИ

*Arduino Ethernet Shield* садржи *RJ-45* конектор који служи за повезивање са локалном мрежом (са локалним рутером, свичем или хабом) помоћу стандардног *ethernet* кабла. Користи дугачке ножице за конектовање са микроконтролером, што не мења првобитни распоред пинова самог микроконтролера, па омогућава надовезивање других додатака (*shield*-ова). Распоред пинова на *W5100* чипу представљен је на слици 9.



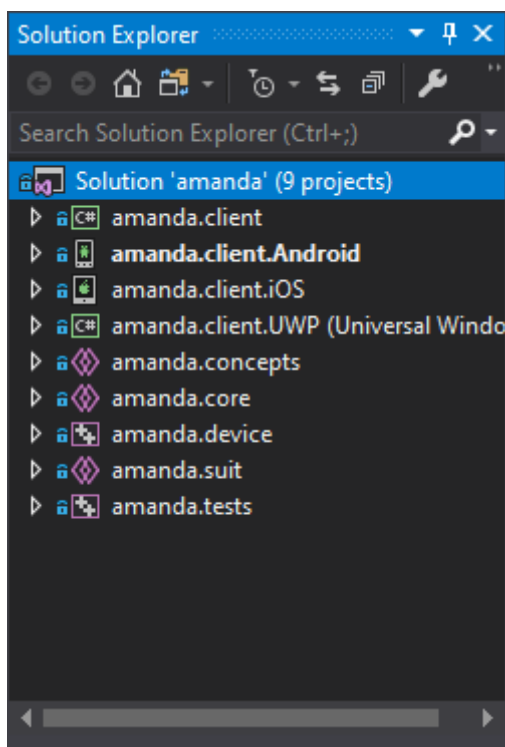
Слика 9 – распоред пинова на *Wiznet W5100*

*Ethernet Shield* садржи слот за *micro-SD* меморијску картицу, која се може користити за чување датотека за опслуживање на мрежи. Компатибилан је са *Arduino UNO* и *Arduino Mega 2560* микроконтролерима (користећи *Ethernet* библиотеку). Читач *micro-SD* картице доступан је преко *SD* библиотеке. Када се ради са овом библиотеком, *SS* (*Slave Select*) је на пину 4. Такође, *shield* садржи и *reset* контролер, који обезбеђује да се *W5100 Ethernet* модул коректно ресетује по укључивању.

*Arduino* комуницира и са *W5100* и са *SD* картицом преко *SPI* магистрале (односно преко *ICSP header*-а). Ово се налази на дигиталним пиновима *10*, *11*, *12* и *13* на *Uno* картици, док су на *Mega* картици то и пинови *50*, *51* и *52*. На обе картице пин *10* се користи за селектовање *W5100*, а пин *4* са *SD* картицу. Ови пинови се не могу користити за општу *I/O* примену. На *Mega* картици, хардверски *SS* пин, *53*, се не користи ни за селектовање *W5100*, ни за *SD* картицу, али се мора држати као излазни пин, иначе *SPI* интерфејс неће радити.

### 3.3.3. КОРИСНИЧКА АПЛИКАЦИЈА

За израду корисничке апликације коришћен је *Xamarin.Forms framework*. Као што је већ речено, апликација урађена у овом *framework*-у може да се покреће на више платформи. Отуда, сам концепт решења састоји се из 4 под-пројекта. Главни под-пројекат јесте дељени пројекат између 3 платформе (*UWP*, *Android* и *iOS*) где се већина, у овом случају цела апликација налази. Остали под-пројекти су везани конкретно за сваку платформу и могу да садрже специфичан код и извршне јединице које су везане искључиво за ту платформу. На слици 10 може се видети *Solution Explorer* где се налазе сви под-пројекти самог решења (прва 4 су везана за клијентску апликацију). Први под-пројекат представља дељни, док су остала 3 платформски специфични.



Слика 10 – под-пројекти читавог система (прва 4 представљају клијентску апликацију)

Принцип дељења пројекта заснива се на *Microsoft .NET Standard*-у који је базиран на узорку *Dependency Injection*. Ово подразумева да је неопходно да свака платформа имплементира прописани стандард како би сама апликација могла да буде писана у једном језику (*C#*), и то само једном, за све платформе. По покретању, ствара се почетна зависност везана за дату платформу, која се преко *Dependency Injection* узорка прослеђује

кроз читав дељени под-пројекат (који се иначе преводи као динамички извршна библиотека - *.dll*), и на тај начин постиже јединственост апликације.

Из ових разлога, кориснички интерфејс апликације је конзистентан кроз све платформе. Поред тога, све *UI* компоненте су потпуно *native*. Из овога се добија апсолутно исти изглед и осећај (енгл. *look and feel*) апликације кроз све платформе, као и да корисницима даје могућност да користе одлике и својства компонената карактеристичним за платформу коју користе, на какво коришћење су иначе навикли, па не морају да се прилагођавају новим.

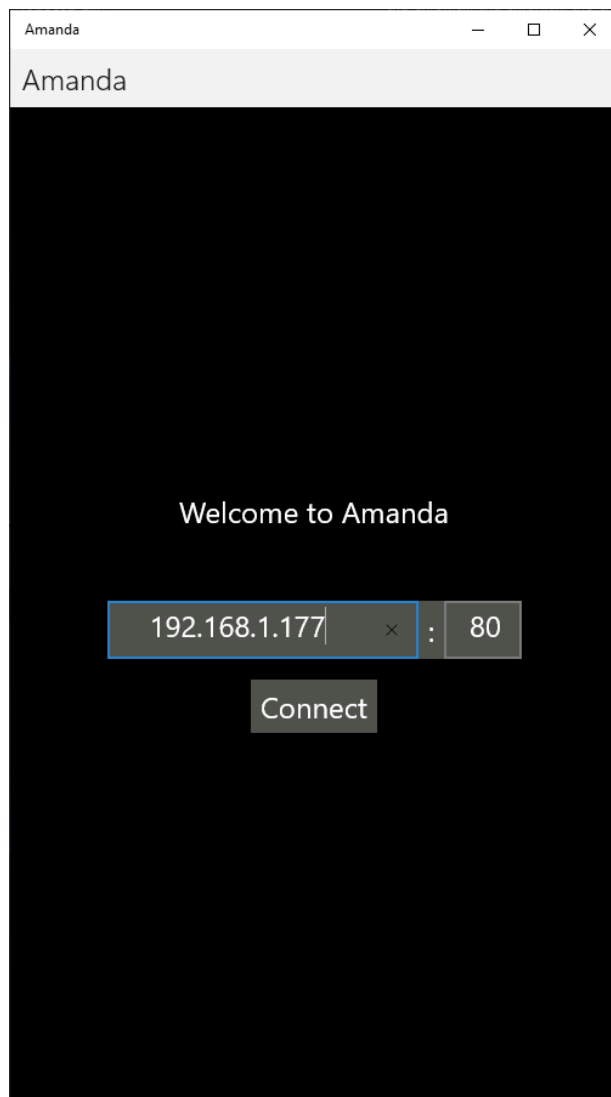
Сам *UI* је такође раздвојен из програмерске перспективе, а пише се кроз *XAML*. То је декларативни начин креирања интерфејса, који је заснован на *XML*-у. Ово није једини начин креирања интерфејса, такође је могуће писати и *C#* код, који даје једнаке одлике, а такође и перформансе самог *UI*. Међутим, принцип *XAML*-а је да се на овај начин прави *View* део апликације који раздваја целине апликације на начин сличан другима који су често коришћени кроз многе друге апликације и технологије а користе концепт *Model-View-Other*. Један од примера је много познати узорак *Model-View-Controller*, односно скраћено *MVC*, међутим, *Xamarin.Forms* користи концепт *MVVM* (*Model-View-ViewModel*) који показује још веће смањење кода и раздвојеност.

Коришћене библиотеке за израду апликације су следеће:

- *.NET System* – системска библиотека од *.NET Framework*-а која се користи за било какав *.NET* пројекат
- *Xamarin.Forms* – библиотека за креирање *cross-platform* апликација
- *.NET Standard Library* – библиотека за стандард при дељењу кода међу пројектима
- *.NET Core+Universal Windows* – библиотеке које се користе за израду *UWP* апликација
- *Mono.Android+Xamarin.Android.Support* – библиотеке које се користе за израду *Android* апликација у *.NET*
- *Xamarin.iOS* – библиотека која се користи за израду *iOS* апликација у *.NET*

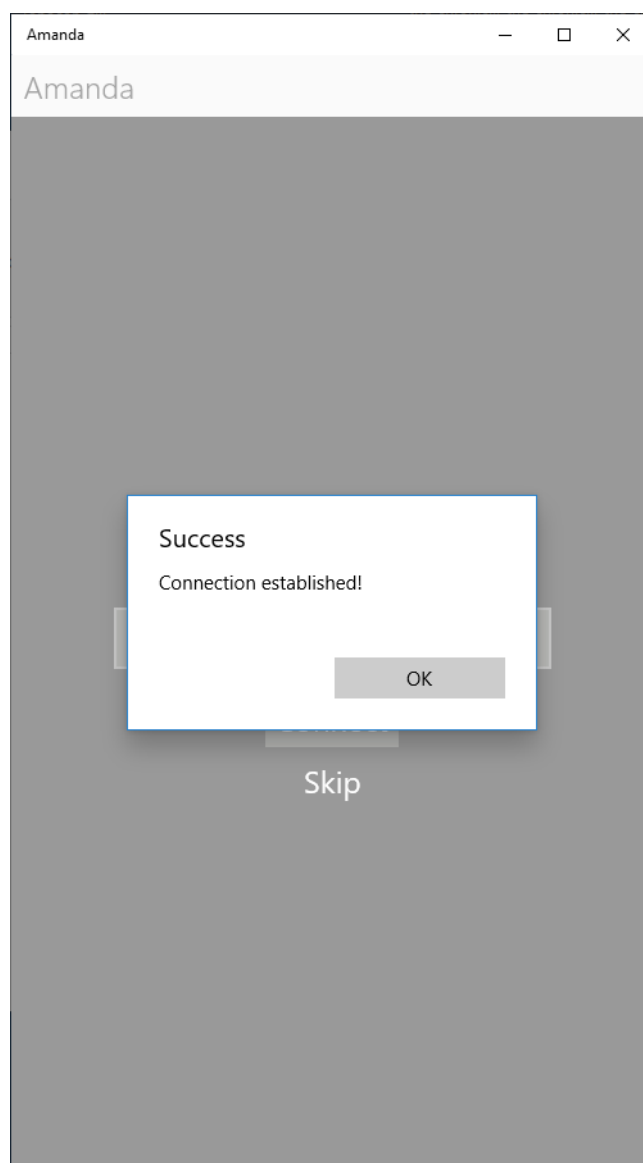
## 4. ОПИС РАДА СИСТЕМА

По отварању апликације, на почетном екрану налазе се опције за успостављање везе саме апликације са уређајем (сервером). Овде, неопходно је унети адресу уређаја и порт на ком се налази сервер (обично 80), а затим кликнути на тастер *Connect*. На слици 11 може се видети изглед почетног екрана.



Слика 11 – почетни екран корисничке апликације

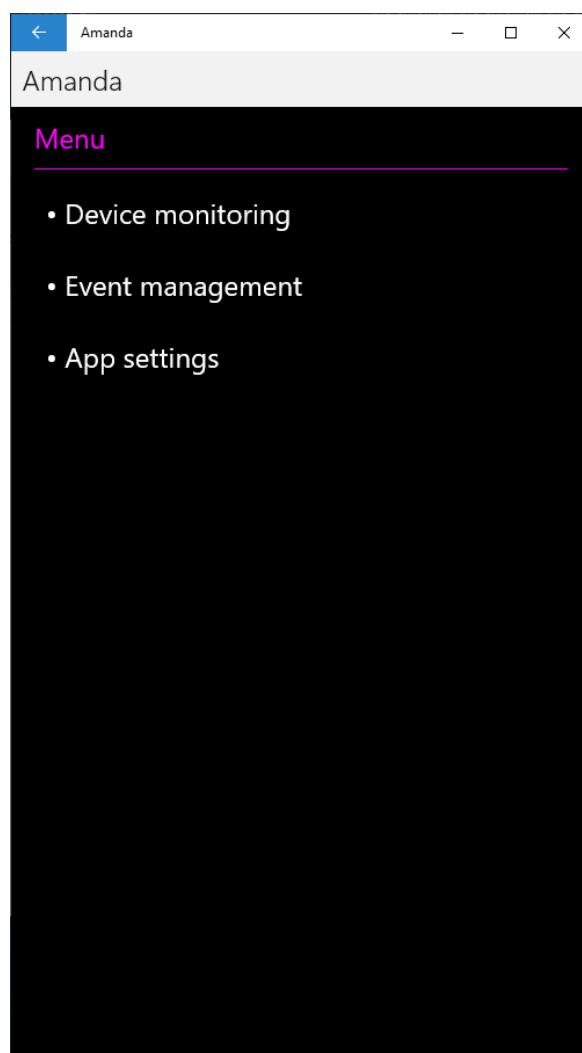
Као што се на слици може видети, адреса самог уређаја не мора бити доменска, већ може и директна (локална или глобална) *IP* адреса. Након што се притисне (додирне или кликне) тастер *Connect*, апликација ће започети конектовање са уређајем. Бројне поруке грешака могу искочити уколико конектовање није успешно. Уколико пак јесте, слика 12 показује поруку која ће у тој ситуацији искочити.



**Слика 12** – екран по успешности конекције са уређајем

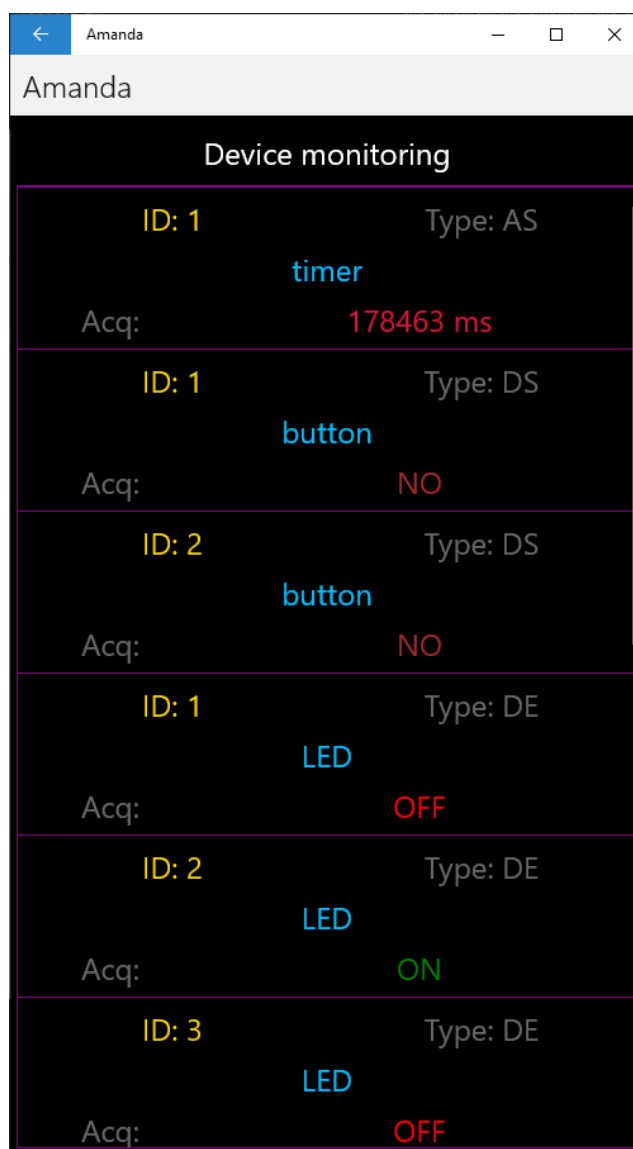
Кликом на *OK* тастер са слике 12, отвара се страница са главним менијем. На овој страници налазе се 3 основне опције. Прва опција води ка страници где се приказују подаци од аквизиције и такође обавља мануелна (ручна) контрола уређаја. Друга опција води до странице где је могуће подешавати аутоматику (односно догађаје) и комплетан рад са њима. Трећа опција је резервисана за подешавања која и даље нису имплементирана, па тиме опција не ради ништа. Ово се може видети на слици 13.





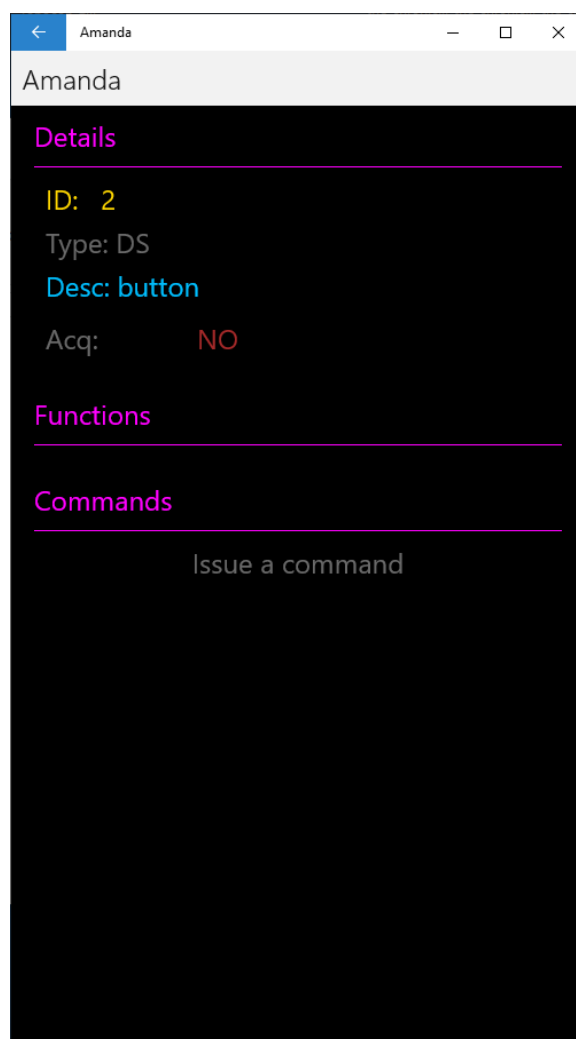
**Слика 13** – главни мени апликације

Кликом, односно додиром на одређену опцију, апликација прелази на странице које су описане изнад. На слици 14 представљен је екран по притиску на прву опцију у менију, „*Device monitoring*“.



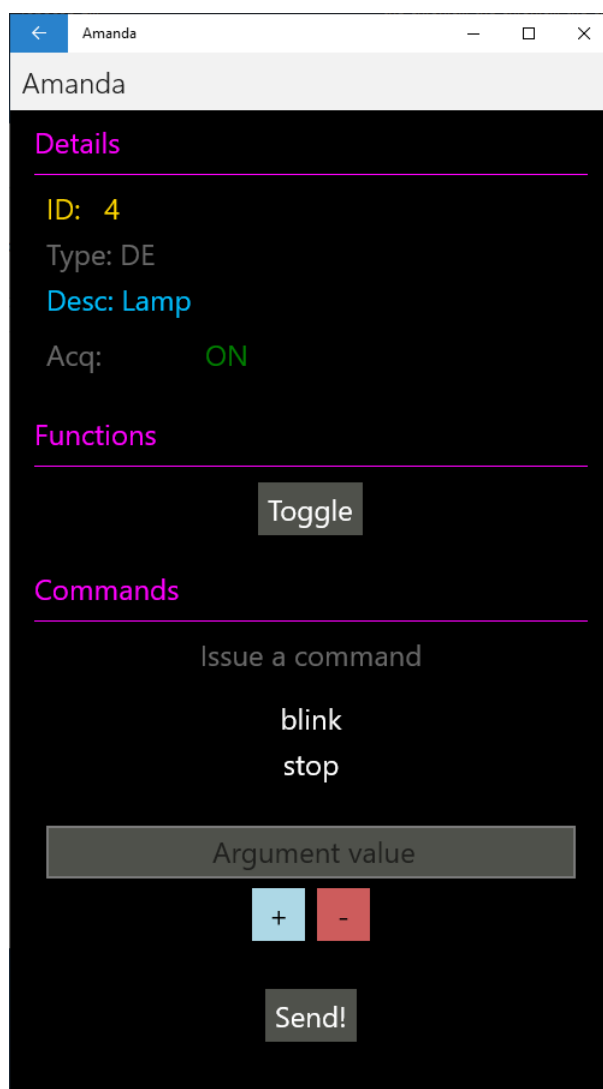
Слика 14 – приказ компонената заједно са аквизиционим подацима

На овом екрану се може видети листа компонената које су везане за уређај и спадају у његов периферни хардвер. На слици 14 се само виде одређене компоненте, јер је сама листа скролабилна. Поред њих, могу се такође видети и подаци који су прикупљени. На пример, може се видети да је *LED* са идентификатором 1 искључена, а *LED* са идентификатором 2 укључена. Други подаци ће детаљније бити објашњени у опису имплементације самог система. Такође, притиском на било коју компоненту, могуће је отворити страницу која приказује детаље те компоненте, а такође и приказује додатне опције које се тичу саме те компоненте. На слици 15 представљен је екран другог тастера који је као периферна компонента прикључен за уређај.



