Automatizirana hranilica i pojilica za jelene

Štivin, Nikola

SVEUČILIŠTE JOSIPA JURJA STROSSMAYER U OSIJEKU FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Stručni prijediplomski studij Elektrotehnika

AUTOMATSKA POJILICA I HRANILICA ZA ŽIVOTINJE

Završni rad

Nikola Štivin

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1S: Obrazac za ocjenu završnog rada na stručnom prijediplomskom studiju

Ocjena završnog rada na stručnom prijediplomskom studiju

Ime i prezime pristupnika:	Nikola Štivin
Studij, smjer:	Stručni prijediplomski studij Elektrotehnika, smjer Automatika
Mat. br. pristupnika, god.	A4718, 27.07.2021.
JMBAG:	0165089917
Mentor:	prof. dr. sc. Tomislav Keser
Sumentor:	prof. dr. sc. Damir Blažević
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 1:	prof. dr. sc. Tomislav Keser
Član Povjerenstva 2:	doc. dr. sc. Tomislav Galba
Naslov završnog rada:	Automatizirana hranilica i pojilica za jelene
Znanstvena grana završnog rada:	**Procesno računarstvo (zn. polje računarstvo)**
Zadatak završnog rada:	Razviti, izgraditi i testirati sustav automatiziranog doziranja hrane i vode u svrhu prehrane jelena u šumskom okruženju. Sustav izgraditi pomoći IoT sustava te omogućiti daljinsko upravljanje i nadzor sustava.

Datum ocjene pismenog dijela završnog rada od strane mentora:	22.01.2025.
Ocjena pismenog dijela završnog rada od strane mentora:	Izvrstan (5)
Datum obrane završnog rada:	3.3.2025
Ocjena usmenog dijela završnog rada (obrane):	Izvrstan (5)
Ukupna ocjena završnog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio stručni prijediplomski studij:	03.03.2025.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 03.03.2025.

Ime i prezime Pristupnika:	Nikola Štivin
Studij:	Stručni prijediplomski studij Elektrotehnika, smjer Automatika
Mat. br. Pristupnika, godina upisa:	A4718, 27.07.2021.
Turnitin podudaranje [%]:	6

Ovom izjavom izjavljujem da je rad pod nazivom**: Automatizirana hranilica i pojilica za jelene**

izrađen pod vodstvom mentora prof. dr. sc. Tomislav Keser

i sumentora prof. dr. sc. Damir Blažević

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Automatska hranilica i pojilica za životinje predstavlja važan dio suvremene tehnologije u poljoprivredi i stočarstvu, ali i u upravljanju divljim životinjama. Ovi uređaju su razvijene kako bi olakšali pružanje hrane i vode životinjama u situacijama kada nije moguće osigurati stalnu prisutnost čovjeka, kao što su udaljena područja ili loši vremenski uvjeti. U posljednjih nekoliko godina postali iznimno popularni, jer omogućavaju efikasniju upotrebu resursa i smanjenje radne snage. Na primjer, prema [1] velike farme u Danskoj, poput „*Lønholm Agro*“ koriste automatske hranilice u kombinaciji s robotskim sustavima za mužnju i praćenje zdravlja stoke, prema [2] farma muznih krava „Topolik“ u Hrvatskoj koristi hranilice kako bi prilagodila prehranu svake krave na temelju proizvodnje mlijeka, kao što je opisano u [3], u SAD-u automatske hranilice prilagođavaju obroke na temelju rasta i težine goveda, što optimizira rezultate tova i povećava učinkovitost proizvodnje mesa. U ovom radu će se automatska hranilica i pojilica promatrati u kontekstu upravljanja divljim životinjama, poput jelena, gdje će se koristiti kako bi se osigurala njihova ishrana tijekom perioda kada su prirodni resursi oskudni, poput zime. Ovakvi uređaji pomažu u održavanju stabilne populacije divljači i sprječavaju štetu na šumama i poljoprivrednim usjevima koju gladne životinje mogu izazvati. Karakteristike ovih uređaja ovise o njihovoj namjeni i okolini u kojoj se koriste. Hranilice za stoku na farmama često imaju veće kapacitete i robusniju konstrukciju kako bi izdržale svakodnevnu upotrebu, dok sustavi za divlje životinje moraju biti izdržljiviji i otporniji na vremenske uvjete, ali i dovoljno diskretni kako ne bi plašili životinje.

1.1. Zadatak završnog rada

Zadatak ovog rada je razviti i implementirati automatsku hranilicu i pojilicu za jelene, koja će omogućiti efikasnu distribuciju hrane i vode u skladu s potrebama životinja, uz minimalnu ljudsku intervenciju. Cilj je kreirati sustav koji može automatski prepoznati prisutnost životinja te prema tome dozirati odgovarajuću količinu hrane i vode, uz mogućnost nadzora putem mobilne aplikacije. Na kraju ovog rada očekuje se da će biti izrađena funkcionalna maketa koja će uspješno obavljati ove zadatke. U drugom poglavlju predstavljeno je teorijsko rješenje, uz prijedloge sklopovskog i programskog dizajna. Treće poglavlje bavi se realizacijom uključujući korišteno programsko okruženje. Detaljno su opisane korištene komponente i njihove karakteristike. U četvrtom poglavlju prikazana su testiranja koja su provedena, zajedno s rezultatima. Peto poglavlje donosi zaključak, gdje su sažeti glavni rezultati i predstavljeni prijedlozi za daljnje poboljšanje.

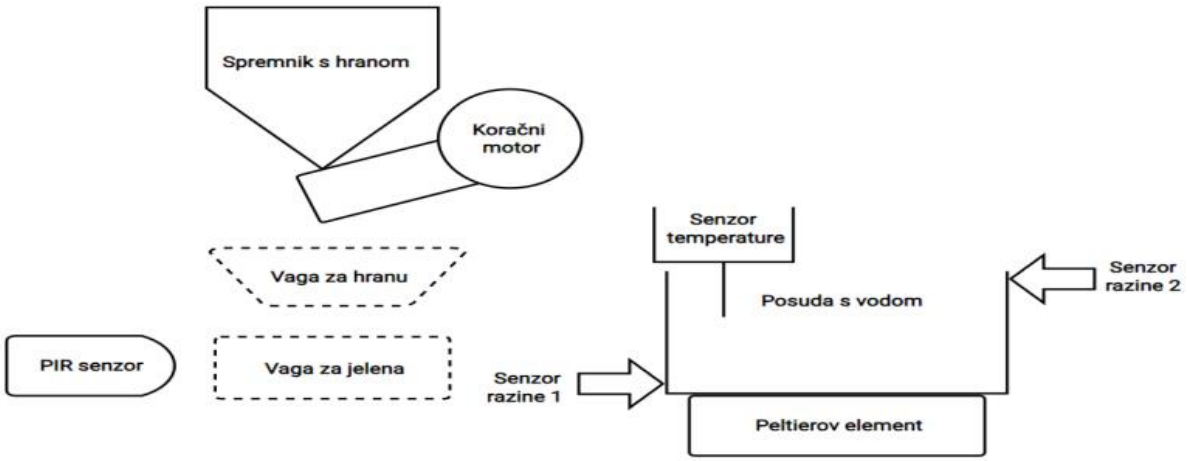
2. SUSTAV ZA AUTOMATSKO DOZIRANJE HRANE I VODE ZA JELENE UZ KONTROLU TEMPERATURE VODE

2.1. Teorijski osvrt na automatsku hranilicu i pojilicu

Ovaj sustav omogućava automatsko doziranje hrane i vode životinjama na temelju njihove prisutnosti i tjelesne mase. Sustav koristi različite senzore i elektroničke komponente za precizno praćenje i kontrolu procesa hranjenja i opskrbu vodom. Temelji se na nekoliko ključnih fizikalnih principa koji omogućavaju praćenje prisutnosti životinja, određivanje njihove tjelesne mase te regulaciju količine hrane i vode, uz praćenje temperature vode.

Za prepoznavanje prisutnosti životinja koristi se princip detekcije infracrvene radijacije. Pasivni infracrveni (*PIR*) senzori djeluju na temelju detekcije toplinskog zračenja koje emitiraju topla tijela, poput životinja. Infracrvena radijacija povećava se s porastom temperature, što omogućuje prepoznavanje životinja unutar zadanog područja. Za mjerenje tjelesne mase životinja, sustav koristi princip prepoznavanja težine na temelju sile koju životinja stvara svojim prisustvom. Izmjerena težina životinje omogućuje izračunavanje potrebne količine hrane na temelju unaprijed određenih kriterija. Ovakav pristup osigurava da svaka životinja dobije adekvatnu količinu hrane, prilagođenu njenoj masi.