Слика 15 – детаљи тастер компоненте

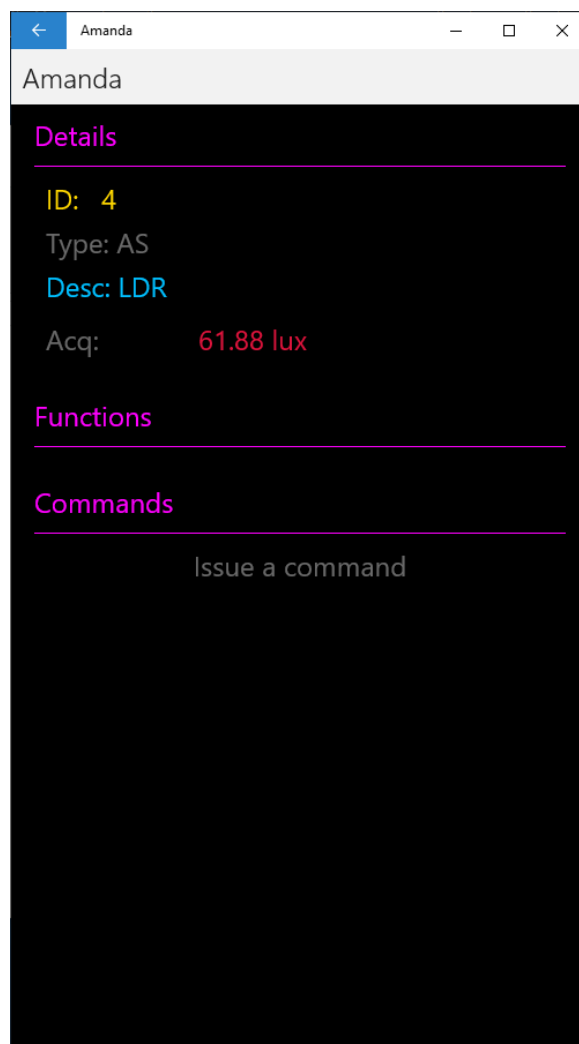
На слици 15 се могу видети неки од параметара самог тастера. Фокус треба да буде на податку “*Acq*”, који представља аквизитован (енгл. *Acquisition, Acquire*) податак. У овом случају, може се видети да сам тастер није притиснут. Поред тога, испод се налазе још два одељка, први који се тиче додатних функционалности саме компоненте, а такође и одељак где је могуће компоненти издавати команде. Дата компонента нема ни додатне функције, нити јој је могуће издавати било какву команду. На слици 16 приказана је друга компонента, лампа, која насупрот тастеру, садржи и додатне функције, а такође и команде.



Слика 16 – детаљи лампа компоненте

Као и код претходног примера, може се видети да је овде лампа, овог пута укључена. Од додатних функционалности, сама лампа садржи опцију (тастер) *Toggle*, чијим притиском се мења стање саме лампе (укључује се или искључује у зависности од тренутног стања). Ово је уједно и једина додатна функционалност за лампу. Функционалности генерално представљају додатне способности групе компонената. Конкретно, сваки дигитални елемент који је повезан са уређајем као његова периферна компонента, која наравно има 2 стања (отуда и дигитална), може да мења стање и самим тим је *toggle*-абилна, па ће имати ову функционалност (нпр. диоде, лампе тј. сијалице, вентилатори који се могу само укључивати/искључивати и било каква друга која је описана са 2 стања). Слично тако, важи и за друге типове компоненти које имају сличне особине. За разлику од њих, команде су специфичне искључиво за конкретну компоненту, дакле, искључиво зависе само од ње. У овом примеру, лампа има 2 команде, а то су *blink* (трептање) и *stop* (прекид трептања). Свака команда такође има и свој предефинисани и унапред познати сет аргумената. На пример, команда *blink* само има 1 аргумент, а то је дужина периоде трептања, и то у милисекундама. На тастере „+“ и „-“ може се додавати још уноса за нове аргументе, као и уклањати постојеће. По притиску на саму команду, притиском на тастер „Send!“ шаље се команда уређају и он је извршава.

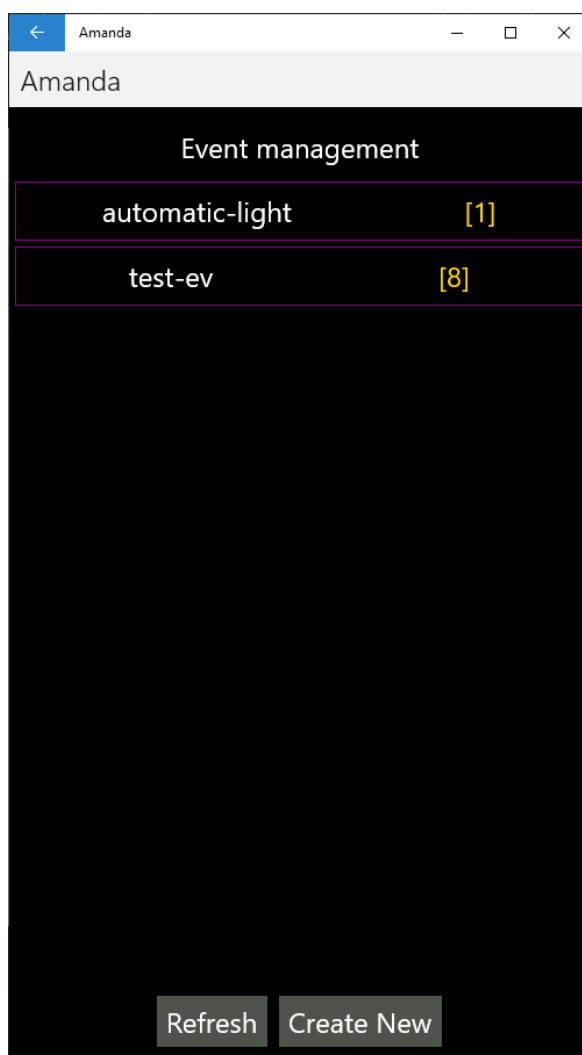
До сада, две представљене компоненте су биле дигиталне. Једна компонента је била сензор, а друга елемент (у овом систему, сензори су улазне компоненте и дају неку информацију, док су елементи излазне компоненте на које се могу слати информације). Примера ради, на слици 17 може се видети аналогна компонента *LDR*, која мери ниво осветљења у просторији.



Слика 17 – детаљи *LDR* компоненте

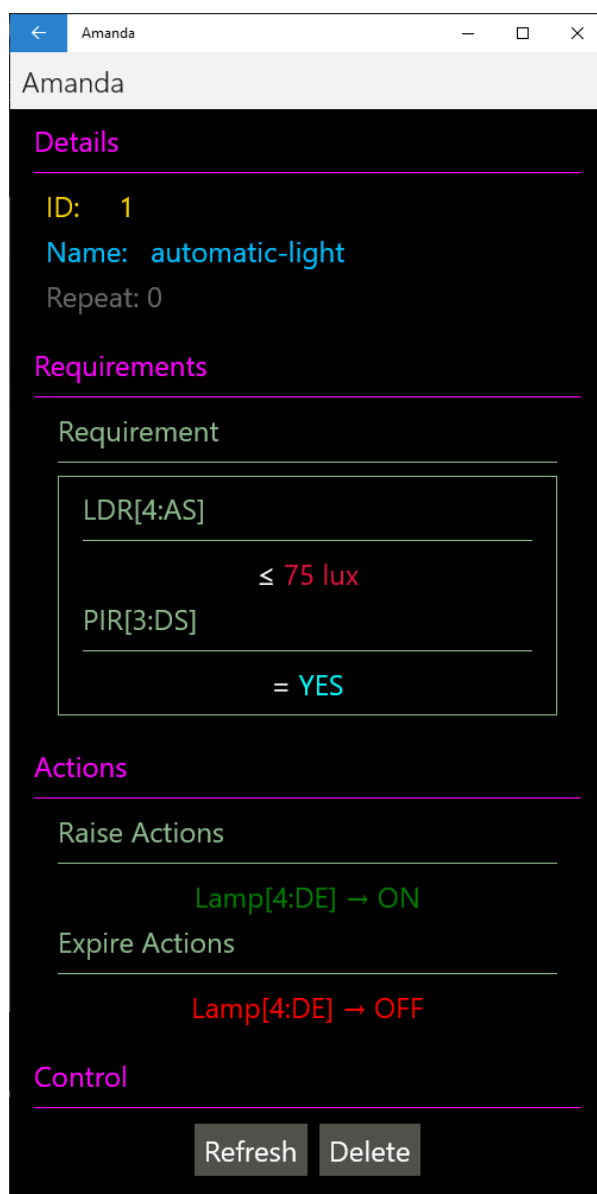
До сада је описан принцип аквизиције, како се могу пратити подаци који пристижу од уређаја, а такође и принцип како се компоненте могу ручно контролисати. Све ово спада у *Device monitoring*, прва опција главног менија апликације са слике 13. Наредна опција у менију је управљање догађајима.

Догађај је целина која представља један аутоматски под-систем у уређају. Помоћу овог концепта, могуће је правити произвољне догађаје, где корисник може да дефинише логику (односно аутоматику) коју жели да уређај ради (извршава) без његове директне или индиректне интервенције. На слици 18 може се видети екран који је добијен притиском на другу опцију у главном менију, *Event management*.



Слика 18 – екран за управљање догађаја

Главнина странице садржи листу свих постојећих догађаја који су дефинисани у систему. У овом случају, систем садржи свега два догађаја. То су *automatic-light*, а затим *test-ev*, са својим идентификаторима. Име сваког догађаја је произвољно и уноси се при креирању самог догађаја. Ово су само примери који илуструју начин рада догађаја. Конкретно, *automatic-light* је догађај који регулише осветљење просторије (или нпр. дворишно осветљење). Други, *test-ev*, је догађај који нема практичног значаја, али илуструје флексибилност и скалабилност овог концепта, а коришћен је за сврхе тестирања. При дну, постоји тастер *Refresh* чија сврха је да освежи листу догађаја уколико је на серверу (уређају) дошло до неке промене. Други тастер, *Create New*, одводи до странице за креирање новог догађаја. Такође, притиском на неки од догађаја из листе, отвара се страница која приказује детаље тог догађаја. На слици 19 могу се видети детаљи првог постојећег догађаја, *automatic-light*.

Слика 19 – детаљи догађаја *automatic-light*

На слици изнад може се видети додељени идентификатор уређаја, као и његово име. Наредни податак је податак *Repeat*. Овај податак представља број понављања самог догађаја, односно, колико максимално пута ће се овај догађај извршавати. Нула у овом податку значи неограничен број пута, док било који други позитиван број значи управо толико. За сваки догађај, битна су 2 концепта. То су услови (енгл. *Requirements*) и акције (енгл. *Actions*). Прецизније, концепт догађаја функционише тако да када његови услови постану тачни, у том тренутку извршиће се акције које су дефинисане под *Raise Actions* (практично акције које се извршавају када се догађај деси – *Raise*), а оног тренутка када услови престану да буду тачни, извршавају се акције под *Expire Actions* (дакле када догађај истекне – *Expire*). Све улазне компоненте, односно сензори, су компоненте које могу да играју улогу у условима, док излазне, односно елементи, могу да играју улогу у акцијама. У овом примеру, када сензор осветљења *LDR* читава ниво осветљења испод 75 lux и када сензор покрета *PIR* чита да се неко креће кроз нпр. двориште (што значи да је осветљење у дворишту слабо, а неко се креће кроз њега), акцију коју треба одрадити



јесте да се лампа (са идентификатором 4) укључи (ово нпр може бити лампа, односно сијалица у дворишту која га осветљава). У супротном, било да се више нико не креће кроз двориште, или уколико оветљење постане веће (дакле догађај се завршио – *Expire*), дешавају се акције под *Expire Actions*, где је дефинисано да се иста лампа, односно сијалица искључује. У доњем делу се могу видети два тастера, а то су тастер *Refresh*, који служи да освежи догађај уколико се нека промена десила на серверу, а затим тастер *Delete*, који служи за брисање самог догађаја. Треба напоменути да је 1 понављање (*Repeat*) догађаја дефинисано једним дешавањем и истеком догађаја (дакле 1 *Raise* + 1 *Expire*).

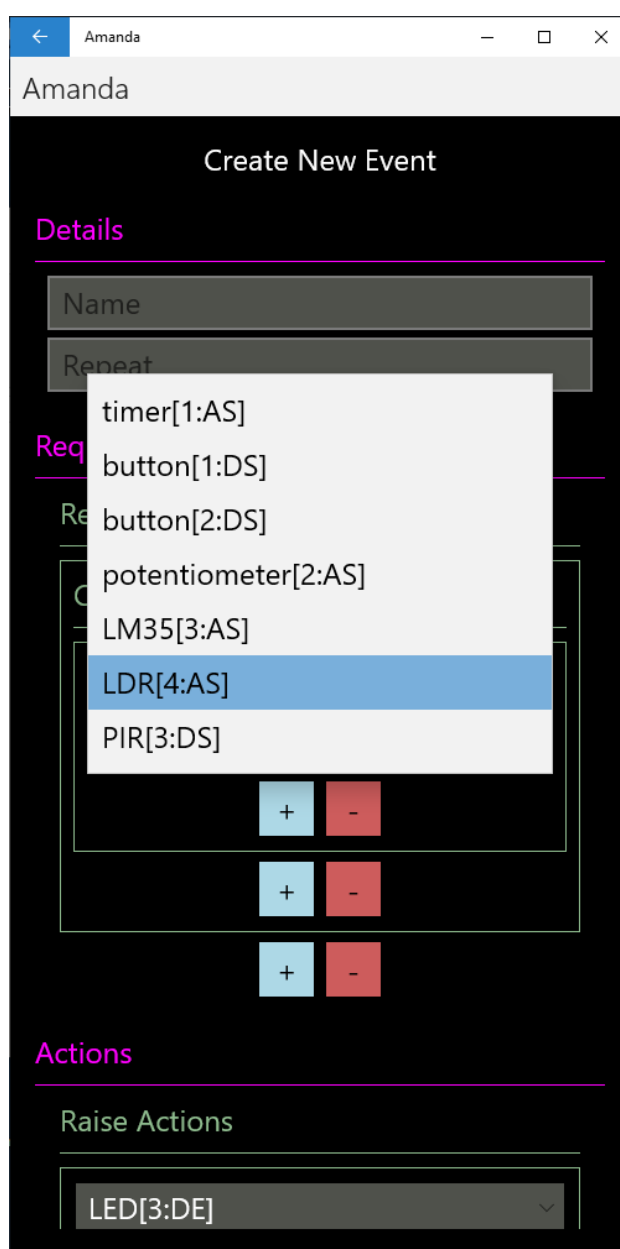
На слици 20 може се видети екран који је добијен притиском на тастер *Create New* са слике 18 (управљање догађајима, тј. *Event management*).

The screenshot shows a mobile application interface for creating a new event. The title bar at the top says 'Amanda'. Below it, the main title is 'Create New Event'. The interface is divided into three sections: 'Details', 'Requirements', and 'Actions'. The 'Details' section contains two input fields: 'Name' and 'Repeat'. The 'Requirements' section contains a 'Requirement' label and a 'Condition' section. The 'Condition' section has a dropdown menu with 'LDR[4:AS]', a comparison operator '≠', and a 'Value' input field. Below the 'Condition' section are three sets of '+' and '-' buttons. The 'Actions' section contains a 'Raise Actions' label and a dropdown menu with 'LED[3:DE]'.

Слика 20 – креирање новог догађаја (део 1)

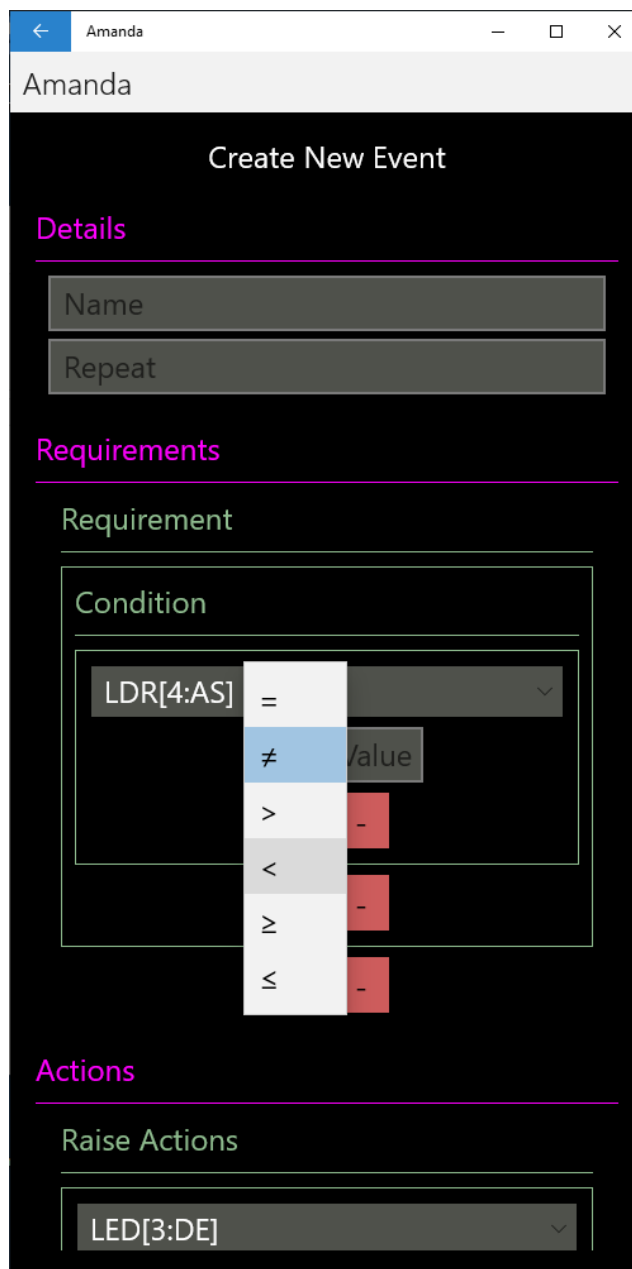
При креирању новог догађаја, неопходно је унети његово име, као и параметар *Repeat* (број понављања), који је претходно објашњен. Након тога, потребно је подесити

жељене услове и акције. У првом одељку налази се могућност подешавања услова. Овде је битно напоменути неколико ствари. Могуће је креирање више услова (тј. *Requirement-a*), и веома је битно да се назначи да је довољно само да је један од тих услова тачан (дакле један *Requirement*), и да ће се целокупан услов сматрати тачним и тиме догађај десити. Ово је аналогно логичкој операцији „или“. Само када ни један услов (*Requirement*) није испуњен, догађај истиче (енгл. *Expire*). Овим је могуће правити логику таква да подржава „или“ логику. У оквиру сваког услова (*Requirement-a*), могуће је наводити (више) стања (енгл. *Condition*). Да би услов (*Requirement*) био испуњен, неопходно је да сва стања (*Conditions*) у оквиру њега буду испуњени. Ово је аналогно логичкој операцији „и“. За свако стање везана је једна улазна компонента (односно један сензор), за кога је могуће везивати више операција која пореде његово стање са референтним које се овде и наводи. На слици 21 може се видети избор сензора (било аналогни или дигитални) који утиче на коначни услов.



Слика 21 – избор сензора у догађају

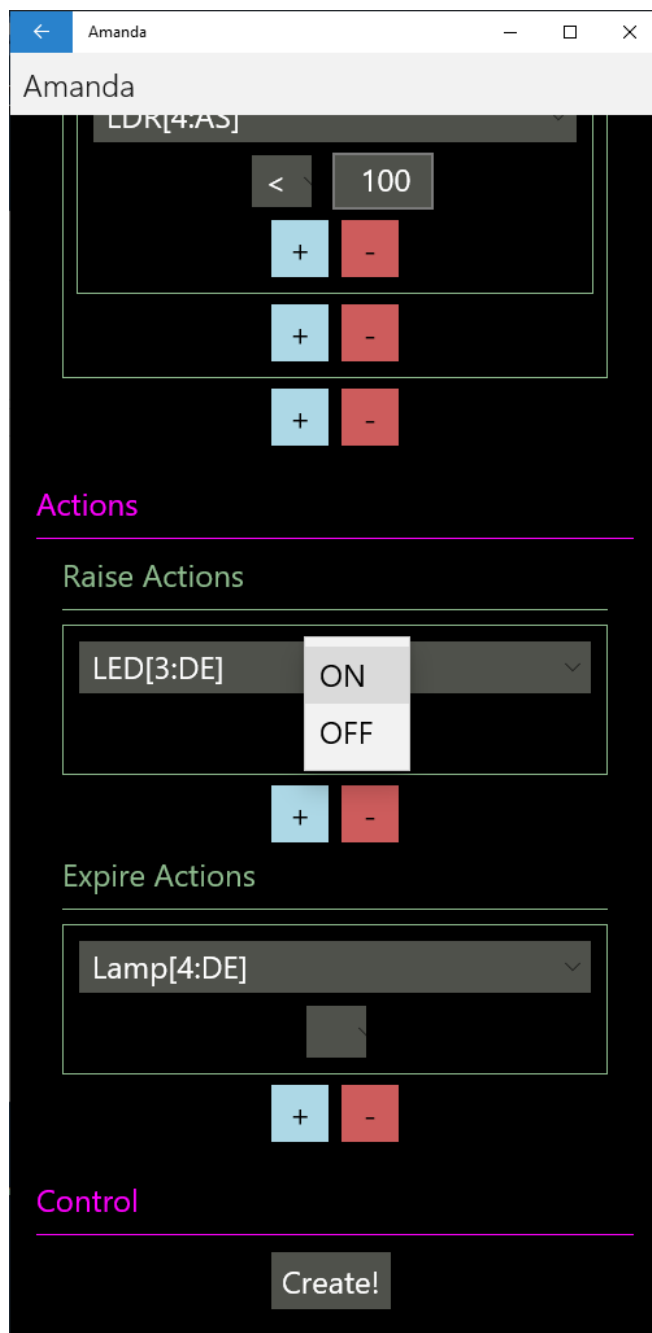
По избору сензора, као што је већ речено, могуће је везивати више оператора за то стање. На слици 22 приказан је избор оператора.