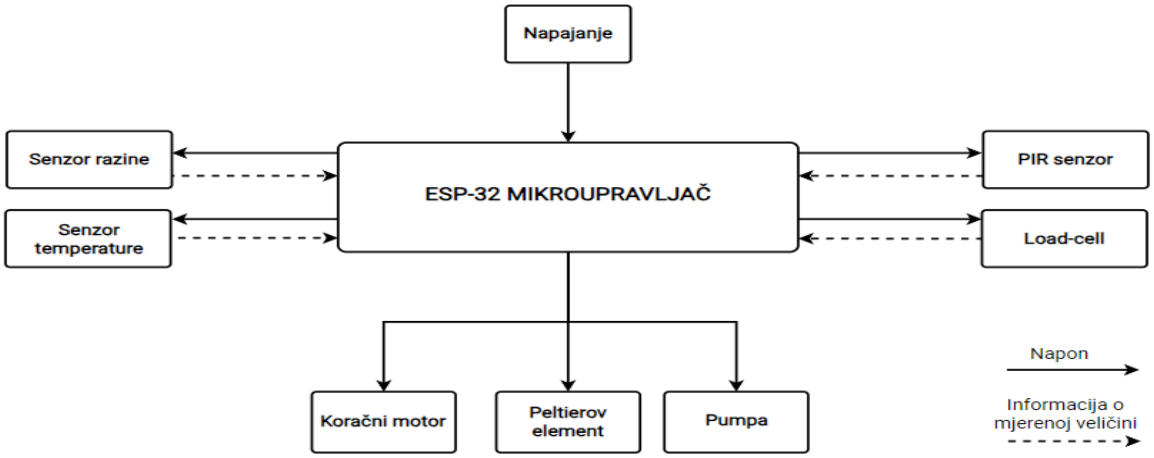
Za održavanje optimalne temperature vode koristi se Peltierov efekt, koji omogućuje premještanje topline između dvije površine pomoću električne struje. Ovaj efekt nastaje prolaskom struje kroz poluvodičke materijale, pri čemu se toplina s jedne strane elementa premješta na drugu, ovisno o smjeru struje. U sustavu za doziranje vode, ovaj princip omogućuje hlađenje ili grijanje vode unutar posude, čime se postiže željena temperatura bez obzira na vanjske uvjete. Mjerenje razine vode u posudi temelji se na Arhimedovu zakonu, prema kojem tijelo uronjeno u tekućinu doživljava potisak jednak težini istisnute tekućine. Ovaj princip se koristi za rad senzora razine (engl. *Float senzor)* koji detektiraju promjenu razine vode u posudi. Sustav detektira kada razina vode padne ispod zadane vrijednosti te aktivira postupak dopunjavanja spremnika. Kompletan koncept sustava može se vidjeti na slici 2.1., gdje su prikazane sve ključne komponente.



Slika 2.1. *Skica osnovnog prikaza automatske hranilice i pojilice*

2.2. Prijedlog sklopovskog rješenja

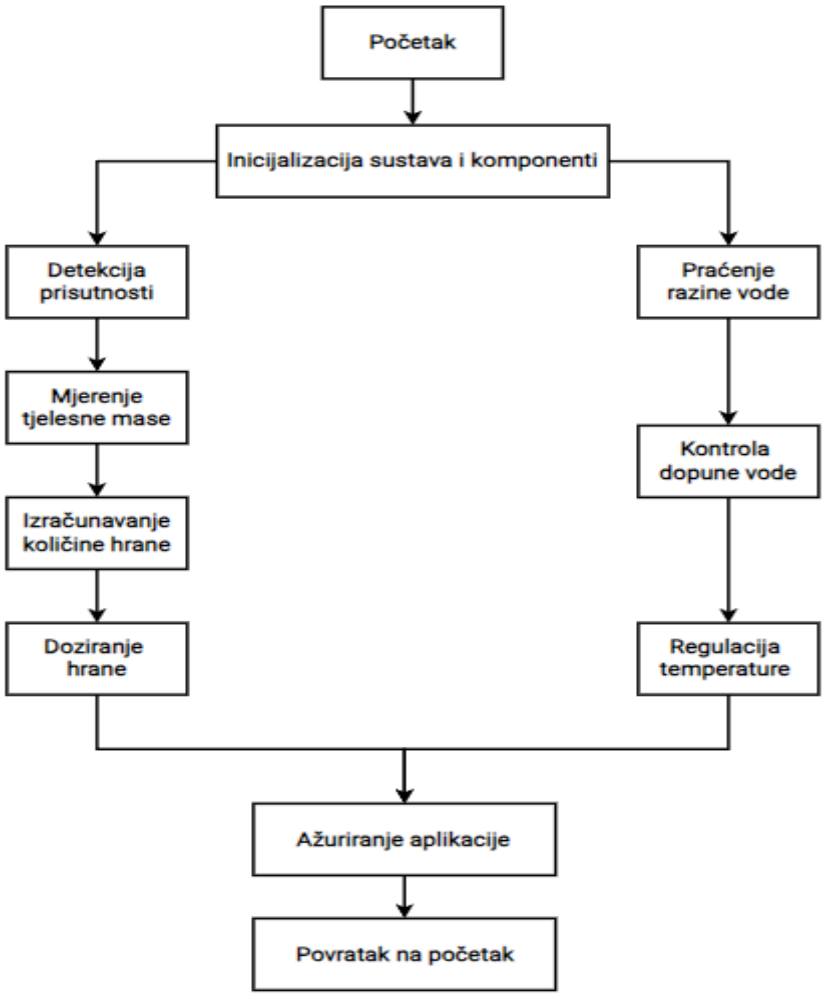
Na slici 2.2. prikazan je predloženi blokovski dijagram sustava koji prikazuje osnovne komponente sustava i njihove međusobne veze. Sustav bi se temeljio na *ESP*-32 mikroupravljaču koji bi služio kao središnji upravljački modul. On bi obrađivao podatke prikupljene sa senzora i upravljao radom izlaznih uređaja na temelju tih podataka. Sustav senzora koristio bi pasivni infracrveni senzor - *PIR* za detekciju prisutnosti životinja, kao i senzor opterećenja (engl. *Load-cell*) za mjerenje njihove težine. Posuda za vodu bila bi opremljena senzorima razine, koji bi mjerili količinu vode u spremniku i omogućili automatsku regulaciju dopune vode. Uz pomoć senzora temperature i Peltier elementa, precizno bi se održavala temperatura vode unutar zadanih granica.



Slika 2.2. *Prijedlog sklopovskog rješenja hranilice i pojilice*

2.3. Prijedlog programskog rješenja

Na slici 2.3. prikazan je jednostavan blok dijagram koji prikazuje prijedlog programskog rješenja za automatsku hranilicu i pojilicu za jelene, obuhvaćajući sve glavne funkcije sustava. Program započinje inicijalizacijom sustava i komponenti. To uključuje postavljanje svih senzora, motora i drugih uređaja potrebnih za pravilno funkcioniranje sustava. Dijagram je podijeljen na dva glavna dijela: algoritam za hranjenje i algoritam za opskrbu vodom. U oba slučaja, ciljevi su automatizirano prepoznavanje potreba životinja, prilagođavanje količine hrane i vode te regulacija vanjskih uvjeta (temperature). Nakon što su procesi hranjenja i dopune vode završeni, sustav ažurira podatke u aplikaciji, a program se vraća na početak gdje čeka sljedeću interakciju.



Slika 2.3. *Prijedlog programskog rješenja*

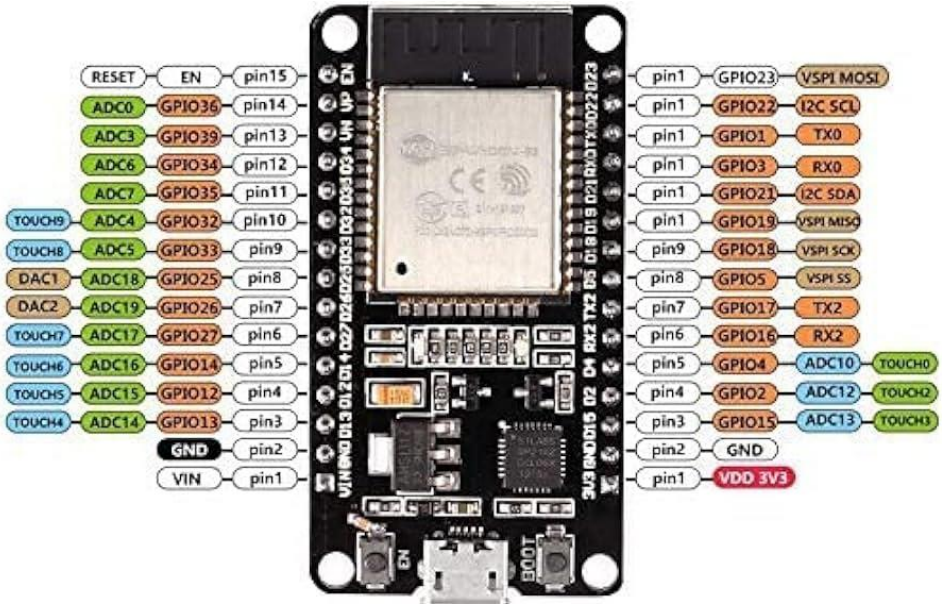
3. REALIZACIJA AUTOMATSKE POJILICE I HRANILICE

3.1. Korištene komponente, alati i programsko okruženje

U sljedećim potpoglavljima su opisane ključne komponente korištene u ovom radu, uz naglasak na njihovu ulogu, funkciju i najvažnije tehničke karakteristike koje su bitne za njegovu realizaciju. Detaljan popis tehničkih specifikacija svakog elementa dostupan je u prilogu 7.1. Hardverske komponente, na koji se upućuje za dodatne informacije.