Слика 22 – избор оператора

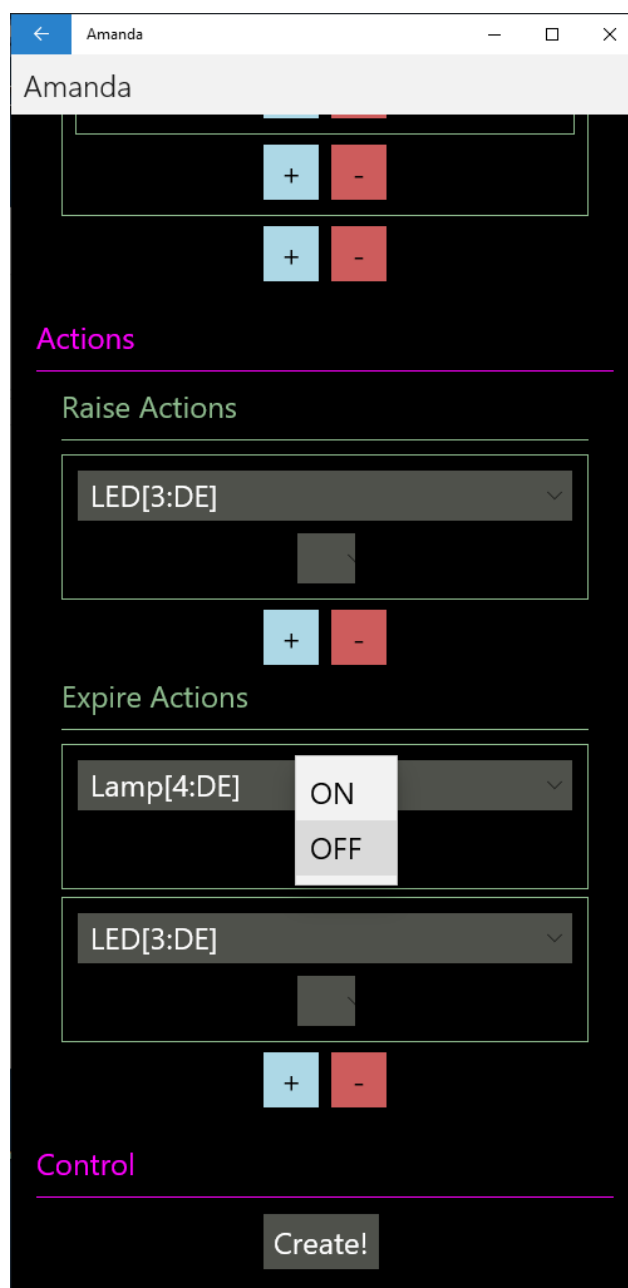
На одговарајуће „+“ и „-“ тастере могуће је дакле додавати нове услове, стања и операторе, као и брисати постојеће. На пример, могуће је додати два оператора за стање индиковано сензором осветљења, тако да нпр. један оператор ограничава да показивање *LDR* буде мање од 100 lux, а друго веће или једнако 50 lux. Такође, и између оператора за стање стоји логичка операција „и“.

Код акција, као што је већ поменуто, могу се управљати излазне компоненте, односно елементи. На слици 23 може се видети да је за *Raise Actions* одабрана *LED* компонента за управљање, а отворен је избор за стање, па нпр. може се дата *LED*-овка укључити.



Слика 23 – подешавање *Raise Actions*

У оквиру акција може се наводити више компоненти које би се контролисале. Тако на пример, на слици 24, може се видети да под *Expire Actions*, одабрана је и *LED* као у *Raise Actions*, али такође и лампа.



Слика 24 – подешавање *Expire Actions*

Кликом на одговарајуће тастере „+“ и „-“ додају се нове компоненте за управљање, односно бришу постојеће. Након што се све подеси по жељи, кликом на тастер *Create!* креира се догађај на серверу, а апликација избацује поруку уколико је дошло до грешке, или уколико је догађај успешно креиран, а затим се враћа на претходну страницу (*Event management*) са слике 18.

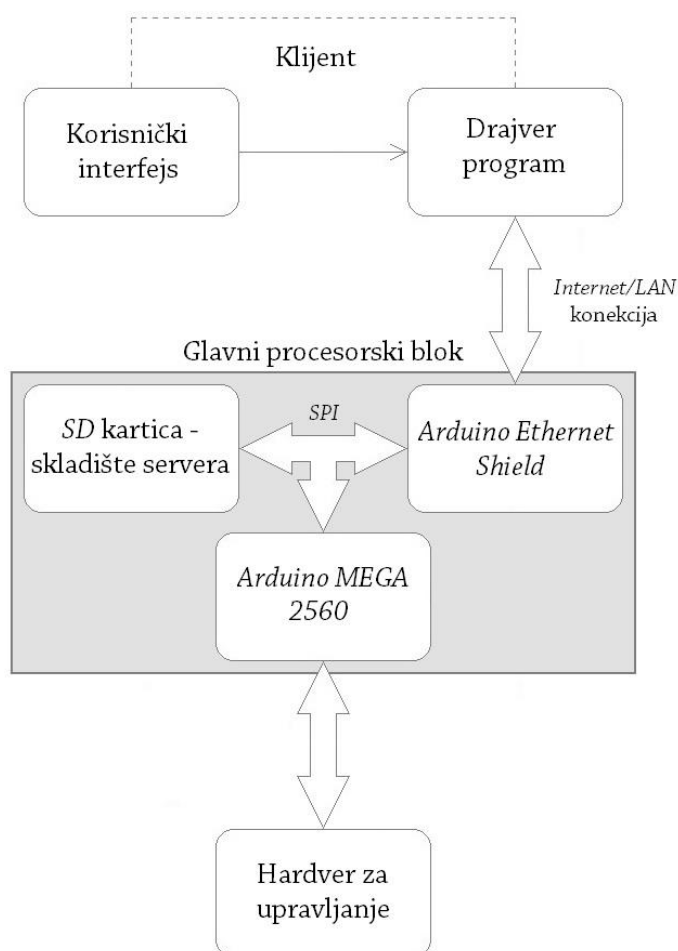
Такође, треба напоменути да сам уређај поседује меморијску *micro-SD* картицу, на којој се ови догађаји и чувају, па тако, након рестартовања и/или губитка напајања уређаја, догађаји перзистирају и опстају.

## 5. РЕАЛИЗАЦИЈА СИСТЕМА

У овом поглављу биће приказана имплементација самог система и то тако што ће бити приказане неке од најважнијих компоненти самог система. У првом одељку ће бити приказана архитектура самог система, док ће у другом одељку бити приказани неки од најважнијих имплементационих захтева приликом израде овог система.

### 5.1. АРХИТЕКТУРА СИСТЕМА

На слици 25 може се видети општа блок шема принципа рада система. Уређај треба да омогући да му се преко апликације путем интернета приступи. Ово може бити кроз локалну рачунарску мрежу (енгл. *LAN*) или на глобалном нивоу (енгл. *WAN*). Крајеви ове конекције јесу *Ethernet Shield* и сама клијентска апликација која се извршава или на мобилном телефону или на персоналном рачунару. На уређају се поред *Ethernet Shield*-а, налази и *micro-SD* меморијска картица, која представља складиште података сервера. Ту се наравно налази и *Arduino MEGA 2560* као главни микроконтролер и периферни хардвер кога, индиректним путем преко главног процесорског блока, корисник управља кроз клијентску апликацију.



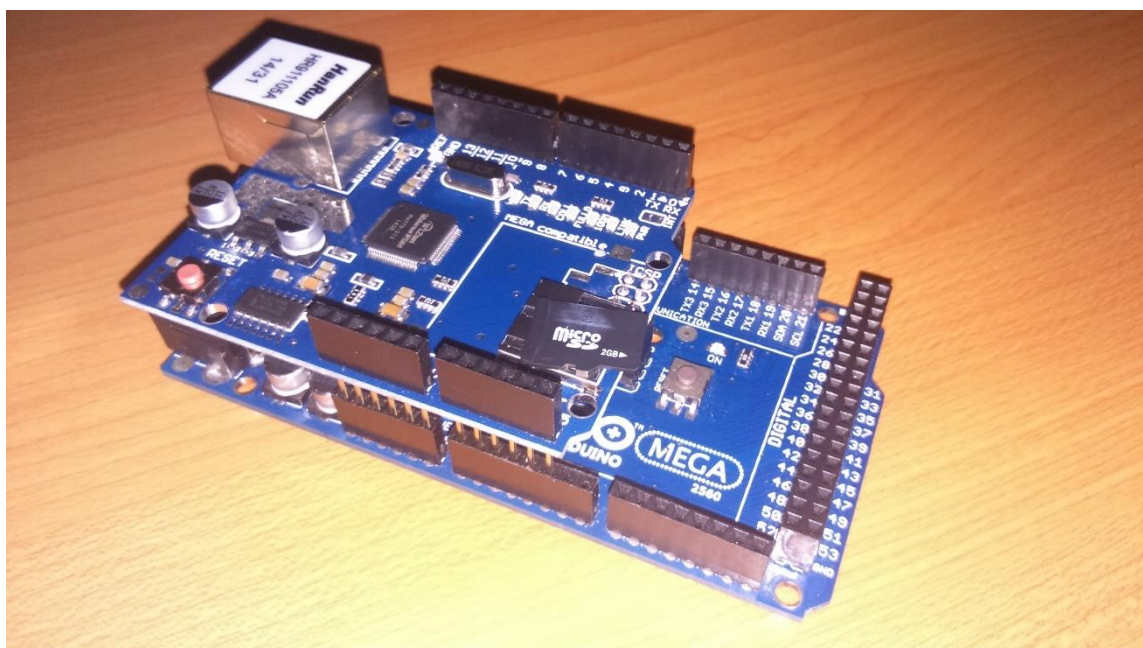
Слика 25 – архитектура и принцип рада система

## 5.2. ХАРДВЕР УРЕЂАЈА

У овом одељку биће објашњен комплетан хардвер уређаја. Првенствено ће бити речи о главном процесорском блоку, а затим ће се обрадити периферни хардвер уређаја. Такође, биће дата блок шема уређаја, као и листинг са пиновима и конектованим тачкама.

### 5.2.1. ГЛАВНИ ПРОЦЕСОРСКИ БЛОК (ГПБ)

Под главним процесорским блоком подразумева се уоквирени део слике 25 где је представљена архитектура система и принцип његовог рада, тачније спој *Arduino MEGA 2560* микроконтролера са додатком *Arduino Ethernet Shield* и складишта података кога чини *micro-SD* картица. На слици 26 се може видети изглед ГПБ-а (*SD* картица није прикључена).



Слика 26 – изглед главног процесорског блока

ГПБ представља главни део уређаја. Он има двоструку улогу: улогу *PLC*-а и улогу сервера на рачунарској мрежи. Улога *PLC*-а је неопходна како би управљање над периферним хардвером било могуће. Под периферним хардвером подразумевају се до сада помињани сензори, електронски елементи, готови уређаји, па и други микроконтролери, односно рачунари који могу да остваре било какву размену података са *Arduinom*, односно, постоји неки вид интерфејса између њих. Улога сервера јесте да омогући приступ уређају, како би корисник могао да оствари контролу. Клијентска апликација комуницира путем интернета преко протокола дефинисаног у тачки 3.1. са сервером и на тај начин размењује податке.

Као што му је улога двострука, тако и сам програм у њему треба да функционише. Неопходно је константно пратити сервер, тачније пратити да ли постоји захтев упућен серверу који још није процесирао, односно клијент који је упутио дати захтев серверу, а да још није добио одговор. На овај начин остварена је комуникација између сервера и клијента. Затим, неопходно је комуницирати са периферним хардвером. Под овим се



подразумева поновно читавање стања сензора, укључивање/искључивање уређаја/елемената, процесирање подешене аутоматике (догађаја) и слично.

#### 5.2.1.1. ГПБ као *PLC*

Улога ГПБ-а као *PLC* своди се на управљање и контролу хардвера који је на било који начин повезан са *Arduinom*. Овај хардвер је подељен у две велике групе: сензори и елементи. Улога сензора је да прикупљају податке (и представљају улазне компоненте система) као што је температура, ниво осветљености, детекција покрета и друге физичке величине. Преко њих је реализована аквизиција података. С друге стране, елементи, служе као средства за управљање (излазне компоненте). Елементи могу бити звучници, сијалице, вентилатори, грејачи и било какав управљив хардвер. Под управљањем се може узети у обзир пуштање тонова одређених трајања на звучницима, укључивање/искључивање сијалица, вентилатора, грејача и друге функције у зависности од самог употребљеног хардвера.

ГПБ треба да обезбеди константно ажурирање хардвера. Под овим се сматра читавање стања свих сензора како би сама аквизиција била валидна. Такође, потребно је пратити и услове и целокупну логику која је задата од стране корисника како би се остварио прави ефекат аутоматског управљања (систем догађаја). Као што је већ поменуто, догађаји се чувају на складишту сервера како би опстала и након искључивања уређаја.

#### 5.2.1.2. ГПБ као СЕРВЕР

Без приступа уређају корисник не би имао никакву контролу над њим јер апликација не би могла да се повеже са уређајем, преко које се контрола и врши. Ово је главни разлог зашто ГПБ мора да функционише и као сервер на рачунарској мрежи. Јавност самог сервера је подешљива, може бити видљив само у локалној рачунарској мрежи (*LAN*), а може бити и глобално видљив на интернету (ово подешавање се не обавља директно на уређају, већ на рутеру или свичу на кога је сам уређај повезан).

До сада је објашњен принцип рада протокола у комуникацији између клијента и сервера (апликације и уређаја). Уређај прима захтеве послате од клијента, а затим га обрађује и шаље кориснику назад одговор. Захтеви могу бити разноврсни, почевши од захтева за скенирање читавог хардвера (где је одговор сет свих сензора заједно са њиховим детаљима и вредностима које су са њих прочитане, као и елемената који стања, односно вредности које су на њих подешене), за креирање новог догађаја, за конектовање на уређај (*hello* и *echo/reply* поруке), за извршавање одређених команди и друге. Када се захтев упути ка серверу, он је дужан да извуче битне информације како би знао на који начин треба да одговори и шта треба да уради. Типови одговора такође има много. Међутим, најчешће су то одговори попут јављања да је захтев успешно обрађен, или да је дошло до неке грешке, као и порука саме грешке која је кориснику читљива и разумљива. Треба напоменути да сервер не садржи странице, с обзиром да се ради о *http* протоколу, стога је неопходно увек слати *POST* захтев на „празну“ страницу (тачније *index* страницу, али не треба ју је наводити јер је она подразумевана). О овоме ће бити више речи у софтверском делу реализације сервера, као и конкретне поруке самог протокола.

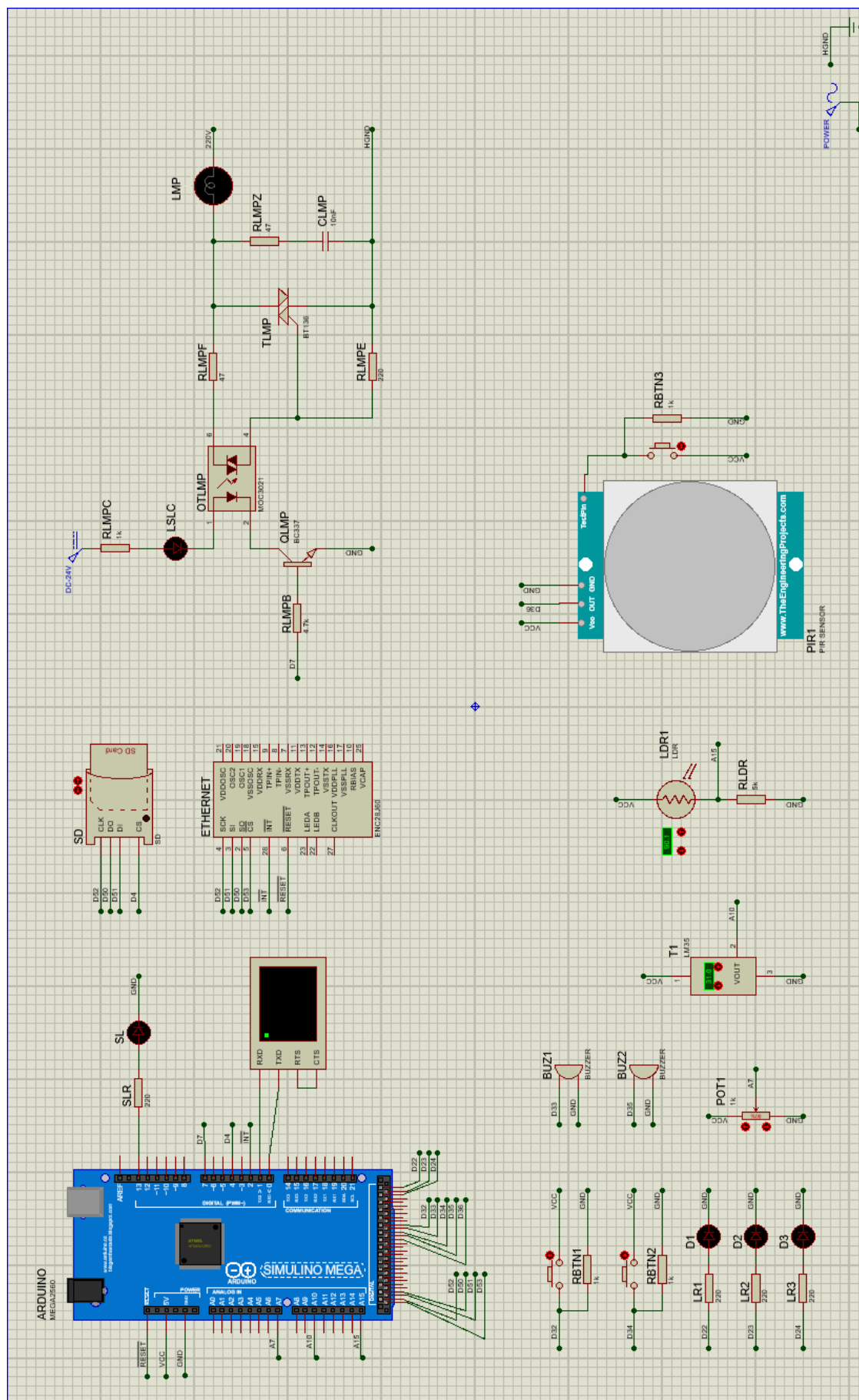
### 5.2.2. ПЕРИФЕРНИ ХАРДВЕР

Веома је битно напоменути да периферни хардвер јесте предефинисан, али није фиксиран. Другим речима, не постоји фиксиран сет хардвера који увек долази уз сам уређај, међутим, онај који се одабере, у програму мора бити предефинисан, као и назначен сет пинова где се која компонента налази. Међутим, у даљем развоју система, веома лако се може направити и нека врста динамичког распознавања периферног хардвера где онда он не мора више бити ни предефинисан, али ово обично може да се примени само за комплексније уређаје који спадају у периферни хардвер, који имају могућност „паметније“ комуникације, како би самом уређају јавили да су се конектовали на њега, шта они представљају и где се налазе. То обично повлачи да је неопходна отворена комуникација ка уређају, или бидирекциона, или нека врста комуникације преко магистрале, што усложњава систем.

У овом одељку ће бити дат само пример могућег периферног хардвера који треба да илуструје принцип рада овог дела система. У наставку дат је листинг хардверских компонената које су коришћене за периферни хардвер, а након тога, на слици 27, дата је и блок шема уређаја.

- 2 тастера
- 3 *LED* (црвена, жута, зелена)
- 1 лампа (у симулатору, а на протоборду искоришћена *RGB LED* као занема)
- 2 *buzzer*-а
- 1 потенциометар
- 1 термометар *LM35Dz*
- 1 *LDR*
- 1 *PIR*

Поред овог периферног хардвера и поред главног процесорског блока (*Arduino MEGA 2560*, *Arduino Ethernet Shield*, меморијска картица *micro-SD* од 2 GB), треба напоменути да је коришћен и рутер *ASUS RT-N12E*.



Слика 27 – блок шема комплетног уређаја

У симулатору не постоји концепт *Arduino Ethernet Shield*-а, али у ту сврху користи се компонента *ENC28J60* која интерфејсно идентично ради. Као што је већ речено, главну позицију у целокупном систему заузима ГПБ. Све остале периферне компоненте повезане су за њега. У табели 1 представљено је пин на пин повезивање свих компоненти.

Лампа	Развојна картица	Тастер 1 (1 kΩ)	Развојна картица
Улаз	D7	Крај 1	5 V
Маса	GND	Крај 2	D32, GND
LED 1 (220 Ω)	Развојна картица	Тастер 2 (1 kΩ)	Развојна картица
A	D12	Крај 1	5 V
K	GND	Крај 2	D34, GND
LED 2 (220 Ω)	Развојна картица	Buzzer 1	Развојна картица
A	D13	+	D33
K	GND	-	GND
LED 3 (220 Ω)	Развојна картица	Buzzer 2	Развојна картица
A	D14	+	D35
K	GND	-	GND
PIR	Развојна картица	Потенциометар	Развојна картица
Vcc	5V	5V	5V
Vout	D36	Клизач	A7
GND	GND	GND	GND
LDR + 5 kΩ	Развојна картица	LM35Dz	Развојна картица
Крај 5 kΩ	5 V	Vcc	5 V
LDR + 5 kΩ	A15	Vout	A10
Крај LDR	GND	GND	GND

Табела 1 – повезивање компоненти система

## 5.3. СОФТВЕР СИСТЕМА

У овом одељку првенствено ће бити објашњен софтвер уређаја, где ће бити поменуто и неколико детаља о протоколу. Затим, биће дати сви детаљи протокола, све врсте порука и њихови параметри. Потом, биће разјашњена имплементација саме клијентске апликације као пример апликације која комуницира са уређајем по дефинисаном протоколу.