ESP-32 mikroupravljač

Prva komponenta je *ESP*-32 mikroupravljač, razvijen od strane tvrtke *Espressif,* koji se koristi kao ključna komponenta ovog rada. Slika 3.1. prikazuje *ESP32*-WROOM-32 modul zajedno s njegovim pinovima. Ova razvojna pločica koristi *ESP32*-WROOM model, koji integrira *Wi-Fi* i *Bluetooth* funkcionalnosti te omogućuje bežičnu komunikaciju s drugim uređajima. Koristi se u širokom rasponu primjena, od niskopotrošnih senzorskih mreža do složenih zadataka, kao što su kodiranje glasa, streaming glazbe i dekodiranje MP3 formata.



Slika 3.1. *ESP-32 mikroupravljač i njegovi pinovi [4]*

Koračni motor 28BYJ-48 s driverom

Prema [5], 28BYJ-48 je često korišten koračni (engl. *Stepper*) motor u malim aplikacijama zbog svoje jednostavnosti upravljanja i pristupačnosti. U ovom radu koristi se pet takvih motora za precizno upravljanje rotacijom, što je ključno za pravilno doziranje hrane. Na slici 3.2. prikazan je motor s upravljačkim sklopom (engl. *Driver*).

Slika 3.2. *Stepper motor s driverom* [6]

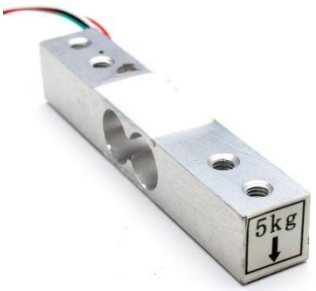
PIR senzor pokreta

Kako se navodi u [7], *PIR* (pasivni infracrveni) senzor pokreta je pouzdan i učinkovit senzor koji se koristi za detekciju pokreta mjerenjem promjena u razinama infracrvene radijacije koju emitiraju okolni objekti. Široko se koristi u sigurnosnim sustavima, automatskim kontrolama rasvjete i drugim aplikacijama koje zahtijevaju detekciju pokreta. Na slici 3.3. prikazan je *PIR* senzor pokreta.

Slika 3.3. *PIR senzor pokreta* [8]

Load-cell senzori + *HX711* breakout

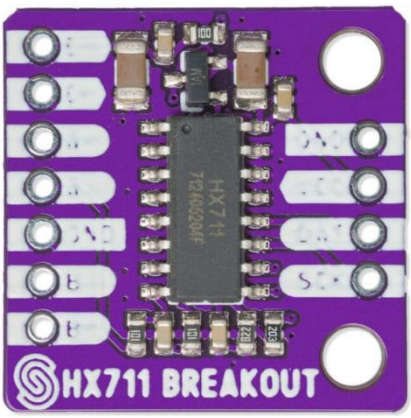
Kao što je opisano u [9], *load-cell* senzor (*Slika* 3.4.) je elektronički uređaj dizajniran za mjerenje sile ili naprezanja. Senzor se sastoji od fleksibilne izolirane podloge s metalnim folijskim uzorkom. Koristi se za mjerenje mase životinja. Kada životinja stane na vagu, senzor detektira promjenu otpora uzrokovanu promijenjenom silom, što omogućava precizno mjerenje težine. Ovaj senzor koristi *Wheatstoneov* most za bilježenje promjena u otporu.



Slika 3.4. *Load-cell senzor s maksimalnim opterećenjem od 5 kg* [10]

Važno je napomenuti da se ovaj senzor koristi u kombinaciji s odgovarajućim pretvaračem i mora se kalibrirati s poznatim težinama kako bi se osigurala točnost mjerenja. U ovom slučaju korišten je *HX711 breakout*.

Prema [11], *HX711 breakout* (*Slika* 3.5.) uključuje analogno-digitalni pretvornik s rezolucijom od 24 bita i dodatnu pojačavajuću elektroniku koja ga čini prikladnim za mjerenje *load-cell* senzora ili drugih senzora s četiri žice povezane u *Wheatstoneovom* mostu.



Slika 3.5. *Ploča s load-cell pojačalom HX711* [12]

Senzor za mjerenje razine vode

Prema [13], senzor za mjerenje razine (engl. *Float senzor*) koristi se za automatsko praćenje i regulaciju razine tekućine unutar spremnika. Kada razina tekućine padne ispod određene granice, senzor šalje signal za punjenje, dok, kada razina prijeđe određenu granicu, senzor pokreće akcije poput zaustavljanja punjenja. Njegov izgled prikazan je na slici 3.6.

Slika 3.6. *Float senzor* [14]

Senzor temperature DS18B20

Kako se navodi u [15], DS18B20 je digitalni senzor temperature koji se koristi za precizno mjerenje temperature. Komunicira putem 1-žične (engl. *1-Wire*) sabirnice koja po definiciji zahtijeva samo jednu podatkovnu liniju (i uzemljenje) za komunikaciju s mikroprocesorom. Također, DS18B20 može uzimati energiju izravno iz podatkovne linije (parazitska snaga), eliminirajući potrebu za vanjskim napajanjem. Slika 3.7. prikazuje DS18B20 senzor temperature.-

Slika 3.7. *DS18B20 senzor temperature [16]*

Peltierov element

Kao što je opisano u [17], Peltierov element (*Slika* 3.8.) je poluvodički uređaj za grijanje i hlađenje koji koristi fenomen koji se naziva Peltierov učinak, gdje se toplina prenosi protokom struje između dva različita metala. Koristi se u sustavu za kontrolu temperature vode. Njegova osnovna funkcija je omogućiti grijanje ili hlađenje vode u posudi ovisno o potrebi, čime se osigurava stabilna temperatura u promjenjivim uvjetima okoline.

Temperaturni raspon: Ovaj modul je dizajniran za rad u širokom temperaturnom rasponu, od -30°C do 70°C.

Slika 3.8. *Peltier element TEC1-1270 [18]*

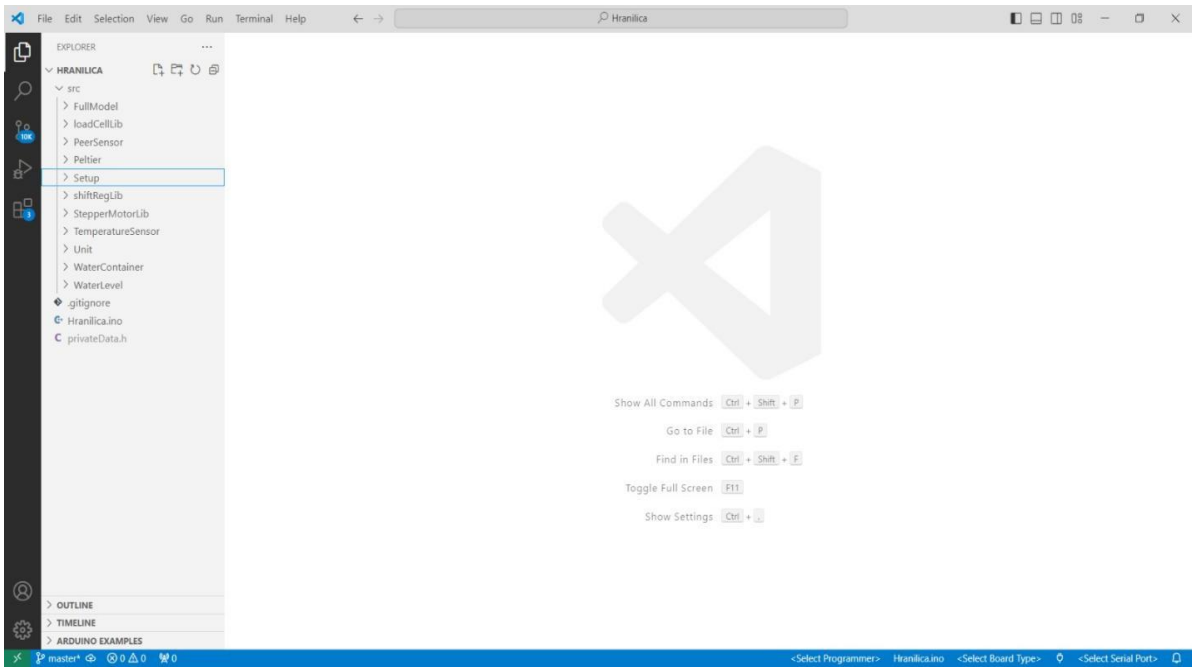
Pumpa za vodu

Prema [19], pumpa za vodu je jednostavan i učinkovit uređaj koji se koristi za premještanje vode unutar sustava. Njezin elektromotor smješten je u vodootpornom kućištu, što omogućuje pouzdan rad u tekućinama. Pumpa se nalazi u spremniku vode unutar kućišta i putem gumenog crijeva pumpa vodu u posudu izvan kućišta. Slika 3.9. prikazuje pumpu za vodu korištenu u ovom radu.

Slika 3.9. *Pumpa za vodu [20]*

Visual Studio Code

Za izradu softvera korišten je *Visual Studio Code*, besplatan i otvoren izvorni editor koda. Odabran je zbog svoje bogate funkcionalnosti, podrške za mnoge programske jezike i mogućnosti dodavanja ekstenzija koje omogućavaju prilagodbu i proširenje alata prema potrebama rada. Korisničko sučelje *Visual Studio Code* je organizirano i intuitivno, što dodatno olakšava razvoj i ispravljanje grešaka u programu. Korisničko sučelje je prikazano na slici 3.10.



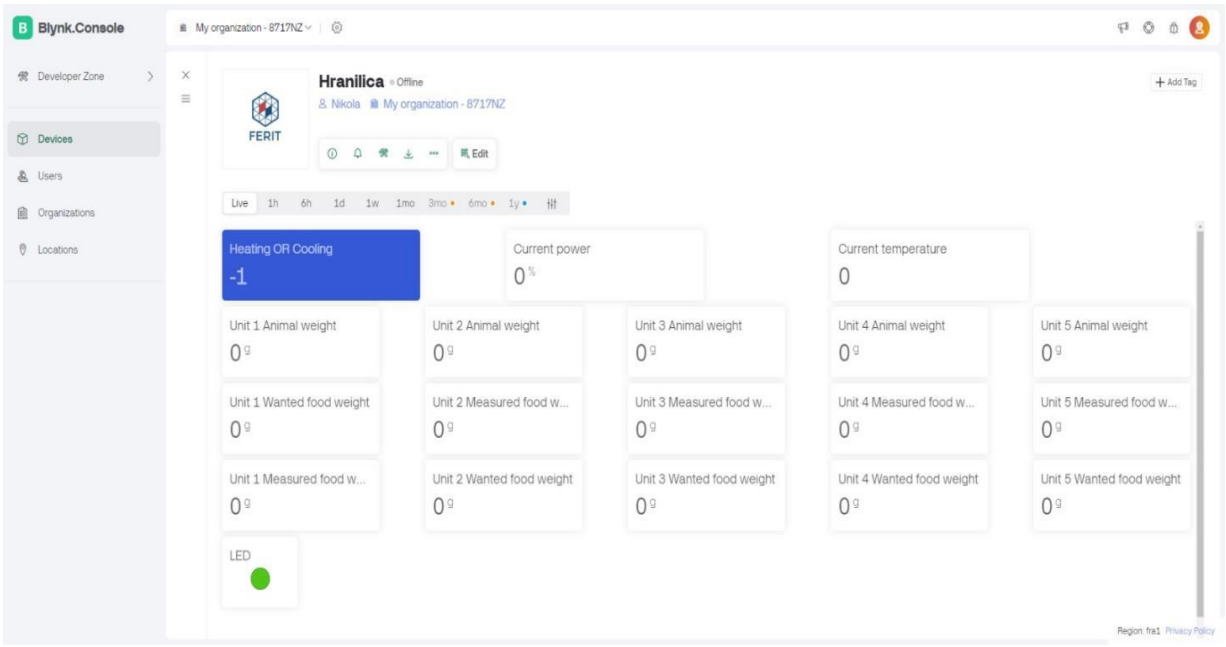
Slika 3.10. *Visual Studio Code - sučelje*

KiCad

Prema [21], *KiCad* je softverski paket otvorenog koda (engl. O*pen-source*) za elektroničko dizajniranje, koji omogućuje kreiranje shema i *PCB* (engl. *Printed Circuit Board*) rasporeda. Za izradu električne sheme korišten je KiCad, zahvaljujući njegovim jednostavnim i pristupačnim alatima za dizajn.

Blynk

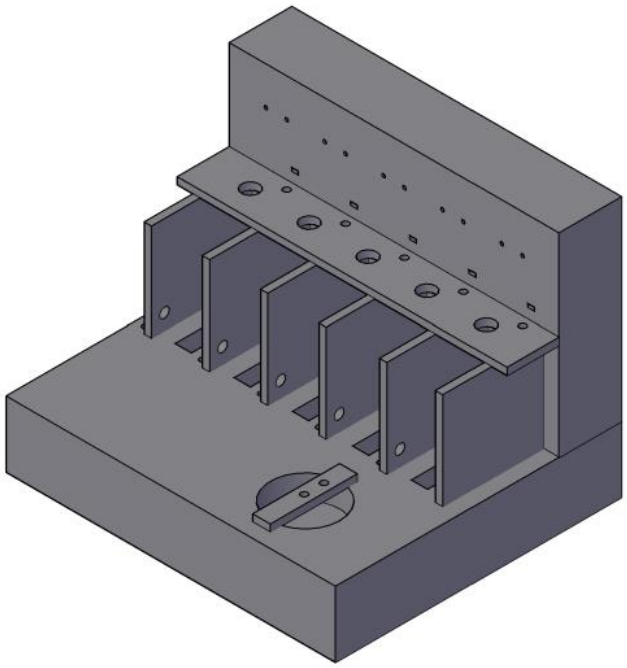
Blynk je platforma za *IoT* (engl. *Internet of Things*) koja omogućuje jednostavno upravljanje i praćenje uređaja putem mobilne aplikacije. Aplikacija koristi *Wi-Fi* ili mobilnu mrežu za povezivanje s mikroupravljačem. U ovom radu, *Blynk* aplikacija se koristi za praćenje težine životinje na svakoj stanici i količinu hrane koju životinja dobije. Osim toga prikazana je kontrola sustava za grijanje i hlađenje, te temperatura vode u sustavu. Slika 3.11. prikazuje *Blynk* sučelje korišteno u ovom radu.