### 5.3.1. СОФТВЕР НА УРЕЂАЈУ

На слици 10 у тачки 3.3.3. су били представљени сви под-пројекти система, где је фокус био на прва 4 који су представљали пројекте везане за клијентску апликацију, а који (осим протокола) нису имали никакве додирне тачке са самим уређајем (што је и био циљ од почетка). Наредних 5 под-пројекта, директно или индиректно везани су за уређај, а то су:

- *Concepts*
- *Core*
- *Device*
- *Suit*
- *Tests*

Прва 3 пројекта из листе (*Concepts*, *Core* и *Device*) су пројекти који су директно везани за уређај. Преостала два (*Suit* и *Tests*) су пројекти који су везани за тестирање. Као и сваки други софтверски продукт, неопходно је постојање неког вида тестирања како би коначни исход поседовао одређени ниво квалитета. Из овог, а и из других разлога, ова два пројекта су веома битна, а о њима ће бити више речи касније.

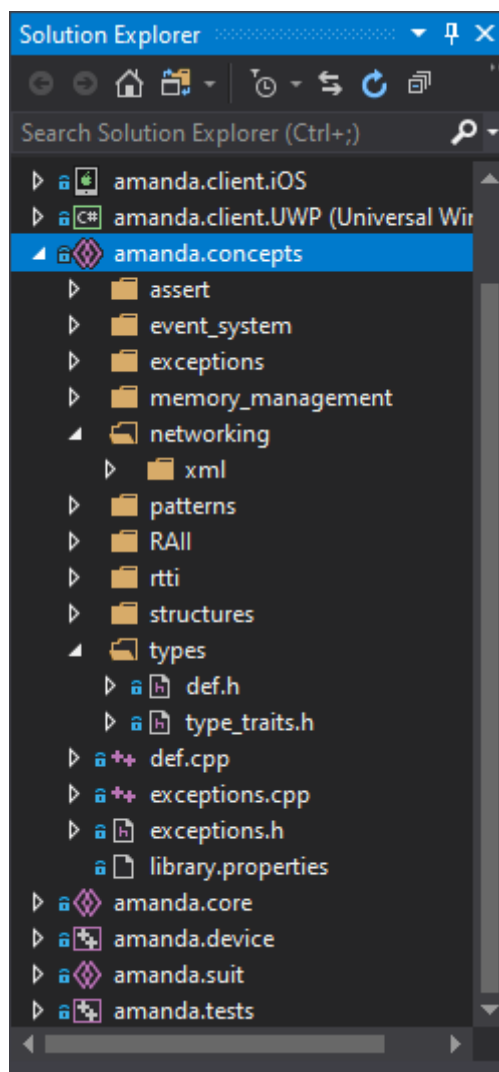
#### 5.3.1.1. CONCEPTS

Као и многе друге платформе и *framework*-ци, и *Arduino* има свој сет библиотека и концепата који се могу користити при изради софтвера за пројекте. Међутим, постоје нека ограничења која се директно или индиректно због тога намећу. Поред тога, уграђени (енгл. *Embedded*) системи често имају мало меморије (нпр. *Arduino MEGA 2560* има свега 8 KB *SRAM*) и то без одређених бољих техника за управљање меморијом попут виртуелне меморије и слично. Даље, како се ради о ниском (енгл. *low-level*) програмирању, многи концепти који постоје у вишим језицима овде не постоје.

Ово су неки од главних разлога за постојање овог пројекта. Поред тога, како је већ постојала потреба за њим, осмишљено је да овај пројекат у што већој мери садржи код који није директно везан за једну платформу (на неки начин да буде *cross-platform*), тј. није везан за *Arduino*, а ово ће бити разјашњено касније.

Главна улога овог пројекта је да представља библиотеку разноврсних концепата који могу да се користе генерално у језику *C++*, па и у *Arduino*-у. Отуда, овај пројекат може бити креиран као статичка или динамичка библиотека, међутим, на крају, одабран је принцип дељеног пројекта (енгл. *Shared Project*), што значи да било који други основни (енгл. *Root*) пројекат који жели да користи бенефиције дељеног, попут библиотека, мора и сам дељени пројекат да укључи, међутим, разлика је само што се дељени пројекат сам по себи не преводи, штавише самостално ни не може да стоји, већ му је неопходан

основни који по његовом укључивању и по превођењу, преводи дељени пројекат као да он сам није одвојен већ писан у потпуности као да припада основном пројекту. Разлог што се зове дељени је управо што због претходно наведеног принципа, више основних могу на овај начин да укључе дељени пројекат и сваки понаособ да сматра као да он засебно поседује дељени, и на тај начин свако користи његове бенефиције, а програмерски добитак се заснива на томе да је лакше одржавати систем јер нема копија кода на местима где се исти кроз различите пројекте више пута користи. На слици 28 може се видети структура овог пројекта кроз *Solution Explorer*.

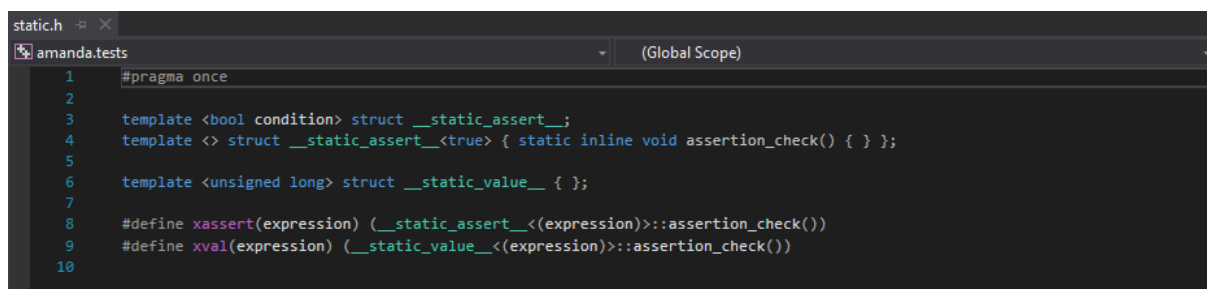


Слика 28 – структура *Concepts* пројекта

Као што је већ речено, овај пројекат представља библиотеку разних концепата. Овај пројекат је писан тако да није зависан директно од платформе. Већински је намењен за платформе које имају мало меморије, попут *Arduino* платформе (али ово никако не представља ограничење, већ побољшање, јер га могу користити и платформе са мало меморије, али исто тако и оне које имају много). Конкретно, *Arduino* за операторе *new/delete* који практично управљају алокацијом/деалокацијом динамичке меморије користе алгоритме који су базирани на принципу сличним као *best-fit* или *first-fit* који узрокују фрагментацију меморије, што доводи до пада система након и краћег времена

јер није могуће испунити меморијске захтеве софтвера. Ово је можда и највећи имплементациони изазов овог пројекта. Дакле, било каква динамичка алокација кроз стандардне операторе *new/delete* је недопустива, и самим тим, неопходно је програмирати на много стриктнији начин.

Неке од платформи у уграђеним системима не садрже статичка тврђења (енгл. *Static Assert*). Стога, у овом пројекту, налази се једна од могућих имплементација која се може видети на слици 29.

The image shows a code editor window with a file named 'static.h'. The code defines two templates: one for a static assertion condition and another for a static value. It also defines two macros, 'xassert' and 'xval', which use the templates to perform assertions at compile time. The code is as follows:

```
1 #pragma once
2
3 template <bool condition> struct __static_assert__;
4 template <> struct __static_assert__<true> { static inline void assertion_check() { } };
5
6 template <unsigned long> struct __static_value__ { };
7
8 #define xassert(expression) (__static_assert__<(expression)>::assertion_check())
9 #define xval(expression) (__static_value__<(expression)>::assertion_check())
10
```

Слика 29 – имплементација *Static Assert*-а

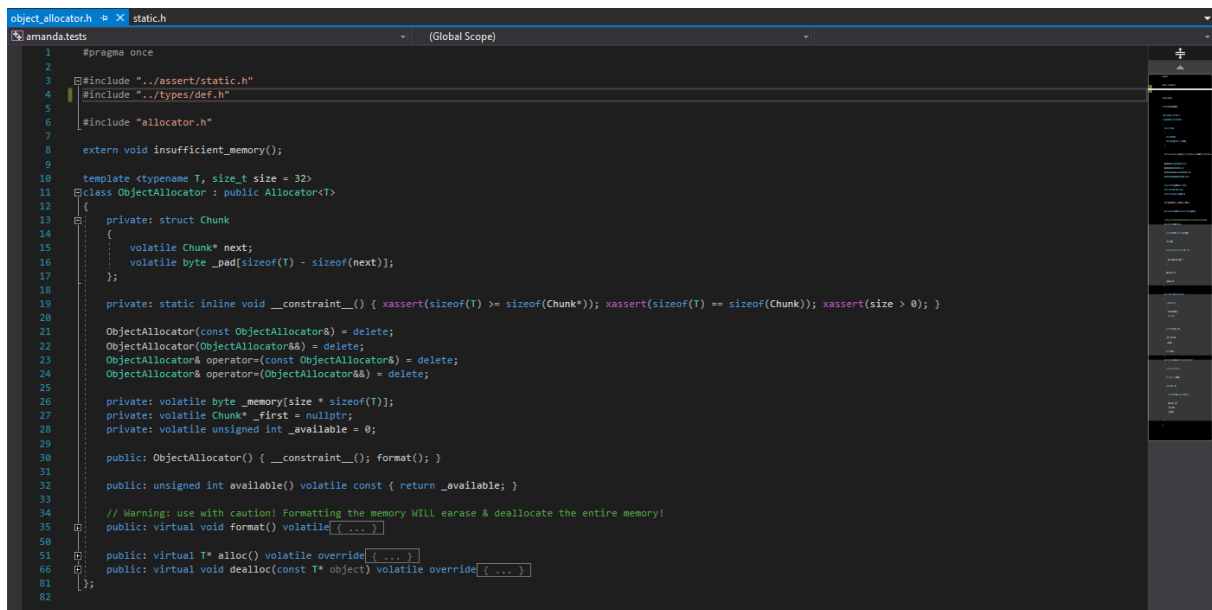
Кроз макрое *xassert* и *xval* могућа је провера израза за време компајлирања, тако да уколико постоји нека грешка, компајлирање ће се зауставити. Постоје грешке које се дешавају само у извршавању (енгл. *Runtime*), које заправо представљају и главнину грешака, а да су синтаксно и семантички кроз компајлер исправне. Међутим, неке од њих се у времену извршавања или јако тешко виде или их је немогуће пронаћи. Међутим, оне често произилазе из таквих околности да је део кода намењен за рад само под одређеним условима. Ти услови се некад могу случајно променути, тако да програмер њих не уочи. У том случају, добра пракса је да се сам програмер обавести да је дошло до промене и да се сам процес компајлирања обустави. Управо овај ефекат се може постићи коришћењем овог концепта.

У овом пројекту имплементиран је и концепт *event* система. Овај систем нема никакве везе са догађајима описаним у претходним тачкама, већ представља концепт у програмирању који подржава пројектни узорак *Observer*. Исти концепт постоји у језику *Microsoft .NET C#*, који је био инспирација да се и у овом, нижем окружењу имплементира, јер је веома робустан концепт (почевши од самих делегата).

Затим се у пројекту може видети имплементација лаких (енгл. *lightweight*) изузетака (енгл. *exceptions*). Наиме, *Arduino* садржи концепт правих изузетака, али се они не препоручују и у почетку су искључени (ручно је могуће укључити их). Стога, имплементиран је концепт лаких изузетака који нису једнако јаки као прави изузеци али на оваквим системима са мало меморије могу представљати одличну замену такву да поред меморијских захтева, не утичу ни на временске перформансе, али цена је да се „померање у десно“ проблем и даље јавља (који се иначе правим изузецима избегава).

Већ је поменуто да је стандардно управљање меморијом лоше због фрагментације (конкретно на *Arduino* платформи, али тако исто може бити и на другим). Стога, имплементиран је и концепт управљања меморијом које је слично *Slab* систему. Читав систем се окреће око главне класе, а то је *ObjectAllocator*, који се може видети на слици 30.



Слика 30 – имплементација *ObjectAllocator*-а

Функционише на принципу да се преалоцира низ од одређеног броја елемената који се статички у време компајлирања задаје, тако да су сви елементи исте величине (што је иначе битно за низ), и то представља алокатор **једног** типа објекта (тј, једне класе). Овај алокатор поседује методе за динамичку алокацију и деалокацију оваквог типа објекта, као и методу која форматира читаву меморију, која треба да се употребљава уз опрез. Ограничење (статичким тврђењем) је да величина објекта класе мора бити најмање једнако величини једног показивача у самом систему (обично 4 В). Разлог за овим је имплементациони.

Поред овог алокатора, постоје специјализовани по имену *ReinterpretingAllocator*, *CastingAllocator* и *TupleAllocator*, који имају своју специјализовану намену, али у позадини сви користе *ObjectAllocator* за основно управљање меморијом.

Затим, у пројекту се може наћи део за мрежно програмирање (*networking*). Конкретно, овде се налази имплементација XML парсера. Већ је поменуто да се у систему користи XML. Наиме, постоје многе библиотеке где су овакви парсери максимално имплементирани и тестирани. Међутим, бројни разлози су постојали због чега је посебна имплементација парсера била неопходна. Постоје два типа XML парсера, а то су DOM и SAX. DOM (*Document Object Model*) парсери су недопустиви за системе са мало меморије јер читав XML парсирају у стабло које праве у меморији што и за мање фајлове могу да настану превелика стабла која заузимају превише меморије (или у овом случају ни нема довољно меморије), или захтевају објекте разних величина што је веома тешко покрити па, чак и да у систему има довољно меморије, узрокују већ поменуто фрагментацију што је недопустиво. Стога, решење је коришћење SAX парсера. SAX (*Simple API for XML*) су парсери који функционишу на принципу *pull-event*. Ови парсери дакле поседују *event* интерфејс такав да се *handler*-и позивају оног тренутка када сам парсер детектује одређени склоп самог XML-а. За само парсирање узимају карактер по карактер, па не морају ни да прочитају цео фајл од једном како би успели да испарсирају све, а такође не праве никакво стање, односно контекст парсираног XML-а, па меморијски нису захтевни. Мана је што корисник сам мора да прави овај контекст, што повећава програмерски

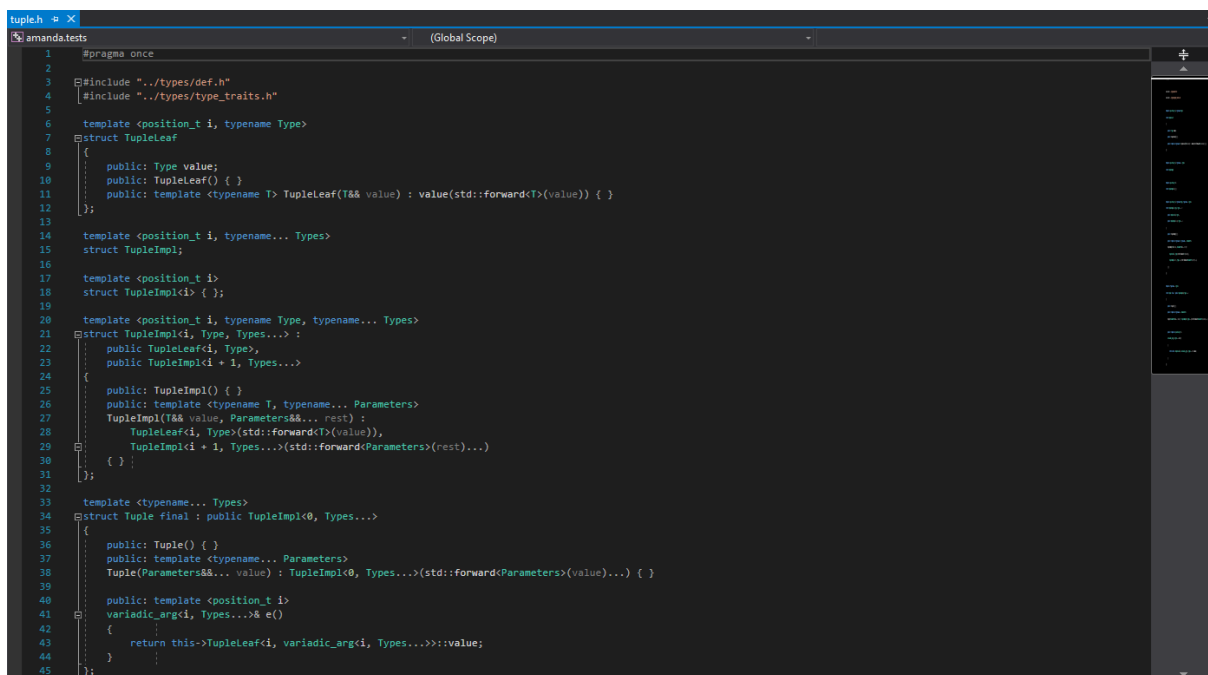


напор, али предност је што тај контекст може да оптимизује за своју примену што доводи до огромног раста меморијских перформанси (често избегавање потребе динамичког алоцирања, а такође и меморијски захтеви драстично опадају). Често се пројектни узорак *Builder* може спојити са овим парсером како би се сам *API* прилагодио и направио много флексибилнијим, а тиме потенцијално и још у малој количини допринело меморијским перформансама. Постоје библиотеке које већ имплементирају *SAX* парсере, али нису компатибилни са *Arduino* платформом, па су ово главни разлози зашто је постојала потреба за новом имплементацијом. Ова имплементација не подржава све концепте *XML*-а, већ најосновнију варијанту која је била неопходна за само функционисање система (нпр. не подржава *namespace*-ове *XML*-а, затим *CDATA* делове и слично).

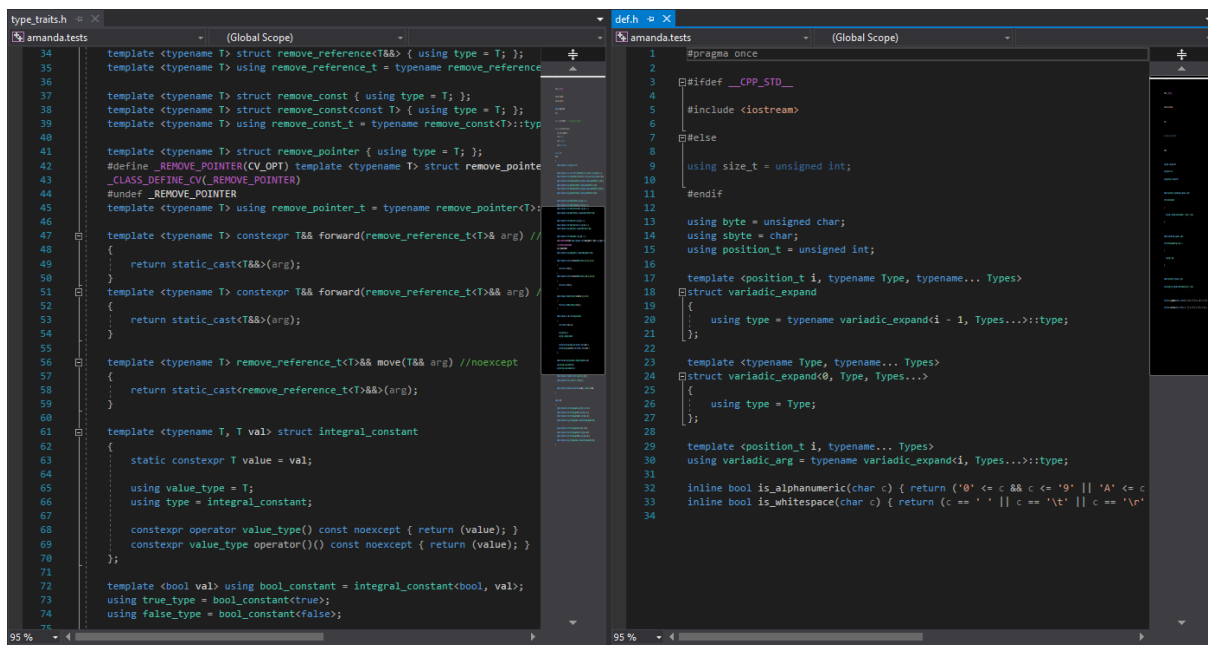
У *Concepts* пројекту такође су имплементирани и основни *API*-и за неке од стандардних пројектних узорака (енгл. *Design Pattern*), а то су конкретно *enumerator* (односно *iterator*) и *singleton*. Иако технички не представља узорак (стога је издвојен у засебан фолдер *RAII*), имплементиран је и *RAII* концепт који се може користити у складу са својим наменама (имплементирана је генеричка варијанта која треба да покрива *RAII* опште намене).

Један од већих и јачих концепата виших језика јесте *Reflection*, који је заснован на информацијама у време извршавања (енгл. *Runtime*). Језик *C++* у својој стандардној варијанти чак и садржи модул *RTTI* (*RunTime Type Information*). *Arduino* такође поседује овакав концепт, али то представља превише захтеван и тежак (енгл. *heavyweight*) концепт као што су изузеци (мада је *RTTI* још веће тежине), па у овом пројекту, имплементирана је *lightweight* варијанта овог концепта, која наравно има своја ограничења, а базирана је на коришћењу виртуелних метода на местима где је виртуелна табела у меморији већ креирана, па се тиме не губи на перформансама.

Као и други *framework*-ци, и у овом пројекту су имплементиране стандардне структуре и колекције попут листе, вектора, реда и слично. Треба напоменути да стандардне имплементације користе операторе *new/delete* за динамичку алокацију, па је препоручљиво да се избегавају уколико се пројекат користи уз системе са мало меморије. Стога, у овом делу пројекта постоје специјализоване имплементације истих структура и колекција (листе највише), које користе већ поменут концепт *ObjectAllocator*-а за динамичку алокацију па се избегава коришћење оператора *new/delete*. Једна од интересантних структура која је имплементирана у овом контексту јесте *tuple*, која се може видети на слици 31, а заснована је на сопственом извођењу читавог стабла генеричке класе.

Слика 31 – имплементација *tuple* структуре

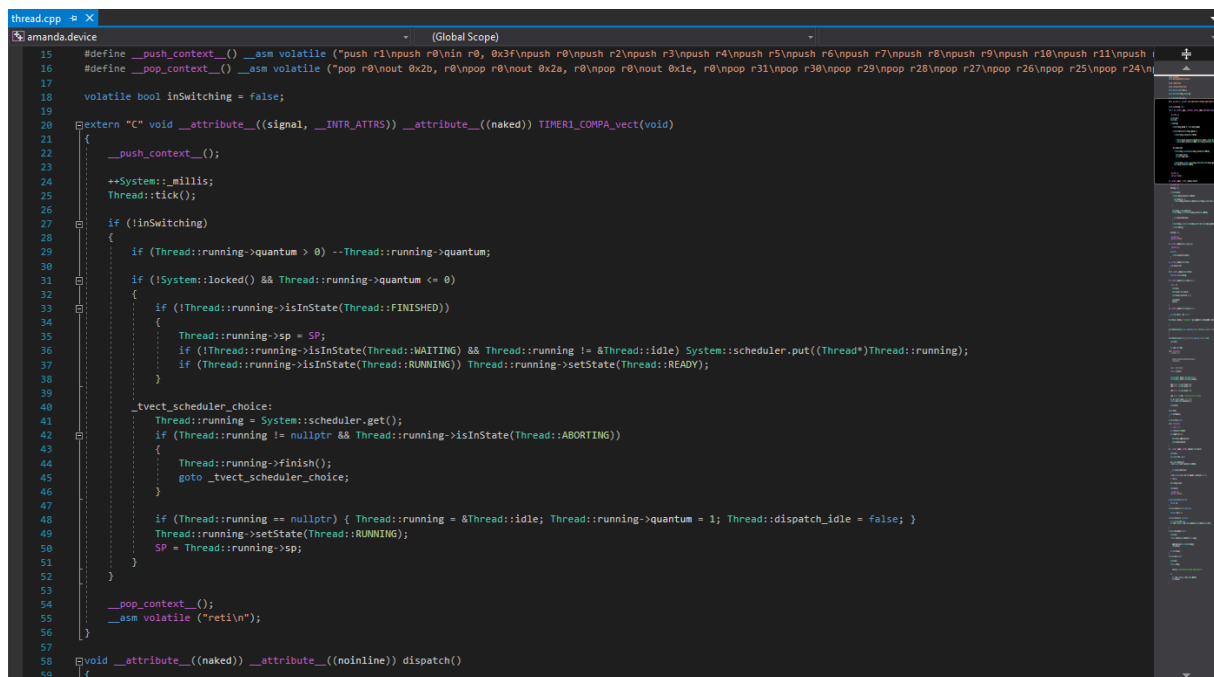
Такође, у овом пројекту могу се наћи неке од компајлерских пречица за постизање разних компајлерских ефеката попут генеричког варијадичког експандовања, типског евалуирања попут уклањања референци од генеричког типа како би се добио основни тип (или додавања) и слично, а део се може видети на слици 32.

Слика 32 – типске особине (енгл. *type traits*)

Као што је већ речено, овај пројекат није везан за конкретну платформу, па је читав код писан тако да „не познаје“ за *Arduino*. Тиме, могуће га је користити и кроз друге платформе и у другим пројектима, нпр. у стандардном *C++* пројекту.

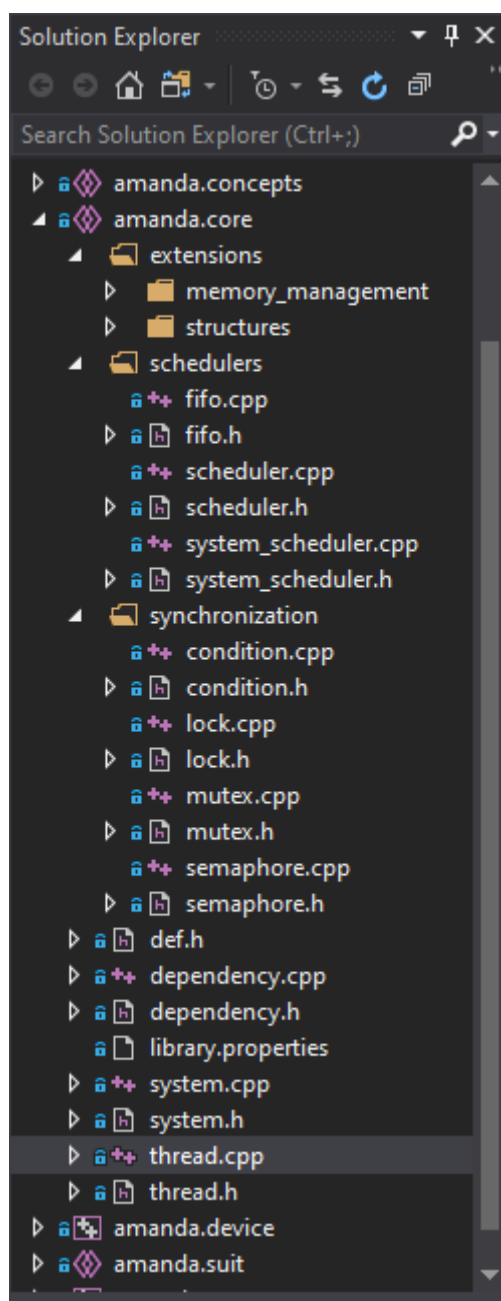
### 5.3.1.2. CORE

Као и претходни пројекат, *Concepts*, и овај пројекат представља дељени пројекат и користи се на исти начин. Међутим, за разлику од претходног, овај пројекат је ипак везан за конкретну платформу, и то за *Arduino*, и то искључиво за *Arduino MEGA 2560* развојну картицу. Овде је имплементирано језгро *context-switching* система, па самим тим, коришћењем овог пројекта могуће је правити *multi-threaded* окружење у *Arduino* пројектима. Међутим, највећи део овог пројекта је поново писан тако да не зависи директно од *Arduino* платформе, далеко само од *Arduino MEGA 2560*, али, постоје одређени (мали) делови који зависе. Уз мали напор и реорганизацију кода у тим деловима, могуће је библиотеку направити да се може користити и на другим платформама које подржавају принцип прекидних рутина, највише прекидне рутине која се окида на неку врсту *timer*-а. Наравно, за сам *context-switching*, неопходно би било за сваку платформу дефинисати шта је контекст и како се он чува/рестаурира и слично. У датотекама *thread.h* и *thread.cpp* налазе се најбитнији делови овог пројекта. На слици 33 приказана је прекидна рутина тајмера где се налази логика за мењање контекста.



Слика 33 – прекидна рутина тајмера за *context-switching*

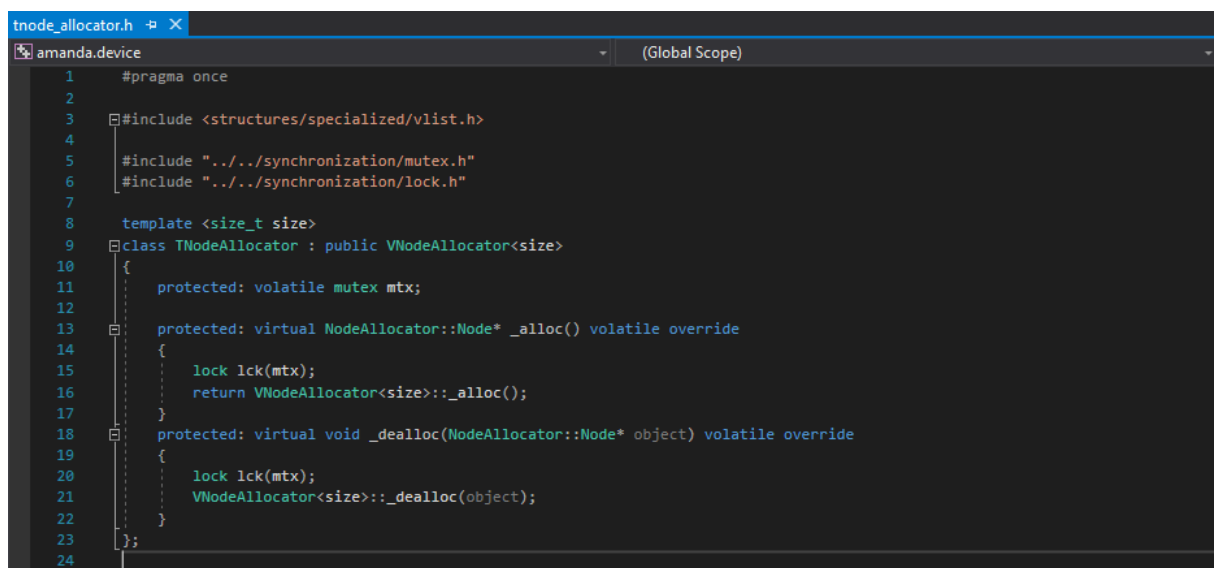
Поред самог *context-switching* система, многи концепти који су неопходни за конкурентно програмирање такође су имплементирани у овом пројекту. Концепти дају прилично сличан *API* као и *API* који постоји у стандардном *C++*-у. У питању су концепти синхронизације и међусобне комуникације нити (енгл. *thread*), а у стандардном *C++*-у то су: *mutex*, *recursive\_mutex*, *timed\_mutex*, *condition\_variable*, *lock\_guard*, *unique\_lock* и други. И у овом пројекту постоје основне и комбиноване варијанте ових концепата, прилагођене за *API* овог система, а структура пројекта, заједно са листингом до сада поменутих ствари налази се на слици 34.

Слика 34 – структура *Core* пројекта

Постоји и подсистем за распоређивање послова, односно нити (енгл. *Scheduling*), са могућношћу лаког креирања нових врста распоређивача (*Scheduler-a*). Тренутно имплементиран и коришћен распоређивач јесте овичан *FIFO* распоређивач, који функционише на *Round-Robin* принципу, јер постоји концепт преотимања (енгл. *preemption*), јер се прекидна рутина тајмера окида на сваких 1 ms, стим да свака нит добија свој квант извршавања који обично траје 50 прекида тајмера (односно 50 ms).

У овом пројекту такође су имплементирани и екстензије које представљају ништа друго него *API* који до сада помињане концепте комбинује. На пример, до сада помињана специјализована листа у *Concepts* пројекту, именом *vlist*, јесте имплементација класичне генеричке листе која не користи стандардне *new/delete* операторе за алокацију чворова листе већ има свој алокатор за ту сврху (мада се могу правити и нови алокатори па при

креирању саме листе, други тип алокатора јој се може проследити, а ово је постигнуто због униформности *API*-а коришћењем концепта наслеђивања класа и пројектних узорака). Овим речено, једна од екстензија у овом пројекту, која се налази у оквиру *memory\_management*, јесте друга имплементација горе поменутог алокатора чворова листе, који сада комбинује претходно поменути алокаатор (који користи *ObjectAllocator*), а такође и концепте међунитне синхронизације (*mutex* и *lock*), тако да обезбеди могућност коришћења алокације чворова листе, а самим тим и саме листе, кроз више нити на безбедан начин. На слици 35 може се видети овај алокаатор (са именом *TNodeAllocator*), где користи други, базични алокаатор (именом *VNodeAllocator*), који је већ описан (који користи *ObjectAllocator*).



```
tnode_allocator.h # X
amanda.device (Global Scope)
1 #pragma once
2
3 #include <structures/specialized/vlist.h>
4
5 #include "../synchronization/mutex.h"
6 #include "../synchronization/lock.h"
7
8 template <size_t size>
9 class TNodeAllocator : public VNodeAllocator<size>
10 {
11     protected: volatile mutex mtx;
12
13     protected: virtual NodeAllocator::Node* _alloc() volatile override
14     {
15         lock lck(mtx);
16         return VNodeAllocator<size>::_alloc();
17     }
18     protected: virtual void _dealloc(NodeAllocator::Node* object) volatile override
19     {
20         lock lck(mtx);
21         VNodeAllocator<size>::_dealloc(object);
22     }
23 };
24
```

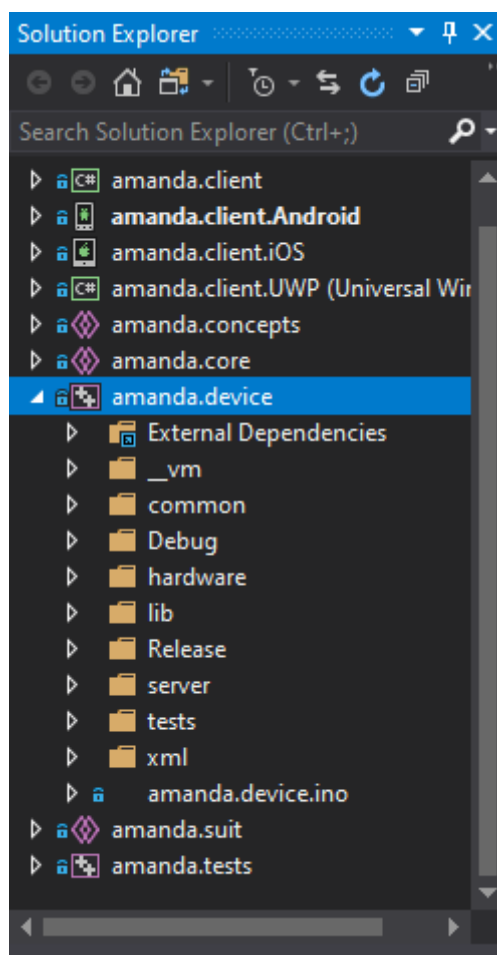
Слика 35 – *thread-safe* алокаатор чворова листе

Треба напоменути да класа *lock* користи *RAII* концепт, али не и директно раније помињану класу. То не значи да се та раније помињана класа не може користити, али класа *lock* представља специјализовану класу за ову сврху тако да се сам код поједностављује ако се она користи у оваквим околностима. У питању је принцип да се у конструктору креираног објекта прослеђени објекат *mutex* закључава, а у деструктору (практично при изласку из методе када се објекат уништава) прослеђени *mutex* откључава.

Поред свега, на неким местима се може приметити да овај пројекат користи претходни (*Concepts*) и ствари које су у њему имплементирани. Тиме, може се закључити да овај пројекат и зависи од претходног, што је и тачно. Међутим, сами дењени пројектни немају могућност да укључују друге. Овај проблем се разрешава у основном (главном) пројекту, тако што овај пројекат захтева од основног да укључи и други (*Concepts*) од кога овај пројекат и зависи.

### 5.3.1.3. *DEVICE*

Овај пројекат представља главни *Arduino* пројекат, који је основни за претходна два и оба укључује (*Concepts* из двоструког разлога, први јер га сам *Device* пројекат користи, а други јер га и *Core* користи, што је већ поменуто у претходној тачки). На слици 36 може се видети структура овог пројекта.



Слика 36 – структура *Device* пројекта

Главни фајл у пројекту јесте *.ino* фајл који генерално представља почетни фајл свих *Arduino* пројеката. Поред њега, налазе се и фолдери како са кодом, тако и други који се не тичу директно пројекта већ нпр. изграђене (енгл. *build*) варијанте пројекта (*Debug* и *Release*). По креирању пројекта, прави се и фолдер *\_\_vm*, који се тиче самог пројекта у *Visual Studio*-у и аутоматски је генерисан, са неопходним фајловима за рад у овом окружењу.

Постоје делови овог пројекта који су од мање важности. На пример, *lib* фолдер у оквиру овог пројекта заправо садржи „оптимизоване“ варијанте *string\_builder*-а и *string*-а за *Arduino*. Наиме, *Arduino* у својој библиотеци има *String* класу која се може употребити у ове ове сврхе, али, меморијски је неоптимизована и изазива фрагментацију. Стога, ове две класе су направљене, а *string64* класа представља стринг од максимално 64 карактера. С обзиром на ово ограничење, могуће је преалоцирање уместо динамичког, што избегава фрагментацију. Интересантно је да је ово сасвим довољно за веома много

примена и да се могу испунити многи захтеви у раду са стринговима (у овом пројекту за све). Поред тога, постоји и фолдер *tests*. У овом фолдеру се налазе неки од тестова који се покрећу директно на *Arduino*-у и немају никакве везе са раније помињаним пројектима који се тичу тестова, о којима ће бити речи касније. Ови тестови служе да одређене битне делове јединично тестирају.

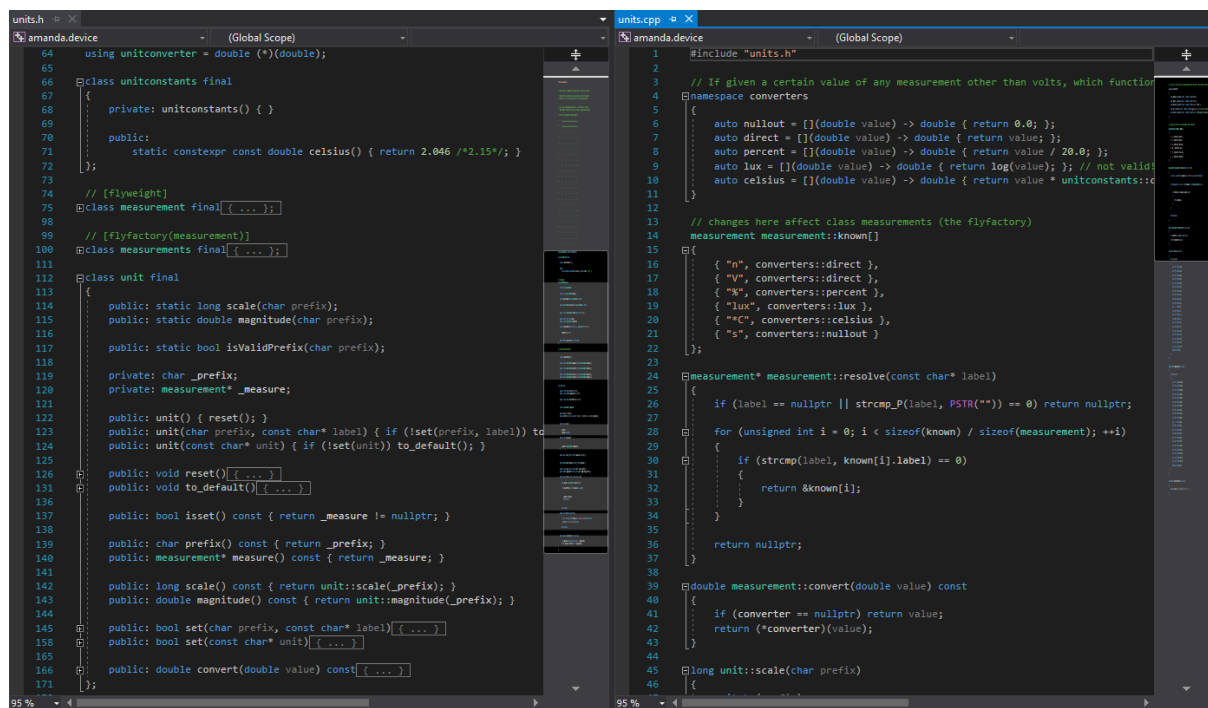
Најбитнији делови овог пројекта јесу фолдери *common*, *hardware*, *server* и *xml*. До сада је већ помињана дуална улога уређаја, као *PLC* и као сервер. Улога као *PLC* директно је везана за код у оквиру фолдера *hardware*, док је улога сервера везана за фолдер *server*.

Оба ова дела пројекта користе *common* и *xml* делове, који ће прво бити обрађени. Наиме, *xml* је релативно једноставан, и његова улога је искључиво да већ помињану имплементацију *SAX* парсера, која се налази у *Concepts* пројекту, уобличи у *API* који је прикладнији за ову примену. Другим речима, сама имплементација парсера у *Concepts* пројекту је полу процедурална, а полу објектно оријентисана. Разлог томе је да се омогући што већа контрола онима који ову библиотеку користе. Стога, постојала је и потреба да се направи *SAX API* који је у потпуности објектно оријентисан, реупотребљив и намењен за *Arduino*. Поред тога, у неким ситуацијама, било је неопходно користити више различитих инстанци парсера који граде различите контексте при парсирању, али тако да се они извршавају над истим контекстом (истим током *XML*-а, односно *stream*-ом), па је и то било неопходно апстраховати на *OOP* принципима.

Други део, *common*, садржи неке честе, уобичајене и заједничке софтверске компоненте које многи системи користе, а овде се налазе оне које су везане за *Arduino*. На пример, постоји класа у оквиру *common/communication* која се тиче серијске комуникације, а у питању је *thread-safe API* за рад са *SerialMonitor*-ем. Такође, овде се и налази концепт тока (енгл. *Stream*), који представља меморијску оптимизацију при преносу података са краја на крај, тако да није потребно користити динамичку меморију и баферовати податке за пренос кроз систем, већ је довољно преносити сам објекат тока који представља замену за бафероване податке, а на самом другом крају, где су подаци коначно потребни, могу се читати или у сам ток писати.