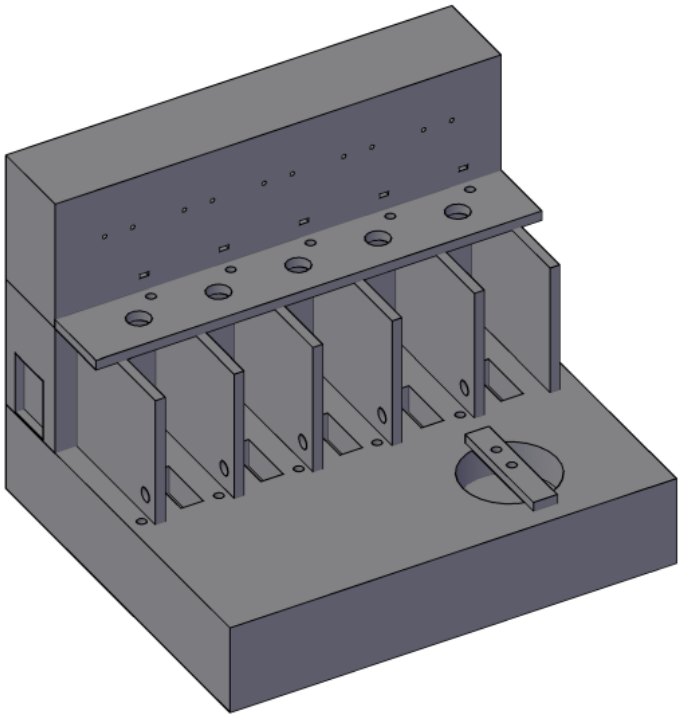
Slika 3.11. *Blynk sučelje*

3.2. Realizacija konstrukcijskog i sklopovskog rješenja

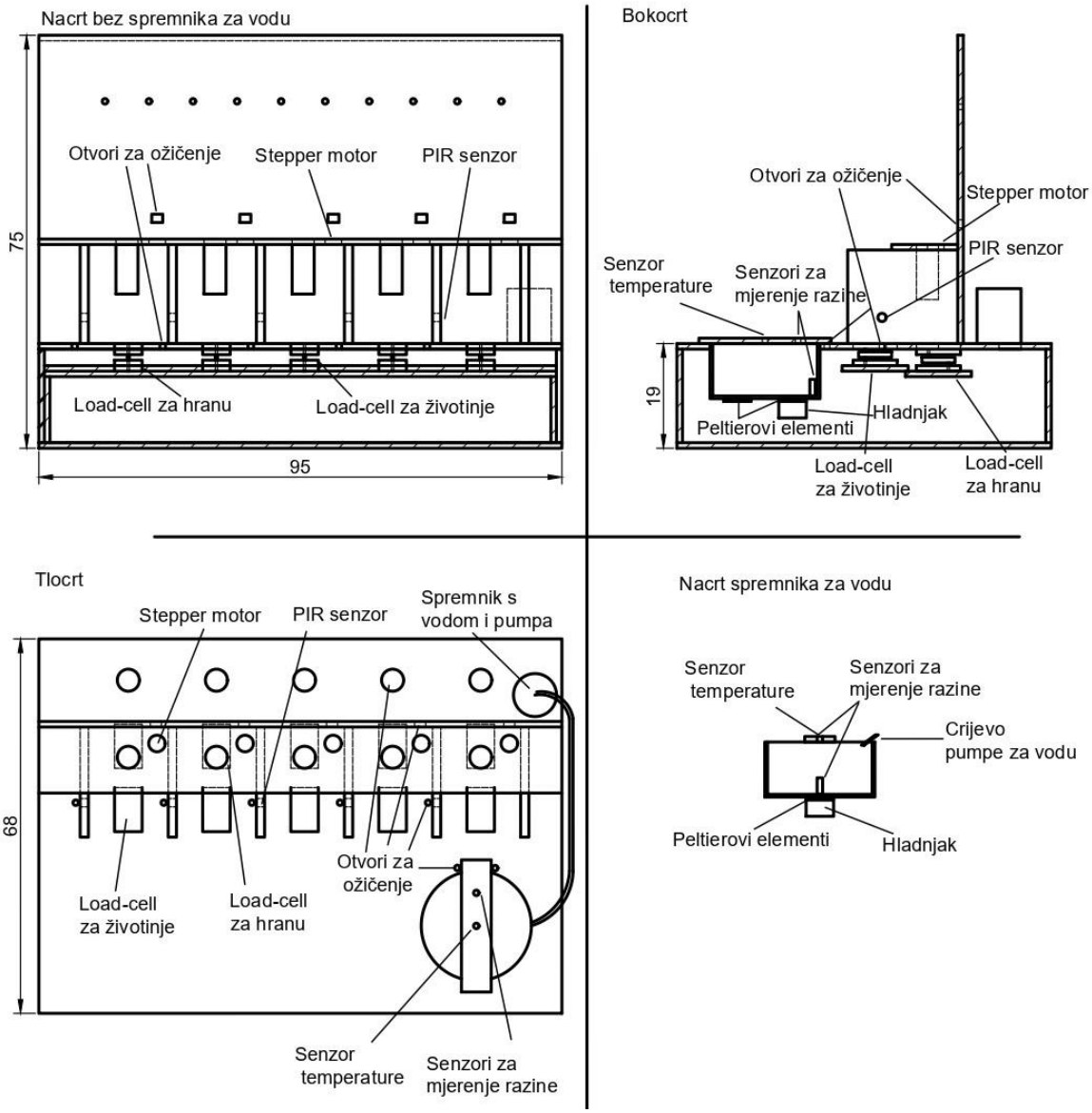
Prilikom izrade same makete automatske hranilice i pojilice, koja je prikazana 3D modelom na slikama 3.12. i 3.13., zbog potrebe za čvrstim, masivnim i stabilnim kućištem u koje se može smjestiti pet stanica za hranu i posuda za vodu, korištene su OSB ploče koje su pričvršćene vijcima. Dimenzije donje stranice su 68 x 95 cm, dimenzije bočnih strana 19 cm, dok je visina cijele konstrukcije 75 cm što se može vidjeti na slici 3.14. Za spremnike za hranu korištene su plastične boce, privezane plastičnim trakama za kućište. Na krajevima plastičnih boca se nalaze poklopci koji su spojeni na stepper motore, što je prikazano na slici 3.15. Ispod svakog poklopca se nalazi plastična cijev, kako se hrana ne bi rasipala. Svaka stanica je međusobno odvojena pregradom, unutar koje je izbušena rupa za postavljanje *PIR* senzora za detekciju.



Slika 3.12. *3D prikaz automatske hranilice i pojilice za životinje s desne strane*



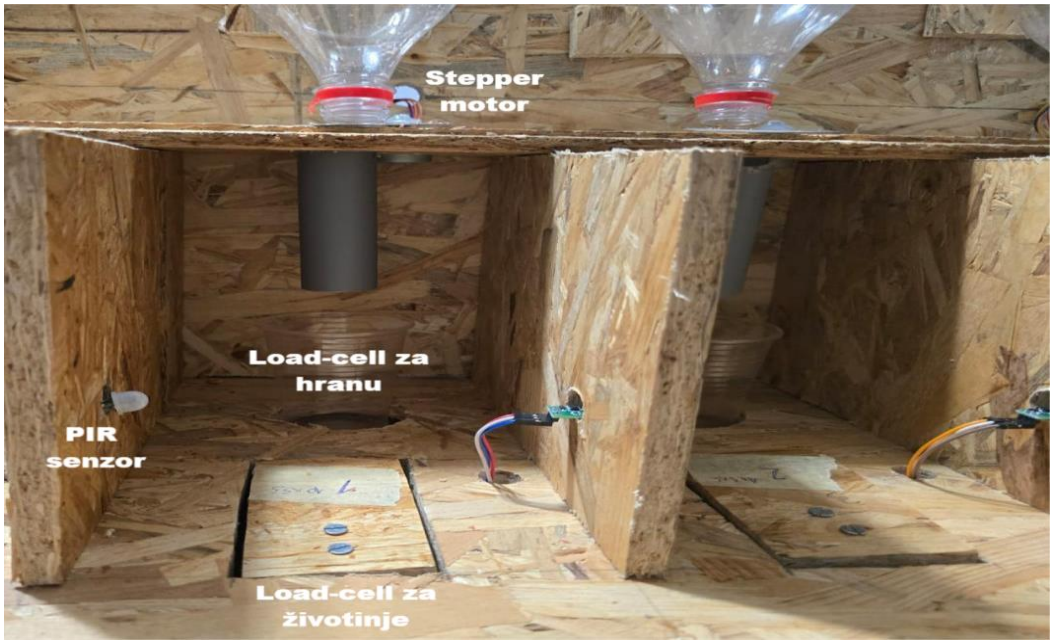
Slika 3.13. *3D prikaz automatske hranilice i pojilice za životinje s lijeve strane*



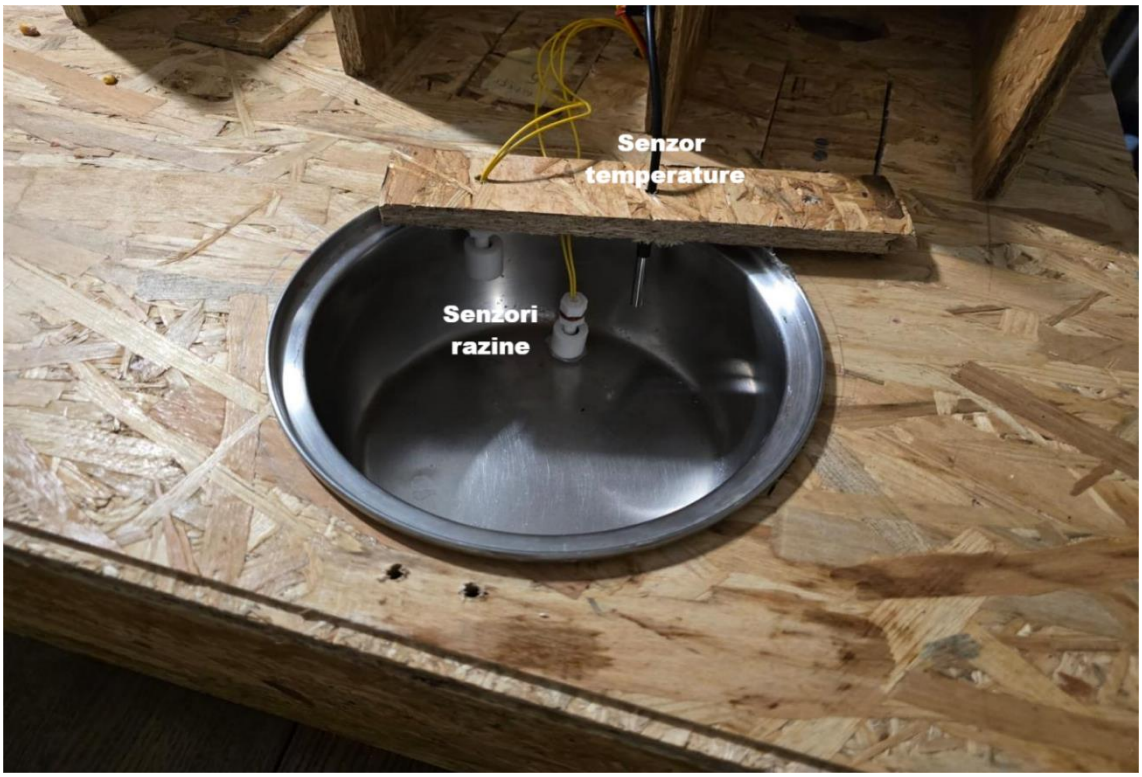
Slika 3.14. *2D prikaz automatske hranilice i pojilice za životinje*

U svakoj stanici su još dodatno izbušena mjesta za postavljanje vaga, jedna za životinju i druga za hranu. Slika 3.16. prikazuje jednu kompletnu stanicu. Kao spremnik za vodu korištena je plastična boca, zapremnine 3 l u koji je uronjena podvodna pumpa za vodu te gumeno crijevo. Slika 3.17. prikazuje posudu za vodu koja se sastoji od metalnog lonca. Unutar lonca su smještena dva senzora razine i senzor temperature, dok se ispod lonca nalazi Peltierov element. Završni izgled kompletne makete prikazan je na slici 3.18.

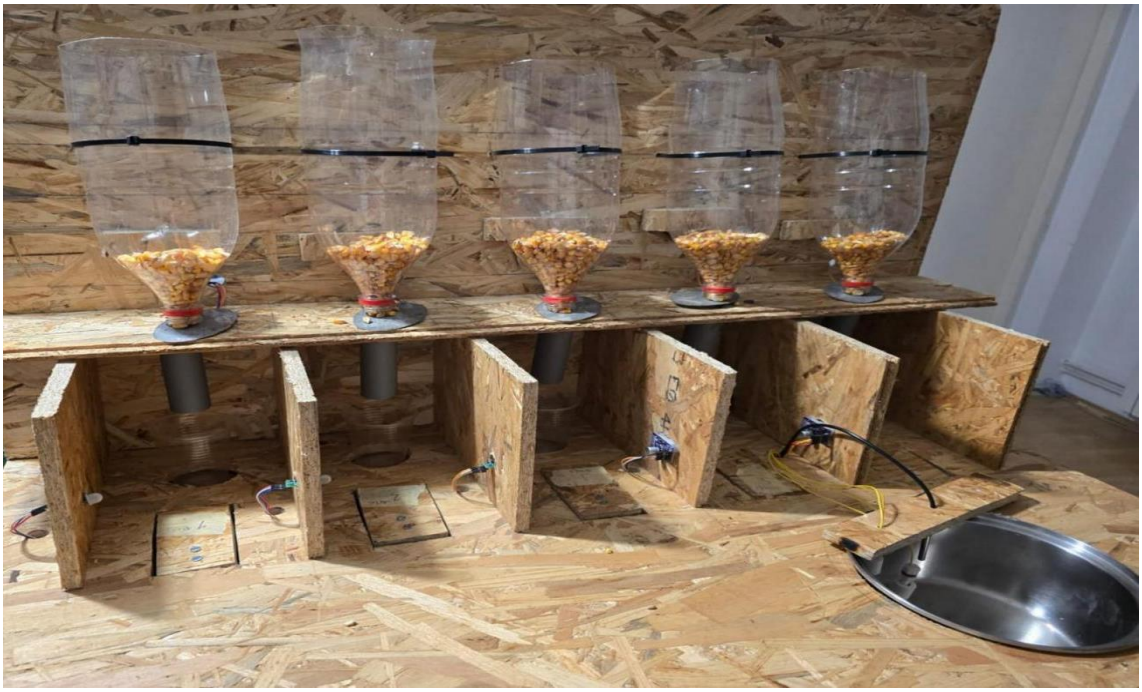
Slika 3.15. *Spremnik za hranu*



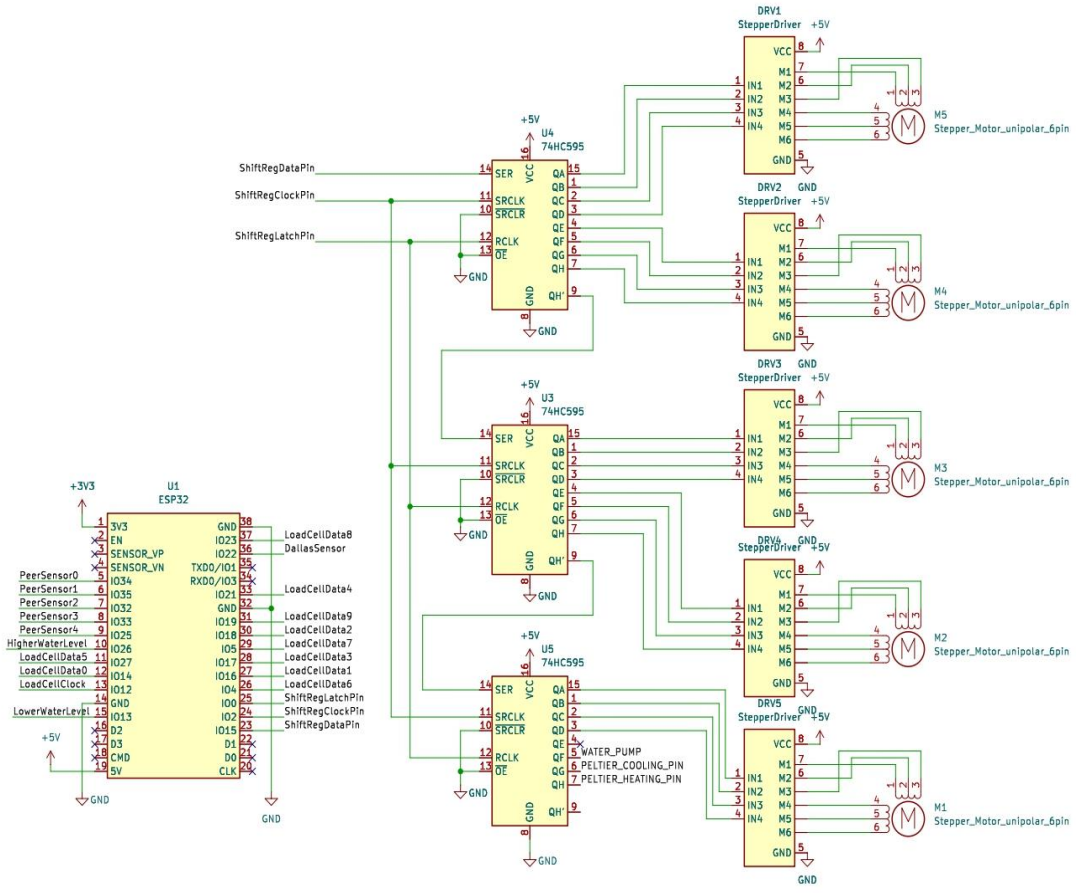
Slika 3.16. *Prikaz stanice za životinje*