Још један концепт који је овде имплементиран јесте подсистем мерних јединица. Са овим подсистемом, сам систем постаје свестан мерних јединица, као и префикса, и такође може да конвертује једне вредности у друге, у зависности од оних које су јој на одређеном месту потребе и од оних које су на том месту дате, а тако да та конверзија има смисла. На слици 37 може се видети део тог система кроз фајлове *unit.cpp* и *unit.h*, а на слици 38 дати су префикси које систем разазнаје, табеларно, који се такође налазе у *units.h* фајлу.





Слика 37 – подсистем за мерне јединице

Prefix	Symbol	Using	Scale
yotta	Y	Y	+24
zetta	Z	Z	+21
exa	E	E	+18
peta	P	P	+15
tera	T	T	+12
giga	G	G	+9
mega	M	M	+6
kilo	k	K	+3
hecto	h	H	+2
deca	da	D	+1
/	/	\space	0
deci	d	d	-1
centi	c	c	-2
milli	m	m	-3
micro	\mu	u	-6
nano	n	n	-9
pico	p	p	-12
femto	f	f	-15
ato	a	a	-18
zepto	z	z	-21
yocto	y	y	-24

Слика 38 – префикси подсистема за мерне јединице



У оквиру *hardware* фолдера, налазе се имплементације свих хардверских аспеката. Једна од битних класа јесте сам *HardwareController*, класа која обједињује читав хардвер (у питању је дакле колекција објеката), и дозвољава неке од операција, као што је нпр. скенирање хардвера. Такође, ту се може наћи и подфолдер *scanning* који садржи имплементацију једног скенера компоненти, такав да на излазу (након скенирања) даје све информације записане у *XML*-у и то по протоколу тако да се директно може послати као одговор сервера оног тренутка када клијент захтева скенирање хардвера. Такође, може се и наћи фолдер *events* који садржи све имплементације неопходне за подсистем догађаја који је раније описан (подешљива аутоматика – неки од раније виђених концепата су истоимени, попут стања - *condition*).

Најбитнији детаљ контролисања хардвера јесте подфолдер *components*. Овде се налазе описи свих компоненти које систем разазнаје, тј. „драјвери“ за све познате компоненте. Донекле, овај фолдер се понаша као библиотека компоненти система. Веома је битно назначити да је овде имплементација била вођена тиме да се не фиксира сет компонената којима систем располаже, већ да се овај сет може у будућности проширивати. У подфолдеру *abstraction*, налазе се најважније апстракције компонената, од којих се основне налазе у фајлу *component.h*. На слици 39 може се видети класа компоненте, која више игра улогу интерфејса (у *C++*-у не постоји концепт интерфејса).

```
30 class IComponent
31 {
32     public: enum Type { NONE = 0, DIGITAL = 1, ANALOG = 2, SENSOR = 4, ELEMENT = 8 };
33
34     public: static const Type DigitalSensor = (Type)(DIGITAL | SENSOR);
35     public: static const Type AnalogSensor = (Type)(ANALOG | SENSOR);
36     public: static const Type DigitalElement = (Type)(DIGITAL | ELEMENT);
37     public: static const Type AnalogElement = (Type)(ANALOG | ELEMENT);
38     public: static Type resolveType(const char* name) { ... }
39
40     public: virtual ~IComponent() { }
41
42     public: virtual VID ID() const = 0;
43     public: virtual Type ctype() const = 0;
44     public: virtual const __FlashStringHelper* description() const { return F("component"); }
45
46     public: bool is(Type type) const { ... }
47     public: bool is_any(Type type) const { ... }
48     public: const __FlashStringHelper* type_str() const { ... }
49
50     public: virtual const __FlashStringHelper* commands() const { return F(""); }
51     public: virtual Command* resolve_cmd(const char* name) const { return nullptr; }
52     protected: void bind_to(Command* cmd) const { if (cmd != nullptr) cmd->component = const_cast<IComponent*>(this); }
53 };
54
```

Слика 39 – главни интерфејс сваке компоненте

Свака компонента има свој идентификатор *ID* типа *VID*, као и тип *Type* (бистки *enum*). Важно је напоменути да ова два податка заједно идентификују једну компоненту, тиме на пример, више компоненти могу имати исти идентификатор, али не и тип истовремено. Генерално, читав периферни хардвер, као што је већ речено, раздвојен је на две велике групе, а то су сензори и елементи (улазне и излазне компоненте). Све компоненте такође генерално по природи могу бити аналогне или дигиталне (али и једно и друго за комплексније компоненте и уређаје). Стога, постоје и подапстракције сензора и елемената, које се могу видети на сликама 40 и 41.

```
class ISensor : public IComponent
{
public: virtual ~ISensor() = 0;
public: virtual byte pin() const = 0;
public: virtual Type ctype() const override { return Type::SENSOR; };
public: virtual const __FlashStringHelper* description() const override { return F("generic sensor"); }
};

class DigitalSensor : public ISensor
{
public: DigitalSensor(const DigitalSensor&) = delete;
public: DigitalSensor(DigitalSensor&&) = delete;
public: DigitalSensor& operator=(const DigitalSensor&) = delete;
public: DigitalSensor& operator=(DigitalSensor&&) = delete;

private: static inline VID newcid() { static unsigned long id = 0; return (VID)++id; }
private: VID _id = newcid();

protected: byte _pin = 0;

public: explicit DigitalSensor(byte pin) : _pin(pin) { pinMode(pin, INPUT); }
public: virtual ~DigitalSensor() = 0;

public: virtual VID ID() const final override { return _id; }
public: virtual byte pin() const final override { return _pin; }
public: virtual Type ctype() const override { return (Type)(ISensor::ctype() | Type::DIGITAL); }
public: virtual const __FlashStringHelper* description() const override { return F("digital sensor"); }

public: virtual DigitalState read() const { ... }
};

class AnalogSensor : public ISensor
{
public: AnalogSensor(const AnalogSensor&) = delete;
public: AnalogSensor(AnalogSensor&&) = delete;
public: AnalogSensor& operator=(const AnalogSensor&) = delete;
public: AnalogSensor& operator=(AnalogSensor&&) = delete;

private: static inline VID newcid() { static unsigned long id = 0; return (VID)++id; }
private: VID _id = newcid();

protected: byte _pin = 0;

public: explicit AnalogSensor(byte pin) : _pin(pin) { pinMode(pin, INPUT); }
public: virtual ~AnalogSensor() = 0;

public: virtual VID ID() const final override { return _id; }
public: virtual byte pin() const final override { return _pin; }
public: virtual Type ctype() const override { return (Type)(ISensor::ctype() | Type::ANALOG); }
public: virtual const __FlashStringHelper* description() const override { return F("analog sensor"); }

public: virtual AnalogValue read() const { ... }
public: virtual unit s_unit() const { unit u; u.to_default(); return u; }
};
```

Слика 40 – апстракција сензора

```

class IElement : public IComponent
{
public: virtual ~IElement() = 0;
public: virtual byte pin() const = 0;
public: virtual Type ctype() const override { return Type::ELEMENT; };
public: virtual const __FlashStringHelper* description() const override { return F("generic element"); };
};

class DigitalElement : public IElement
{
public: DigitalElement(const DigitalElement&) = delete;
public: DigitalElement(DigitalElement&&) = delete;
public: DigitalElement& operator=(const DigitalElement&) = delete;
public: DigitalElement& operator=(DigitalElement&&) = delete;

private: static inline VID newcid() { static unsigned long id = 0; return (VID)++id; }
private: VID _id = newcid();

protected: byte _pin = 0;
protected: mutable DigitalState _current = LOW;

public: explicit DigitalElement(byte pin) : _pin(pin) { pinMode(_pin, OUTPUT); digitalWrite(_pin, _current); }
public: virtual ~DigitalElement() = 0;

public: virtual VID ID() const final override { return _id; }
public: virtual byte pin() const final override { return _pin; }
public: virtual Type ctype() const override { return (Type)(IElement::ctype() | Type::DIGITAL); }
public: virtual const __FlashStringHelper* description() const override { return F("digital element"); }

public: virtual void write(DigitalState state) const { ... }
public: virtual DigitalState current_state() const { return _current; }
public: virtual void toggle() { write(!_current); }
};

class AnalogElement : public IElement
{
public: AnalogElement(const AnalogElement&) = delete;
public: AnalogElement(AnalogElement&&) = delete;
public: AnalogElement& operator=(const AnalogElement&) = delete;
public: AnalogElement& operator=(AnalogElement&&) = delete;

private: static inline VID newcid() { static unsigned long id = 0; return (VID)++id; }
private: VID _id = newcid();

protected: byte _pin = 0;
protected: mutable AnalogValue _current = 0;

public: explicit AnalogElement(byte pin) : _pin(pin) { pinMode(_pin, OUTPUT); analogWrite(_pin, _current); }
public: virtual ~AnalogElement() = 0;

public: virtual VID ID() const final override { return _id; }
public: virtual byte pin() const final override { return _pin; }
public: virtual Type ctype() const override { return (Type)(IElement::ctype() | Type::ANALOG); }
public: virtual const __FlashStringHelper* description() const override { return F("analog element"); }

public: virtual void write(AnalogValue value) const { ... }
public: virtual AnalogValue current_value() const { return _current; }
public: virtual unit e_unit() const { unit u; u.to_default(); return u; }
};

```

Слика 41 – апстракција елемента

Намена ових апстракција јесте да се, поред уздизања самог интерфејса, олакша писање нових компоненти јер основне карактеристике сваке компоненте могу да се на овакав начин издвоје. На слици 42 дат је пример имплементације тастера као компоненте.

```

#include "abstraction/sensor.h"

class BTN : public DigitalSensor
{
public: explicit BTN(byte pin) : DigitalSensor(pin) { }
public: virtual const __FlashStringHelper* description() const override { return F("button"); }
};

```

Слика 42 – имплементација тастера

С обзиром да тастер спада у улазну компоненту, она је дефинитивно сензор, и то дигитални. Довољно је наследити од исте класе, а затим редефинисати методе да се само детаљније сама компонента опише. У овом случају, довољно је само редефинисати методу која враћа опис компоненте. Слично тако, на слици 43, може се видети имплементација температурног сензора *LM35Dz*.

```
#include "abstraction/sensor.h"
#include "../common/units.h"
#include <system.h>
#include <thread.h>

class LM35 : public AnalogSensor
{
protected:
    AnalogValue temperature = 0;
    Time lts = 0;
    AnalogValue measure()
    {
        if ((System::millis() - lts) >= 60000 || (lts == 0))
        {
            temperature = 0;

            for (byte i = 0; i < 30; ++i)
            {
                temperature += AnalogSensor::read();
                Thread::delay(1);
            }

            temperature /= 30.0;
            lts = System::millis();
        }

        return temperature;
    }

public:
    explicit LM35(byte pin) : AnalogSensor(pin) { }
    virtual const __FlashStringHelper* description() const override { return F("LM35"); }

    virtual AnalogValue read() const override { return const_cast<LM35*>(this)->measure() / unitconstants::celsius(); }
    virtual unit s_unit() const override { char lbl[] { '*', 'C', 0 }; unit u(lbl); return u; }
};
```

Слика 43 – имплементација температурног сензора *LM35Dz*

У питању је аналогни сензор. Међутим, у овом случају било је неопходно редефинисати неколико метода више него у претходном примеру. Дакле, за сваку компоненту (важи и за постојеће, али фокус је на нове које би се у систем додавале накнадно), уз минималан напор потребно је са свега неколико метода описати како сама компонента функционише (дакле, написати раније поменути „драјвер“ за њу), а потом, класа се може користити на даље на произвољно много места и начина.

У фолдеру *server*, налазе се имплементације прихватања и опслуживања клијентских захтева, као и подизање самог сервера. У оквиру овог фолдера налазе се подфолдери *events* и *storage*. Код *events* подфолдера, дешава се парсирање и обрада догађаја на серверу, односно, на пример, парсирање и креирање новог догађаја који је захтеван од стране клијента (или на пример, исто тако, брисање постојећих догађаја). Већ је помињано како се догађаји чувају на складишту сервера како би перзистирали и након искључивања уређаја, па се стога, *storage* подфолдер помиње у овом контексту. Наиме, ту се налази рад са *SD* меморијском картицом, а користи се само уз догађаје.

Најважнији делови сервера почињу кроз фајлове *ethernet.h* и *request.h*. Кроз *ethernet.h* налазе се апстракције *HTTPServer* и *HTTPClient* које користе апстракције библиотеке *Ethernet.h* од *Arduino*-а, а представљају сумирани интерфејс за лакши рад са конекцијама и крајњим „тачкама“ у комуникацији између клијента и сервера. Важно је напоменути да ове апстракције, иако у имену имају *http*, знају само да комуницирају до

тог нивоа у слојевима који су претходно били обрађени, не укључујући и њега самог. Међутим, у истом фајлу, налази се апстракција *HTTPRequestParser*, која је уско повезана са претходне две, а њена намена је да представља интерфејс, тачније надкласу за конкретне парсере *http* захтева, који управо парсирају захтеве по овом протоколу, укључујући и тај слој. Имплементација овакве класе налази се у *request.h* фајлу, названа *RequestHandler*, заједно са апстракцијом *RequestBodyParser*, која, као што је до сада поменуто, по протоколу вишег нивоа, узима из тела захтева послат *XML*, а потом њега парсира и одлучује која **акција** треба да се изврши (у зависности од самог садржаја овог *XML*-а).

У овом тренутку, треба напоменути неколико ствари око захтева који се упућују ка серверу, првенствено говорећи о заглављима *http* захтева. Већ је до сада поменуто да сервер не разазнаје странице, иако је то најосновнији концепт *http* захтева. Сваки захтев мора да приступа почетној, *index* страници, и то искључиво не наводећи њено име, већ празну путању „/“. У супротном, сервер ће послати неку врсту поруке која индикује лош захтев (енгл. *bad request*). Такође, то мора бити искључиво *POST* метода захтева (с обзиром да „линк адреса“ не постоји јер сам сервер не разазнаје странице, *GET* захтеви, који информације управо шаљу кроз линк адресу, немају смисла, а такође се заправо кроз тело захтева шаљу информације (у *XML*-у), што обавезује коришћење *POST* методе, па се она из ових разлога користи.

Што се коришћених заглавља тиче, сервер само посматра одређене, док остале у потпуности игнорише јер такви у овом контексту немају смисла ни примену. Конкретно, постоје 3 неопходна и 2 опциона заглавља. Неопходни су *Content-Type*, *Host* и *Content-Length*. За *Content-Type*, потребно је искључиво навести *application/xml; charset=utf-8*. С обзиром да се користи *XML* у телу, и да је тако протокол дефинисан, ово мора бити вредност датог заглавља. Параметар *host*, иако неопходан, његова вредност се не користи нигде директно. Међутим, остављен је као неопходан због постојања шансе да ће се у будућности при побољшавању и дањем развијању система користити. Параметар *Content-Length* је битан како би сервер знао дужину самог тела, односно, колико карактера треба да чека да употпуни захтев и започне парсирање самог тела (тј. *XML*-а). Преостала 2, опциона заглавља, јесу *Expect* са вредношћу *100-continue*, као и *Connection*, који може имати вредности *Close* или *Keep-Alive*. Стога, изглед уобичајеног заглавља захтева може изгледати на следећи начин:

```
POST / HTTP/1.1
Host: localhost
Content-Type: application/xml; charset=utf-8
Content-Length: 54
Expect: 100-continue
Connection: Close
```

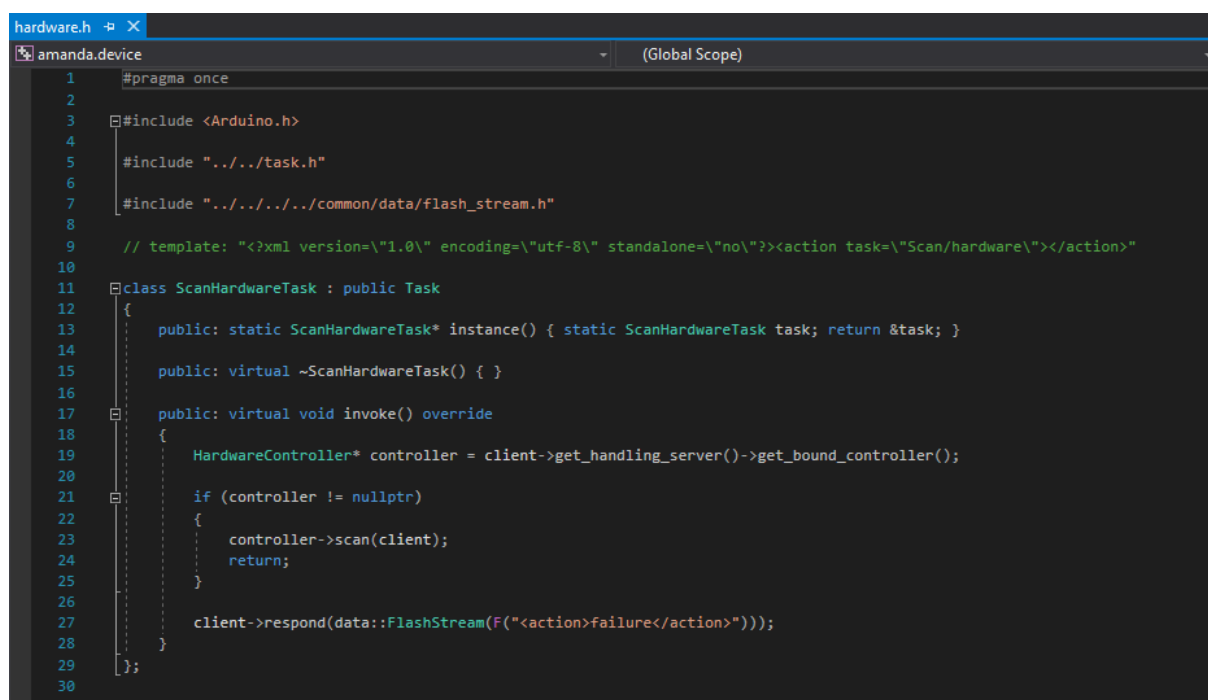
Треба напоменути да су вредности *Host* и *Content-Length* овде насумично изабрани и не представљају фиксирани податаке.

Поменуто је да апстракција *RequestBodyParser* узима тело захтева, што је заправо *XML*, а након парсирања одлучује која **акција** треба да се изврши. Концепт акција управо представљају радње које се дешавају као последица коректно издатог захтева од стране клијента. Постоје две групе акција, а то су процеси (енгл. *process*) и задаци (енгл. *task*). Суштинска разлика између ова два концепта не постоји, међутим, разлог за њихово

увођење и раздвајање јесте у двострукој потреби за једноставнијим, лакшим акцијама и сложенијим, тежим, а при томе, процеси често могу бити асоцијација нечега што „траје“ (попут креирања догађаја, јер сам догађај представља нешто што није нужно само тренутна ствар), док су задаци често краткотрајни, односно тренутни (нпр. издавање команде да се лампа укључи).

Бирање акција, као и радње које они обављају налазе се у подфолдеру *actions*. *API* за акције је проширив, па самим тим, могуће је правити нове акције. На неки начин, ове акције се могу повезати са *MVC* узорком који је чест у *web* програмирању, где акције практично представљају методе контролера које се позивају у зависности од тога шта сам клијентски захтев тражи да се обради. Слично тако је и у овој ситуацији, *XML* тела захтева индикује која акција ће се позвати. Ту се такође наводи да ли се ради о процесу или задатку. Разлика ова два је једноставна: задаци, када се затраже, могу само да имају још додатно произвољан низ аргумената који параметризују сам задатак, док процеси могу имати произвољну подструктуру *XML*-а у самом захтеву, који се комплетно прослеђује *XML* подпарсеру који је везан за сам процес.