Slika 3.17. *Posuda za vodu*



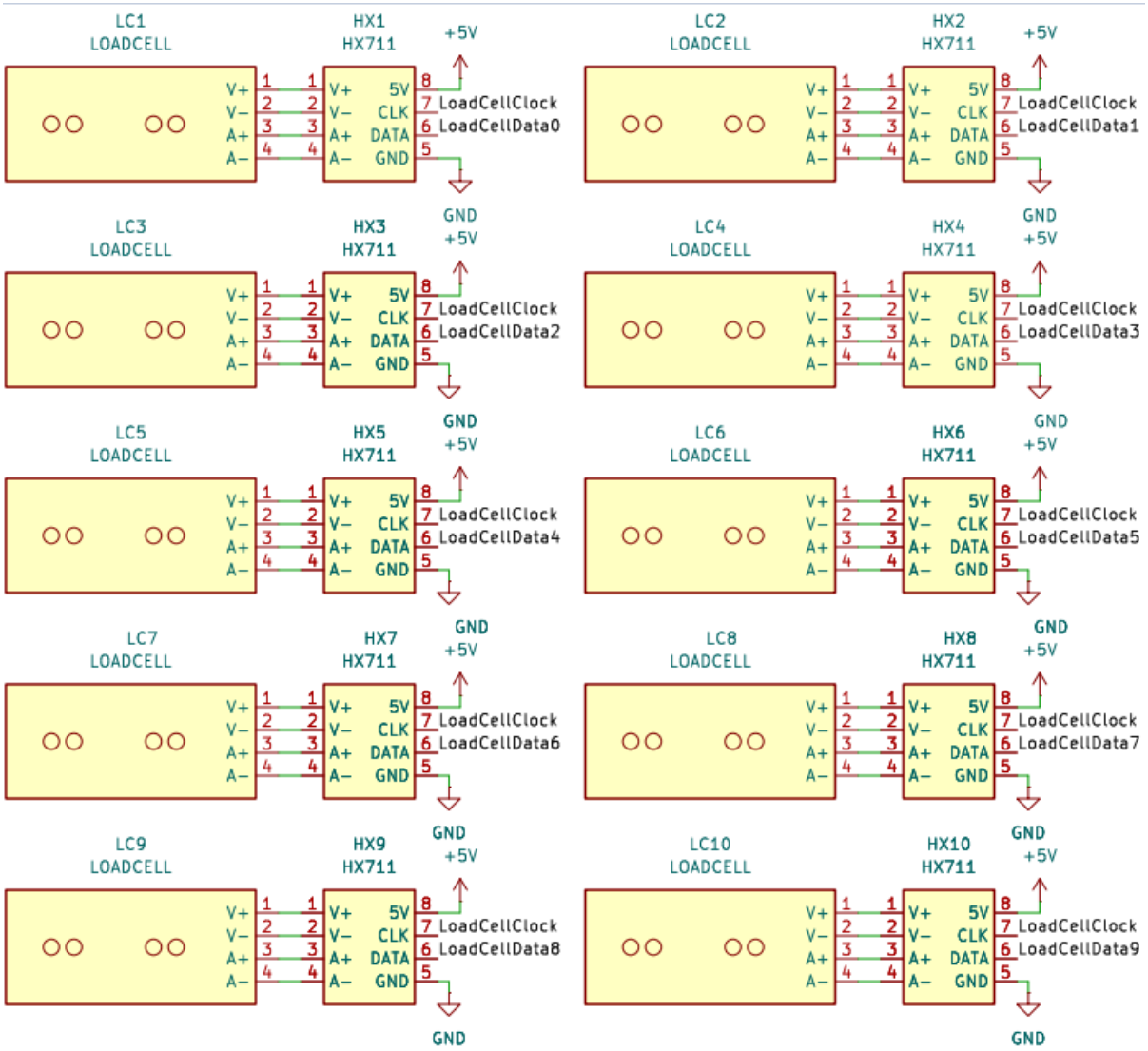
Slika 3.18. *Završni izgled makete*



Slika 3.19. *Prikazuje upravljanje stepper motoria i serijsku komunikaciju s mikrokontrolerom*

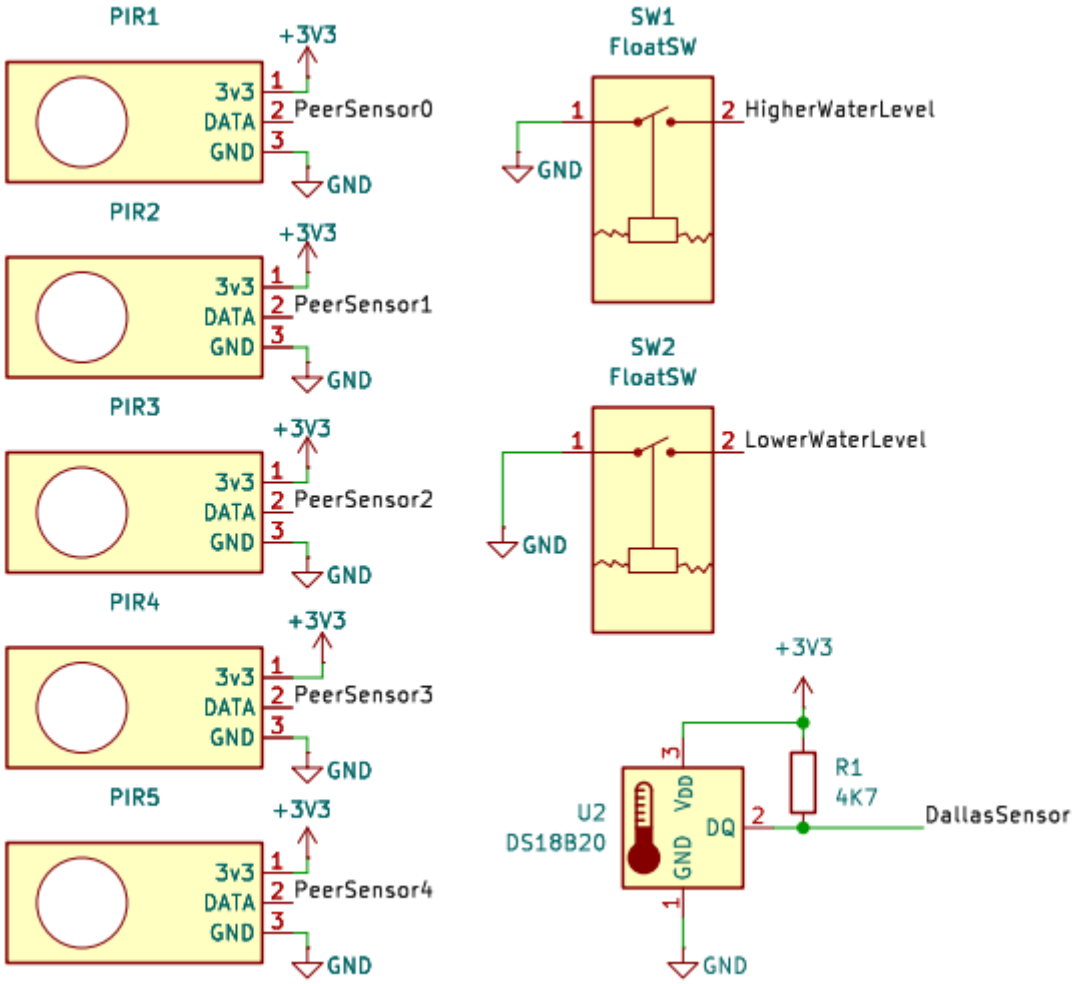
Zbog složenosti električne sheme i potrebe za jasnijim prikazom, shema je podijeljena na više dijelova. Kao što je prikazano na slici 3.19. za upravljanje korištenih *stepper* motora (M1-M5) koriste se *stepper driveri* (DRV1-DRV5). Za serijsku komunikaciju s mikrokontrolerima koriste se *74HC595 shift* registri. *Shift* registri su korišteni zbog toga što omogućavaju upravljanje velikim brojem pinova pomoću samo nekoliko digitalnih pinova s *ESP32* mikrokontrolera (ShiftRegDataPin, ShiftRegClkPin, ShiftRegLatchPin).

Load-cell senzori (LC1-LC10) koriste se za mjerenje težine ili sile. Svaka jedinica povezana je s odgovarajućim pojačalom *HX711*, koje omogućava precizno očitavanje analognih signala iz senzora. *Data* i *clock* pinovi s *HX711* pojačala povezuju se direktno na *ESP32* mikrokontroler, što se može vidjeti na slici 3.20, koji obrađuje podatke i omogućuje mjerenje mase. *PIR* senzori (*PIR*1-*PIR*5) detektiraju prisutnost ili kretnje u blizini. Jedan pin im je spojen na napajanje 3.3V, drugi na *GND* i posljednji pin je spojen na *ESP32*.