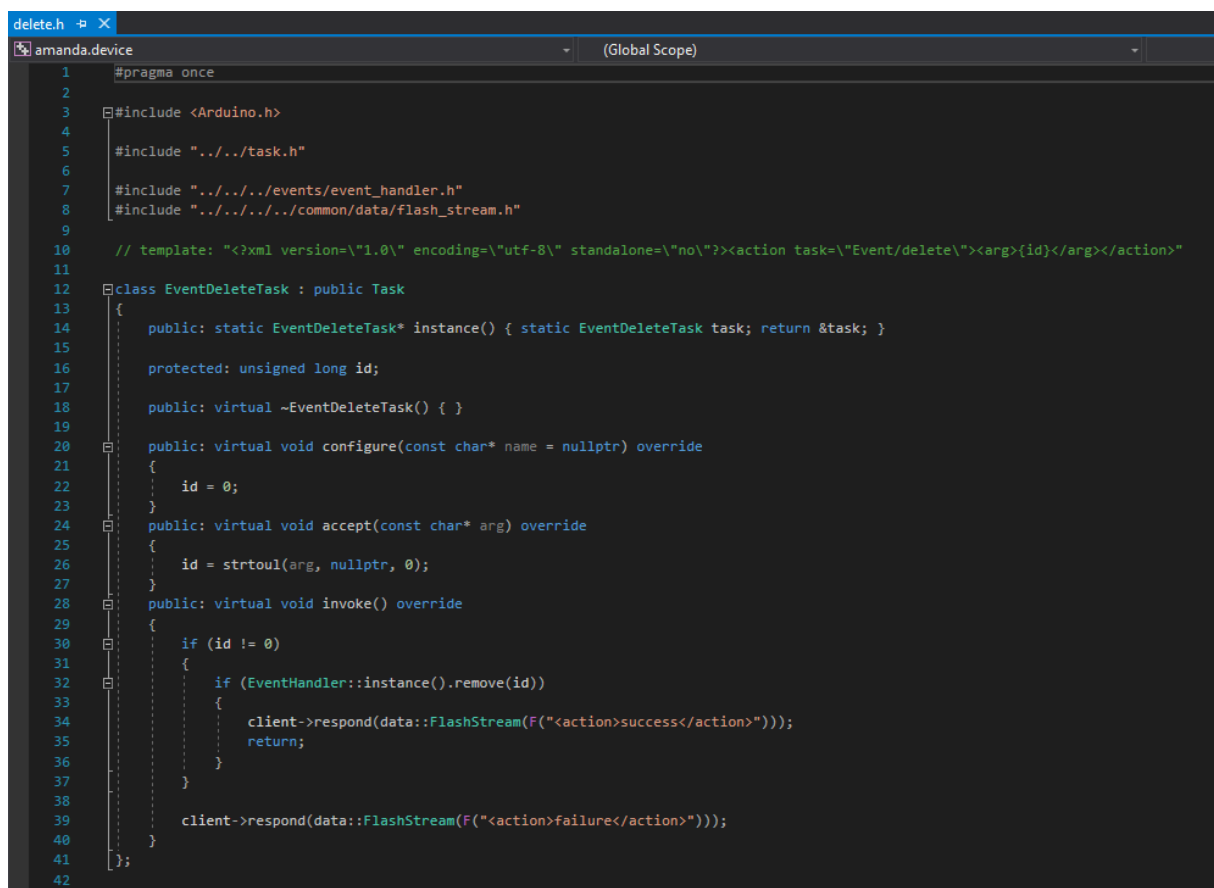
На слици 44 дат је пример задатка који се тиче скенирања хардвера. На слици се може видети *XML* у коментару, који представља тело захтева који практично позива овај задатак. Сваки задатак се позива навођењем атрибута *task* у оквиру *action* тага који је увек корен за сваки захтев упућен серверу. Вредност овог атрибута је заправо име самог задатка, а имена су дефинисана у *task.cpp* фајлу. Сваки таск може да редефинише неколико метода, од којих је главна *invoke()*, а ова метода садржи конкретан код који дефинише шта сам задатак треба да уради. Овде такође треба обавити и одговарање клијенту, што се може вршити кроз поље које се налази у надкласи, а то је *client*. У овом конкретном примеру, након што се дохвата *HardwareController*, његова метода *scan()* прихвата параметар *OutputStream*, ток на који треба да испише сам резултат скенирања, што директно може бити клијент јер сама класа *HTTPClient* јесте *OutputStream*.



```
hardware.h  X
amanda.device  (Global Scope)
1  #pragma once
2
3  #include <Arduino.h>
4
5  #include "../task.h"
6
7  #include "../../common/data/flash_stream.h"
8
9  // template: "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='Scan/hardware'></action>"
10
11  class ScanHardwareTask : public Task
12  {
13  public: static ScanHardwareTask* instance() { static ScanHardwareTask task; return &task; }
14
15  public: virtual ~ScanHardwareTask() { }
16
17  public: virtual void invoke() override
18  {
19      HardwareController* controller = client->get_handling_server()->get_bound_controller();
20
21      if (controller != nullptr)
22      {
23          controller->scan(client);
24          return;
25      }
26
27      client->respond(data::FlashStream(F("<action>failure</action>")));
28  }
29
30  };
```

Слика 44 – *task* за скенирање хардвера

Још један пример задатка може бити брисање догађаја, који се може видети на слици 45. Поново се може приметити коментар који представља тело захтева који позива овај задатак. Може се видети да у овом случају постоји таг *arg* који представља аргумент унутар кореног тага *action*, а семантика која стоји иза њега јесте да тај, први и једини обавезни аргумент представља *id* самог догађаја који се жели брисати. Из овог разлога, редефинисане су још две методе, *configure()* и *accept()*. Метода *configure()* служи да конфигурише (иницијализује) сам задатак јер, као што се може приметити, сви задаци треба да буду *singleton* инстанце, јер се реупотребљавају. Притом, прослеђује се параметар *name*, који представља опционо прослеђено од стране клијента, уколико сам жели да се тај задатак на одређен начин именује. Овај концепт именовања се у суштини користи само при креирању догађаја (што представља процес), а управо то име се користи као име да се сам догађај тако и назове. Друга метода, *accept()*, прима стринг аргумента који прослеђен кроз *XML*, а позива се за сваки овакав прослеђени аргумент. Ту је дакле могуће парсирати аргументе, а у овом случају, потребно је само стринг конвертовати у број.

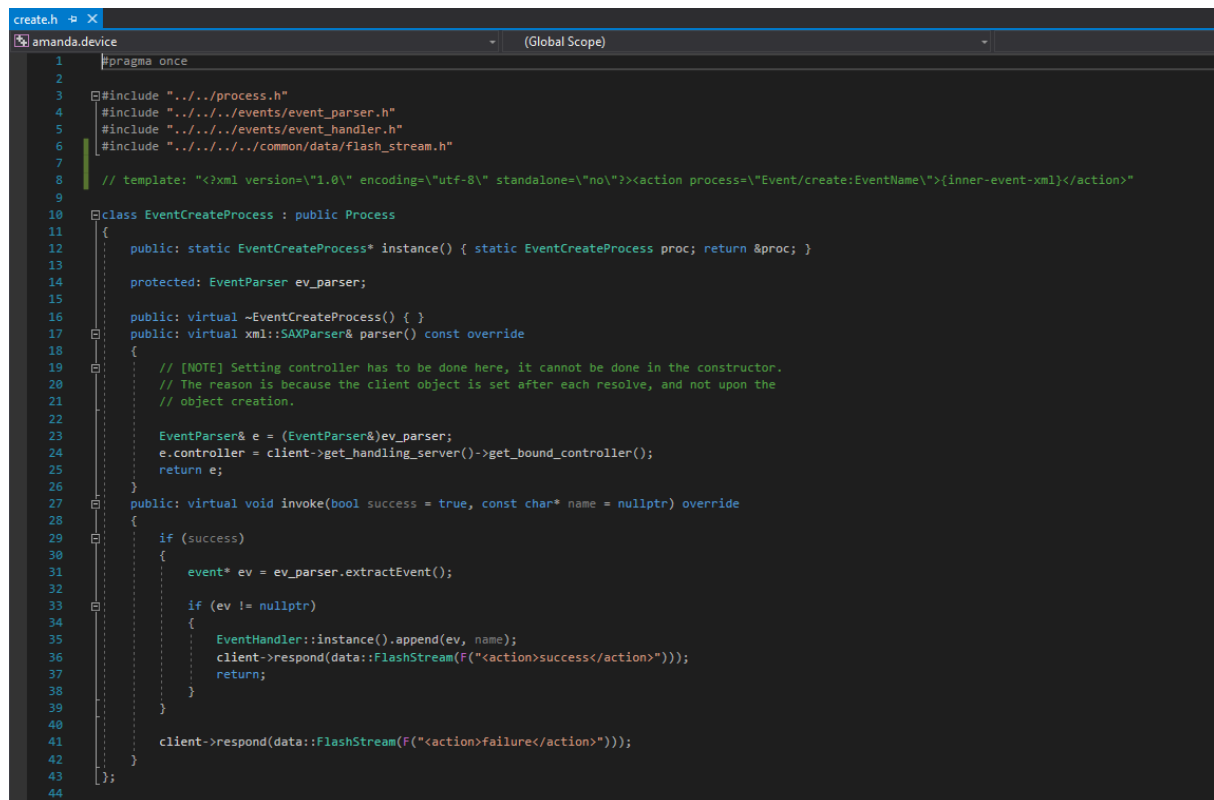


```
delete.h X
amanda.device (Global Scope)
1 #pragma once
2
3 #include <Arduino.h>
4
5 #include "../task.h"
6
7 #include "../../../events/event_handler.h"
8 #include "../../../common/data/flash_stream.h"
9
10 // template: "<?xml version='1.0' encoding='utf-8' standalone='no'?><action task='Event/delete'><arg>{id}</arg></action>"
11
12 class EventDeleteTask : public Task
13 {
14 public: static EventDeleteTask* instance() { static EventDeleteTask task; return &task; }
15
16 protected: unsigned long id;
17
18 public: virtual ~EventDeleteTask() { }
19
20 public: virtual void configure(const char* name = nullptr) override
21 {
22     id = 0;
23 }
24 public: virtual void accept(const char* arg) override
25 {
26     id = strtoul(arg, nullptr, 0);
27 }
28 public: virtual void invoke() override
29 {
30     if (id != 0)
31     {
32         if (EventHandler::instance().remove(id))
33         {
34             client->respond(data::FlashStream(F("<action>success</action>")));
35             return;
36         }
37     }
38     client->respond(data::FlashStream(F("<action>failure</action>")));
39 }
40 };
41
42
```

Слика 45 – *task* за брисање догађаја

За разлику од задатка, на месту где долазе аргументи, код процеса, ту се може налазити било каква произвољна структура *XML*-а. На слици 46 дат је пример процеса који креира нов догађај.





```

1  #pragma once
2
3  #include "../process.h"
4  #include "../events/event_parser.h"
5  #include "../events/event_handler.h"
6  #include "../common/data/flash_stream.h"
7
8  // template: "<?xml version='1.0' encoding='utf-8' standalone='no'><action process='Event/create:EventName'>{inner-event-xml}</action>"
9
10 class EventCreateProcess : public Process
11 {
12 public: static EventCreateProcess* instance() { static EventCreateProcess proc; return &proc; }
13
14 protected: EventParser ev_parser;
15
16 public: virtual ~EventCreateProcess() {}
17 public: virtual xml::SAXParser& parser() const override
18 {
19     // [NOTE] Setting controller has to be done here, it cannot be done in the constructor.
20     // The reason is because the client object is set after each resolve, and not upon the
21     // object creation.
22
23     EventParser& e = (EventParser&)ev_parser;
24     e.controller = client->get_handling_server()->get_bound_controller();
25     return e;
26 }
27 public: virtual void invoke(bool success = true, const char* name = nullptr) override
28 {
29     if (success)
30     {
31         event* ev = ev_parser.extractEvent();
32
33         if (ev != nullptr)
34         {
35             EventHandler::instance().append(ev, name);
36             client->respond(data::FlashStream(F("<action>success</action>")));
37             return;
38         }
39     }
40
41     client->respond(data::FlashStream(F("<action>failure</action>")));
42 }
43 };
44

```

Слика 46 – *process* за креирање догађаја

Сада се може приметити да уместо атрибута *task*, користи се атрибут *process*. Након што се наведе име процеса за вредност овог атрибута (*Event/create*), могуће је ставити двотачку, а затим навести произвољан стринг. Управо ово је горе поменуто име које се појављивало у оквиру *configure()* методе код задатака (и код задатака је ово могуће дакле наводити). Код процеса, ово је други параметар методе *invoke()*. За овај конкретан процес, ово је неопходан параметар, јер сваки догађај мора имати своје име. Уколико се пак не наведе, биће дато аутоматско име „*@unnamed*“. За процесе је битно редефинисати две методе. Једна је већ поменута метода *invoke()*, која као код задатака, позива се када се сам процес захтева. Поред поменутог параметра за називање процеса, прослеђује се још једна *bool* вредност која индикује да ли је подструктура *XML*-а валидно парсирана од стране *XML* подпарсера који је везан за сам процес. Друга метода, *parser()*, је управо метода која треба да врати инстанцу оваквог подпарсера који ће се користити за парсирање унутрашње стурктуре захтева.

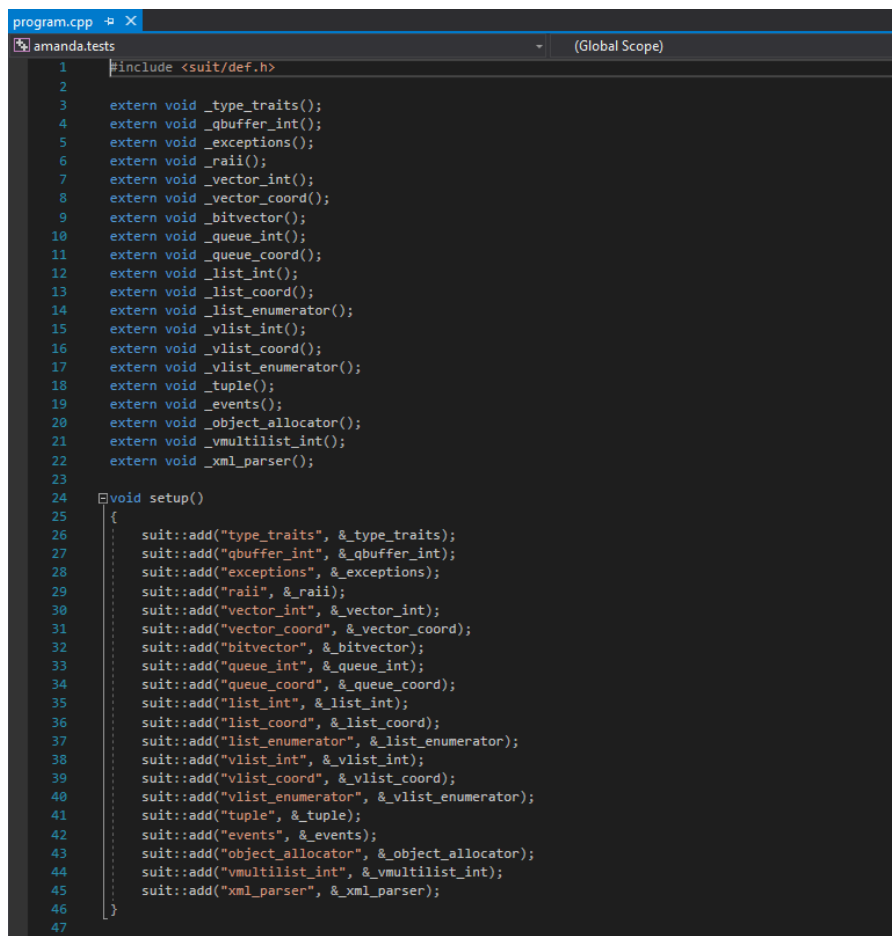
Битно је напоменути да за нове задатке и процесе није их довољно само дефинисати на овакав начин, већ се њихове инстанце морају и вратити у методатам *resolve()* које се налазе у *task.cpp* и *process.cpp*.



#### 5.3.1.4. *SUIT & TESTS*

Ова два пројекта имају улогу да јединично тестирају делове претходно описаног система. Битно је рећи да се ова два пројекта не покрећу на *Arduino*-у, већ директно на рачунару, тј. представљају у суштини стандардне *C++* пројекте. Прецизније, *Suit* пројекат је поново дељени пројекат који садржи мало језгро *Unit-Testing* окружења које се може користити у стандардним *C++* пројектима. Пројекат *Tests* је основни пројекат за претходни, и садржи конкретне тестове који могу да се покрећу. Стога, *Tests* пројекат укључује *Suit* пројекат. Међутим, поред њега, *Tests* пројекат укључује и *Concepts* пројекат. Разлог овоме је управо потреба да се тај пројекат тестира, јер сам *Tests* уз *Suit*, иако имао могућност за тестирање, не би имао конкретан други код који треба да тестира. Наравно, захтев и јесте да се сам *Concepts* тестира, а ово је могуће јер као што је речено, овај пројекат није писан тако да је везан само за *Arduino* платформу, већ да буде што је могуће више *cross-platform*, па се самим тим и може покретати као стандардни *C++* пројекат, или барем у оквиру таквог, јер је дељен. Пројекти *Core* и *Device* се не могу овде тестирати, али у оквиру *Device* пројекта, већ је поменуто да он сам садржи своје тестове који управо то и раде.

Када се у оквиру празног *C++* пројекта укључи *Suit* пројекат, потребно је уместо *main()* методе, коју више не треба дефинисати, заправо дефинисати *setup()* методу. Ова метода треба да у *testing-suit* убаци функције које ће се позивати, где свака појединачна функција представља један тест. На слици 47 може се видети метода *setup()*.



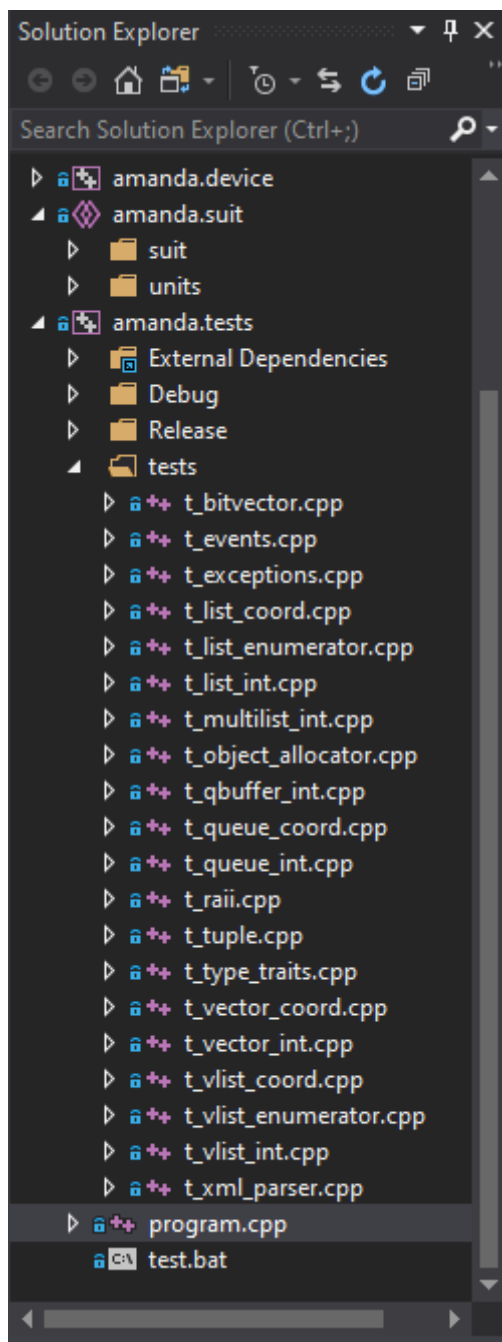
```

program.cpp
amanda.tests
(Global Scope)
1  #include <suit/def.h>
2
3  extern void _type_traits();
4  extern void _qbuffer_int();
5  extern void _exceptions();
6  extern void _raii();
7  extern void _vector_int();
8  extern void _vector_coord();
9  extern void _bitvector();
10 extern void _queue_int();
11 extern void _queue_coord();
12 extern void _list_int();
13 extern void _list_coord();
14 extern void _list_enumerator();
15 extern void _vlist_int();
16 extern void _vlist_coord();
17 extern void _vlist_enumerator();
18 extern void _tuple();
19 extern void _events();
20 extern void _object_allocator();
21 extern void _vmultilist_int();
22 extern void _xml_parser();
23
24 void setup()
25 {
26     suit::add("type_traits", &_type_traits);
27     suit::add("qbuffer_int", &_qbuffer_int);
28     suit::add("exceptions", &_exceptions);
29     suit::add("raii", &_raii);
30     suit::add("vector_int", &_vector_int);
31     suit::add("vector_coord", &_vector_coord);
32     suit::add("bitvector", &_bitvector);
33     suit::add("queue_int", &_queue_int);
34     suit::add("queue_coord", &_queue_coord);
35     suit::add("list_int", &_list_int);
36     suit::add("list_coord", &_list_coord);
37     suit::add("list_enumerator", &_list_enumerator);
38     suit::add("vlist_int", &_vlist_int);
39     suit::add("vlist_coord", &_vlist_coord);
40     suit::add("vlist_enumerator", &_vlist_enumerator);
41     suit::add("tuple", &_tuple);
42     suit::add("events", &_events);
43     suit::add("object_allocator", &_object_allocator);
44     suit::add("vmultilist_int", &_vmultilist_int);
45     suit::add("xml_parser", &_xml_parser);
46 }
47

```

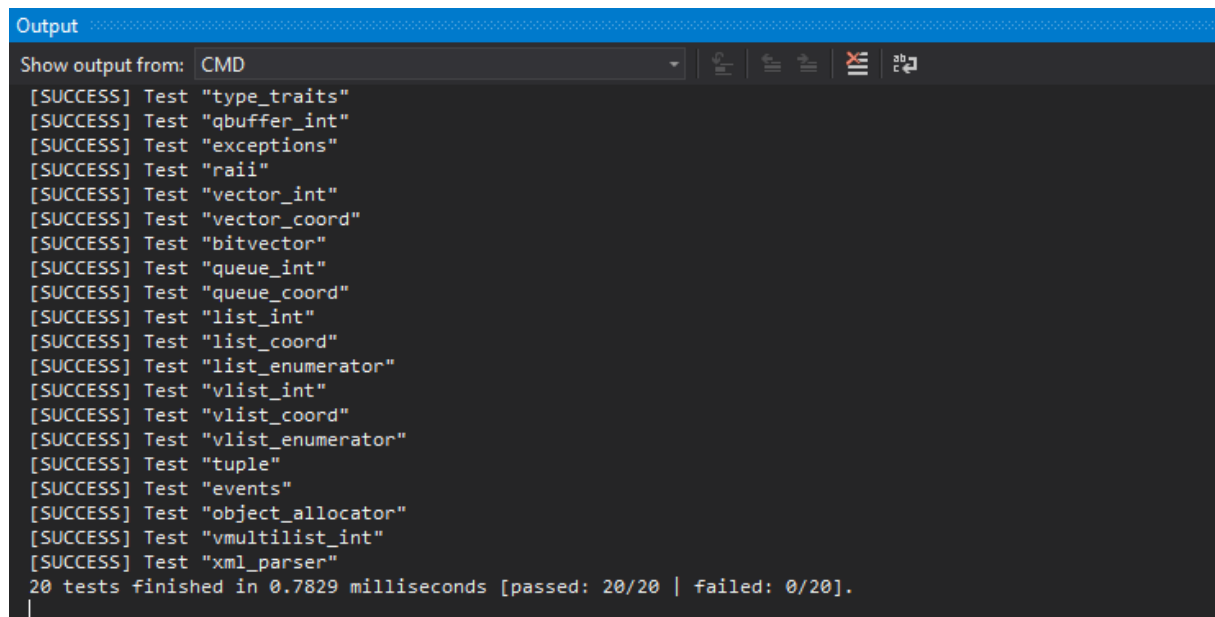
Слика 47 – дефинисање постојећих тестова

На слици 48 приказана је структура ова два пројекта.



Слика 48 – структура *Suit* и *Tests* пројекта

Сви тестови су могли бити дефинисани у једном фајлу, нпр. у *program.cpp* поред саме *setup()* функције. Међутим, ради лакшег сналажења, сваки тест је написан у засебан фајл, а сви се налазе у оквиру *tests* фолдера. За покретање тестова, потребно је изградити (енгл. *build*) пројекат, и то у *release* моду, а затим покренути *test.bat* (десним кликом, а затим одабрати опцију *Run*). На слици 49 представљен је исход покретања тестова који се исписује у конзоли *Visual Studio* окружења.



```
Output
Show output from: CMD
[SUCCESS] Test "type_traits"
[SUCCESS] Test "qbuffer_int"
[SUCCESS] Test "exceptions"
[SUCCESS] Test "raii"
[SUCCESS] Test "vector_int"
[SUCCESS] Test "vector_coord"
[SUCCESS] Test "bitvector"
[SUCCESS] Test "queue_int"
[SUCCESS] Test "queue_coord"
[SUCCESS] Test "list_int"
[SUCCESS] Test "list_coord"
[SUCCESS] Test "list_enumerator"
[SUCCESS] Test "vlist_int"
[SUCCESS] Test "vlist_coord"
[SUCCESS] Test "vlist_enumerator"
[SUCCESS] Test "tuple"
[SUCCESS] Test "events"
[SUCCESS] Test "object_allocator"
[SUCCESS] Test "vmultilist_int"
[SUCCESS] Test "xml_parser"
20 tests finished in 0.7829 milliseconds [passed: 20/20 | failed: 0/20].
```

Слика 49 – исход тестова

Уколико неки тест не прође, на том месту ће стајати и порука која индикује разлог зашто је дати тест пао. Наравно, у самом тесту, то је морало бити назначено од стране програмера. Тестови су свакако базирани на принципу *assert* провера које морају бити тачне, уколико барем једна није, тест се прекида и грешка се пријављује. Треба такође напоменути да ово није статички *assert* као раније, већ динамички, који се проверава у току рада програма (енгл. *Runtime*).

### 5.3.2. ДЕТАЉИ ПРОТОКОЛА У КОМУНИКАЦИЈИ

Већ су виђени неки од принципа у протоколу који се користе за комуникацију између уређаја и апликације. Конкретно, то је био део захтева кога клијентска апликација шаље серверу у виду акција које жели да се изврши. Постоје два типа акција, а то су задаци и процеси. Формати ова два типа захтева, респективно, изгледају на следећи начин:

```
* <action task="{name}"><arg>{value}</arg>...</action>
* <action process="{name}">{inner-xml}</action>
```

Постоје већ дефинисани задаци и процеси, и за сваког се зна унапред шта од аргумената треба слати, њихов број, тј. каквог формата треба да буде унутрашња подструктура процеса и слично. Наравно, могуће је у систему развијати и нове акције.

У протоколу могуће је размењивати комплетне информације о догађајима. У наставку је дат пример једног догађаја. Треба напоменути да се овде ради само конкретној структури догађаја, без његовог имена. Овај податак се посебно шаље, у зависности од тога да ли је у питању креирање догађаја (у том случају податак се шаље кроз до сада описано именовање задатака и процеса), а уколико је у питању одговор сервера на захтев скенирања свих постојећих догађаја (тренутак када клијент захтева да му сервер одговори које све догађаје има записане код себе), око датог *XML*-а се додаје таг *event\_handle* са атрибутом *name*. Такође, пример у наставку директно може бити искоришћен као подструктура процеса за креирање догађаја.

```
// <event repeat="0">
//   <requirements>
//     <pack>
//       <condition vid="4" ctype="AS"> // LDR
//         <lss>100.0</lss>
//       </condition>
//       <condition vid="3" ctype="DS"> // PIR
//         <equ>1</equ>
//       </condition>
//     </pack>
//   </requirements>
//   <actions>
//     <raise>
//       <write vid="3" ctype="DE"> // LED3
//         <state>1</state>
//       </write>
//     </raise>
//     <expire>
//       <write vid="3" ctype="DE"> // LED3
//         <state>0</state>
//       </write>
//     </expire>
//   </actions>
// </event>
```

У наставку је дата листа постојећих задатака и процеса, као и њихови формати и намене. Заглавље ће се изоставити из сваке инстанце *XML* документа, а оно што је веома важно за сам протокол јесте да је заглавље **фиксирано** и увек мора да изгледа на следећи начин:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
```

- Задатак *Echo/hello*
  - Формат захтева: `<action task="Echo/hello"></action>`
  - Формат одговора: `<message>Hello World.</message>`
  - Намена: успостављање конекције, односно, на дати захтев, уређај увек на исти начин одговара, па уколико се дата порука кроз одговор и прими, клијентска апликација зна да успешно може на даље да комуницира са уређајем, па самим тим, може се рећи да је конекција успешна
- Задатак *Echo/reply*
  - Формат захтева: `<action task="Echo/reply"><arg>{message}</arg></action>`
  - Формат одговора: `<message>{message}</message>`
  - Намена: иста као задатак *echo/hello*, само што уместо фиксиране поруке коју враћа, овог пута враћа се порука која је прослеђена кроз први и једини аргумент; представља типичан пример *echo* или *ping* принципа, такође могуће за коришћење у оквиру конекције са уређајем и/или провере конекције

- Задатак *IO/basic*
  - Формат захтева:

```
<action task="IO/basic">
  <arg>{vid}:{ctype}</arg>
  <arg>{operation}</arg>
  <arg>{value unit}</arg>
</action>
```
  - Формат одговора: `<action>failure</action>` или `<action>success</action>`, осим у случају ако је тражен *READ*, онда `<state>{state}</state>` за дигиталне компоненте или `<value unit="{unit}">{value}</value>` за аналогне
  - Намена: рад са основним *IO* операцијама; нпр на компоненту одређену паром *vid:ctype* одрадити *operation* (*READ/WRITE*), а у случају *WRITE*, трећи аргумент дефинише коју вредност (за дигиталне компоненте користити 0/1, без *unit*-а).
- Задатак *Scan/hardware*
  - Формат захтева: `<action task="Scan/hardware"></action>`
  - Формат одговора: `<scan>{component-xml}...</scan>`
  - Намена: скенирање хардвера; треба напоменути да се под `{component-xml}` сматра *XML* скен једне компоненте, где корени таг за децу има листу оваквих *XML* подструктура, а структура једне може да изгледа на следећи начин:

```
<component vid="4" ctype="DE" description="Lamp" commands="|blink|stop|">
  <state>0</state>
</component>
```

Такође треба напоменути да свугде у систему, што се вредности тиче, може се користити таг *state* за индиковање дигиталног стања (вредности искључиво 0 и 1), или таг *value* за аналогну вредност, који такође поседује *unit* атрибут (ово укључује и префиксе, нпр. \*C, lux, ms итд.)
- Задатак *Scan/events*
  - Формат захтева: `<action task="Scan/events"></action>`
  - Формат одговора: `<scan>{event-xml}...</scan>`
  - Намена: скенирање догађаја; као и у претходном примеру, под `{event-xml}` се сматра један догађај, у облику који је изнад описан, али тако да је још додатно окружен тагом *event\_handle*, који за атрибут *id* садржи идентификатор догађаја, а за атрибут *name* садржи име догађаја.

- Задатак *Cmd/exec*
  - Формат захтева:

```
<action task="Cmd/exec">
  <arg>{vid}:{ctype}</arg>
  <arg>{cmd_name}</arg>
  <arg>{arg}</arg>...
</action>
```
  - Формат одговора: `<action>success</action>` или `<action>failure</action>` уколико сама команда ништа не одговара, или уколико одговара, сама команда дефинише одговор
  - Намена: извршавање команди над одређеном компонентом; компонента је дефинисана паром *vid:ctype* из првог аргумента, а име команде из другог; остали аргументи, уколико постоје, директно прате аргументе саме команде и њој се прослеђују
- Процес *Event/create*
  - Формат захтева:

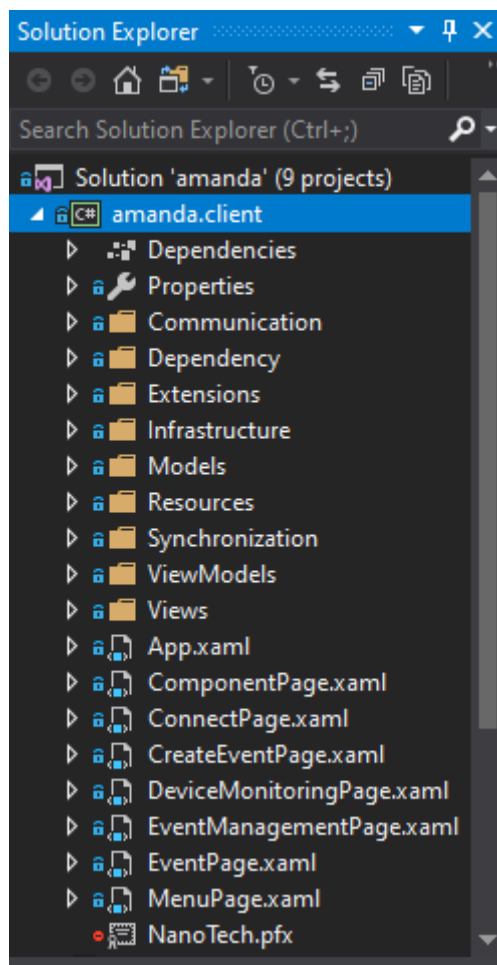
```
<action process="Event/create:{event-name}">{inner-event-xml}</action>
```
  - Формат одговора: `<action>success</action>` или `<action>failure</action>`
  - Намена: креирање догађаја; већ је објашњено да подструктура кореног тага треба да буде структура догађаја представљена изнад; такође, треба навести и име догађаја, иначе ће добити подразумевано
- Задатак *Event/delete*
  - Формат захтева: `<action task="Event/delete"><arg>{id}</arg></action>`
  - Формат одговора: `<action>success</action>` или `<action>failure</action>`
  - Намена: брисање догађаја; *id* представља аргумент који индикује који догађај се брише

На овај начин, постигнут је ефекат да клијентска апликација може да буде израђена у било којој технологији, докле год има отворен *API* за рад са *http* захтевима, уређај независно од ње прима захтев по протоколу и на тај начин обавља неопходне активности које му сама корисничка апликација задаје.

### 5.3.3. СОФТВЕР КЛИЈЕНТСКЕ АПЛИКАЦИЈЕ

Као што је већ речено, клијентска апликација је рађена у *Microsoft .NET Xamarin.Forms* технологији, која омогућава израду *cross-platform* апликација, и то кроз 3 главне платформе (*UWP*, *Android*, *iOS*). Отуда постоје и 4 пројекта, један који је дељен (не на начин који су делили *C++* и *Arduino* пројекти), а остали представљају директне пројекте за сваку платформу који користе дељени. У овој реализацији, 3 платформски зависна пројекта након креирања нису мењана, целокупна апликација је успешно развијена у потпуности у дељеном пројекту, па ће само он бити обрађиван, док остале 3 служе искључиво само да изграде читав пројекат на крају.

На слици 50 може се видети структура овог пројекта. Као што је већ речено, за израду страница апликације, односно за *UI* део, користи се *XAML*, али свакако да у позадини постоји и *.cs* фајл који садржи интерну логику странице. Стога, сваки *XAML* ентитет се заправо одатле састоји из 2 фајла.



Слика 50 – структура пројекта клијентске апликације

*App.xaml* је пар фајлова *.xaml/.cs* који, иако тако изгледа, се не односи ни на једну страницу. То је почетни фајл у ком се креира сама инстанца апликације (*.cs*), а такође ту се налазе и неки додатни ресурси које читава апликација користи (*.xaml*).

У *Communication* фолдеру имплементирани су концепти који се тичу комуникације по протоколу. *API* који језик *C#* даје такав је да распознаје и *http* протокол у слојевима, тако да се не мора водити рачуна о заглављима и сличним стварима као на *Arduino*-у. Довољно је само припремити *XML* серијализоване стрингове и њих доставити кроз *API* при позиву, тј. слању захтева ка серверу. На слици 51 налази се статичка класа која садржи припремљене *XML* поруке по протоколу, док се на слици 52 може видети принцип слања једне поруке.

```

public static class Protocol
{
    public const double RefreshSpeed = 500;

    public const string HelloMessage = "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='Echo/hello'></action>";
    public const string HelloReply = "<?xml version='1.0' encoding='utf-8' standalone='no'><message>Hello World.</message>";
    public const string TimeMessage = "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='IO/basic'><arg>1:AS</arg><arg>read</arg></action>";
    public const string ScanHardware = "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='Scan/hardware'></action>";
    public const string ScanEvents = "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='Scan/events'></action>";

    public static string IODigitalRead(uint id)
    {
        const string xml = "<?xml version='1.0' encoding='utf-8' standalone='no'><action task='IO/basic'><arg>{0}:DS</arg><arg>read</arg></action>";
        return string.Format(xml, id);
    }

    public static string IODigitalWrite(uint id, bool state)...
    public static string IOAnalogRead(uint id)...
    public static string IOAnalogWrite(uint id, double value, string unit)...

    public static string Command(uint id, CType ctype, string name, params string[] args)...

    public static string CreateEvent(Event e)...
    public static string DeleteEvent(uint id)...

    public const string ActionSuccess = "<?xml version='1.0' encoding='utf-8' standalone='no'><action>success</action>";
    public const string ActionFailed = "<?xml version='1.0' encoding='utf-8' standalone='no'><action>failure</action>";
}

```

Слика 51 – класа са порукама протокола

```

public static async Task<string> Send(string xml)
{
    var client = Dependency.Resolve<HttpClient>();
    Connection con = Dependency.Resolve<Connection>();

    using (var body = new StringContent(xml, Encoding.UTF8, "application/xml"))
    {
        using (var response = await client.PostAsync("http://" + con.Address + ":" + con.Port + "/", body))
        {
            if (response.IsSuccessStatusCode)
            {
                var content = await response.Content.ReadAsStringAsync();

                if (content != null)
                {
                    return content;
                }
            }
        }
    }

    return string.Empty;
}

```

Слика 52 – принцип слања захтева

У оквиру фолдера *Dependency* налази се лака варијанта имплементације пројектног узорка *Dependency Injection*. Примене ове инстанце узорка искључиво су везане за глобалне зависности које се користе само кроз овај пројекат.

Постоје и неке услужне класе које садрже само помоћни код и/или скраћују код у неким деловима. Конкретно то су фајлови у оквиру *Extensions* и *Synchronization* фолдера. Такође, фолдер *Infrastructure* има мало комплекснији сет услужних класа од претходне две, а тиче се познанства мерних јединица, као и начин на који се оне у различитим ситуацијама приказују (овај фолдер је предвиђен да садржи комплетну софтверску инфраструктуру система која је неопходна да постоји на клијентској апликацији, међутим, на крају је само постојала потреба за креирањем система мерних јединица).

Преостала 3 фолдера, *Models*, *Views* и *ViewModels* представљају целине које заједно образују већ обрађени *MVVM pattern*. Сваки фолдер има своје кореспондне јединице у овом узорку.



## 6. ЗАКЉУЧАК

Реализовани систем који је направљен омогућава кориснику да врши комплетну аквизицију података и надгледање уређаја, као и околине у којој се налази, да га уређај обавештава уколико се одређене активности дешавају, а такође, омогућава му и да аутоматски регулише жељене параметре. Овакав систем представља једну целину која је функционална, а која се даље може надограђивати.

У систему не постоји никакав вид заштите, што је први задатак који треба урадити у надоградњи система. Такође, треба осмислити добар начин корисничке ауторизације, како не би долазило до неауторизованог приступа. Многи аспекти заштите података, као и сами алгоритми, треба да се имплементирају на уређају. Ово потенцијално може да захтева додавање других меморијских модула.

Код акција догађаја, тренутно је могуће само радити основне *IO* операције. Систем би се могао проширити тако да на овом месту подржи и извршавање команди. Ово првенствено није било могуће због превеликих меморијских захтева самих команди, па је иницијално донешена одлука да се то не подржава, али у једном тренутку, употребљен је узорак *Builder* који је довео до драстичног смањења меморијских захтева, па је сада могућа подршка истог.

Разне оптимизације се такође могу урадити на неким местима. На пример, код изузетака, могао би се наћи начин да се избегну коришћења директних стрингова који на *Arduino*-у заузимају *SRAM*. Освежавање догађаја и рад са сервером (односно клијентима) би могло да се издвоји кроз две нити, што би захтевало и међусобну синхронизацију већих делова кода. Такође, могла би се употребити *xor-doubly linked list* као меморијски ефикасна двоструко уланчана листа на одређеним местима.

Разни додатни задаци и процеси би могли да се прошире у систему. На пример, измена самих догађаја би могла да се креира као процес. Тренутно, уколико корисник жели да измени неки догађај, прво мора да га обрише, а затим да креира нов, који је измењена варијанта претходног. Наравно, ово не мора да се уради на страни уређаја. Клијентска паликација може да направи могућност измене тако што ће, након што корисник унесе измене, прво послати захтев за брисање догађаја, а затим за креирање новог, измењеног. Мана овог приступа је што се шаљу два захтева и отварају две засебне конекције, а такође, као нус ефекат, клијенту ће се *ID* самог догађаја променити. Стога, решење са новим процесом јесте ефикасније.

На серверу уређаја би могло да се подржи низ акција у једном захтеву. Уколико клијентска апликација жели више акција, тренутно мора за сваку да пошаље по један засебан захтев, што је мање перформантно по многим параметрима. Уколико се подржи низ кроз један захтев, не само да се перформансе повећавају, већ је и могућа атомичност извршавања тих акција на уређају. Наравно, уколико корисник не жели атомичност, и у тој варијанти, он може слати засебно захтеве.

Поред тога, итекако је систему неопходно још „драјвера“ за друге врсте компоненти и уређаја. Ово је посао који нема директног краја, али би било добро написати минимални сет који би покривао већину честих случаја који се догађају у пракси.

## Литература

- [1] Документација из *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V*, аутор: *Atmel Corporation*, преузето са: <https://www.alldatasheet.com/>
- [2] Документација из *8-bit Microcontroller with 256K Bytes In-System Programmable Flash*, аутор: *Atmel Corporation*, преузето са: <https://www.alldatasheet.com/>
- [3] Документација за *Arduino*, линк: <https://www.arduino.cc/>
- [4] Форум са разним питањима и решеним проблемима за *Arduino* које су имали други програмери, линк: <https://forum.arduino.cc/>
- [5] Форум са разним питањима и решеним проблемима кроз разне аспекте програмирања које су имали други програмери, линк: <https://stackoverflow.com/>
- [6] Документ *A Software Developer's Guide to HTTP Part III-Connections*, линк: <https://odetocode.com/articles/743.aspx>
- [7] *RFC 2616 Fielding, Header Definitions of Hypertext Transfer Protocol HTTP/1.1*, линк: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
- [8] Софтверски пакет *Labcenter PROTEUS 8*, линк: <https://www.labcenter.com/>
- [9] *Arduino Library for Proteus Simulation* која садржи *Arduino Mega2560*, линк: <https://projectiot123.com/2019/01/04/arduino-library-for-proteus-simulation/>
- [10] *PIR Sensor Library for Proteus Simulation* коришћен са *Arduino-м*, линк: <https://www.theengineeringprojects.com/2016/01/pir-sensor-library-proteus.html>
- [11] Софтверски пакет *Oracle VM VirtualBox*, линк: <https://www.virtualbox.org/>
- [12] Софтверски пакет *WinImage*, линк: <http://www.winimage.com/>
- [13] Документација за *Microsoft .NET Xamarin.Forms cross-platform Applications*, линк: <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/>
- [14] *iOS Home App*, аутор: *Apple Inc.*, линк: <https://www.apple.com/ios/home/>
- [15] Софтверски пакет и развојно окружење *Microsoft Visual Studio 2017*, линк: <https://visualstudio.microsoft.com/>
- [16] Екстензија за развој *Arduino* софтвера кроз окружење *Microsoft Visual Studio*, *Visual Micro*, линк: <https://www.visualmicro.com/>
- [17] *XML Tutorial*, аутор: *w3schools*, линк: <https://www.w3schools.com/xml/>
- [18] *XML Parsers*, аутор: *javatpoint*, линк: <https://www.javatpoint.com/xml-parsers>
- [19] *Web документација за језик C++, C++ Reference, Standard C++ library*, линк: <http://www.cplusplus.com/reference/>
- [20] Материјали за израду вишенидног окружења са предмета *Оперативни Системи 1*, аутор *Драган Милићев*, линк: <http://os.etf.bg.ac.rs/OS1/index.htm>