Slika 3.20. *Shematski prikaz povezivanja load-cell senzora s ESP-32*

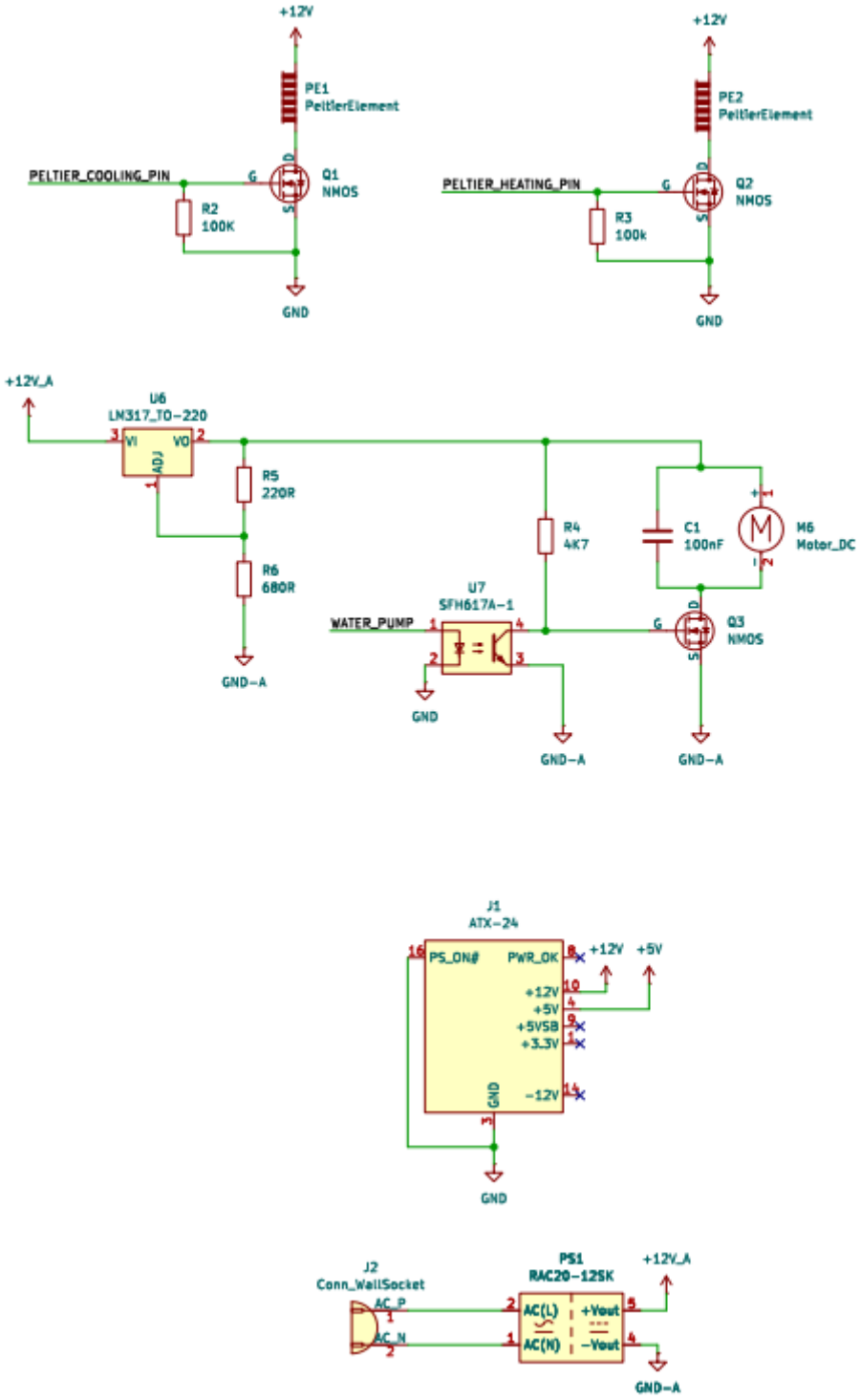
Senzor temperature (DS18B20) služi za mjerenje temperature. Podaci se prenose direktno na *ESP32*, što omogućuje upravljanje sustavom na temelju temperature (*Slika 3.21.*).



Slika 3.21. *Shematski prikaz povezivanja PIR senzora, senzora razine i senzora temperature DS18B20 s ESP-32*

Za napajanje sustava korištena su dva izvora napajanja prikazana na slici 3.22. Prvi izvor je *AC-DC* pretvarač (PS1) koji napaja samo pumpu za vodu. Ova komponenta pretvara *AC* mrežni napon u stabiliziran *DC* napon. Drugo napajanje je *ATX* napajanje (J1) koje se često koristi u računalima. Iz ovog izvora dobiva se nekoliko napona: +12V, +5V, +3.3V, -12V i +5VS od kojih mi koristimo samo +12V i +5 V. Peltier elementi (PE1 i PE2) koji se koriste za grijanje i hlađenje djeluju tako da, ovisno o smjeru struje, jedna strana postaje vruća, dok druga postaje hladna. Kontroliraju se pomoću *MOSFET* (*NMOS*) tranzistora (Q1 i Q2). Q1 kontrolira hlađenje, dok Q2 kontrolira grijanje. Aktiviranjem odgovarajućih pinova na *MOSFET*-ima (*PELTIER_COOLING_PIN i PELTIER_HEATING_PIN*), koji su označeni na slici 3.22., omogućuje se protok struje kroz elemente, što rezultira promjenu temperature. Pumpa (*DC* motor (M6)) također se kontrolira preko tranzistora (Q3) koji se aktivira preko *optokaplera* U7 (*SFH617A*).

Optokapler služi za izolaciju signala što omogućava sigurnu kontrolu motora (pumpe) bez direktne električne veze s mikrokontrolerom. Regulator napona *LM317* omogućuje preciznu kontrolu napona za motor.

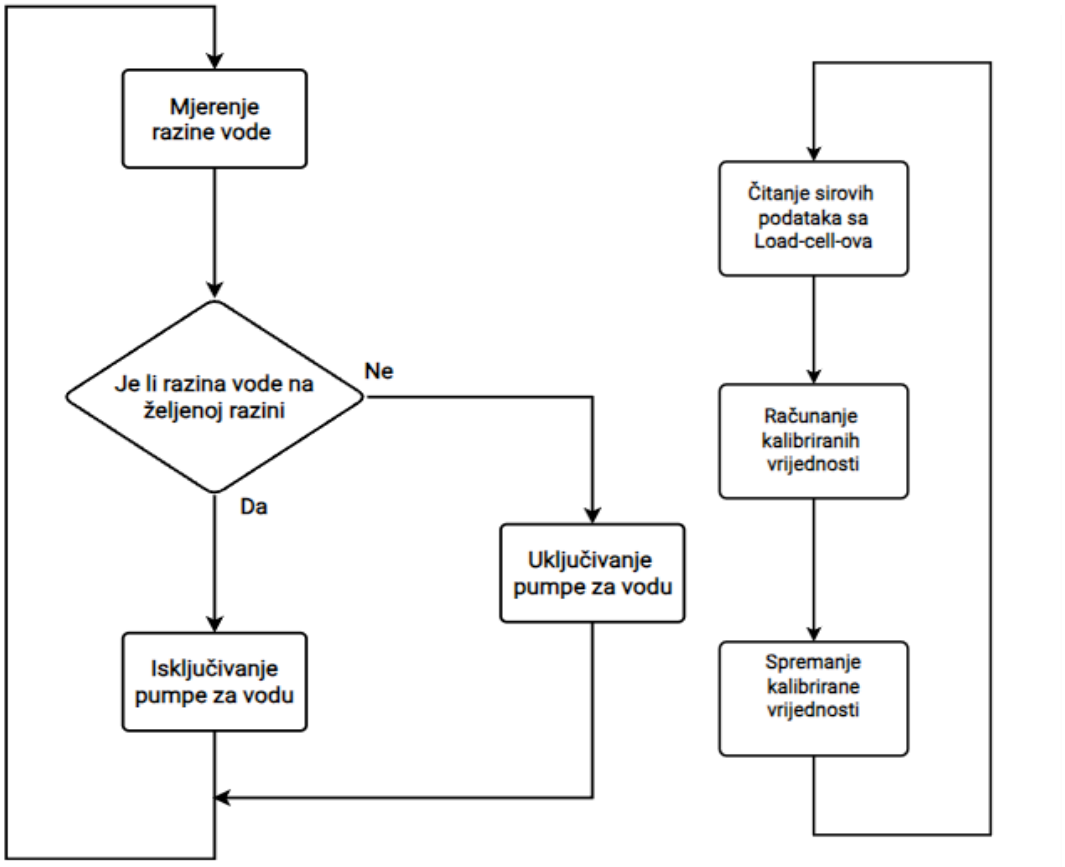


Slika 3.22. *Prikazuje shemu napajanja sustava*

3.3. Realizacija programskog rješenja

U ovom dijelu rada detaljno će se objasniti programsko rješenje prikazano s pomoću blok dijagrama.

Lijevi dijagram sa slike 3.23. opisuje kontrolu razine vode. Prva operacija koja se izvodi u petlji je mjerenje trenutne razine vode u posudi. Ulazni argument ovdje predstavlja senzor koji vrijednost razine vode šalje sustavu. Ova funkcija vraća rezultat koji je vrlo bitan za sljedeći korak. Sljedeći korak je usporedba izmjerene vrijednost s unaprijed zadanom vrijednošću. Ovo je ključni korak koji odlučuje o daljnjim operacijama. Ako je razina vode na zadanoj razini ili iznad, sljedeći korak je isključivanje pumpe za vodu, a ukoliko je razina vode ispod željene razine, sljedeći korak je uključivanje pumpe za vodu. Nakon bilo kojeg od ova dva ishoda program se vraća na početak ciklusa, ponovno mjeri razinu vode i prolazi kroz cijelu petlju koja omogućava kontinuiranu kontrolu i regulaciju razine vode u stvarnom vremenu.

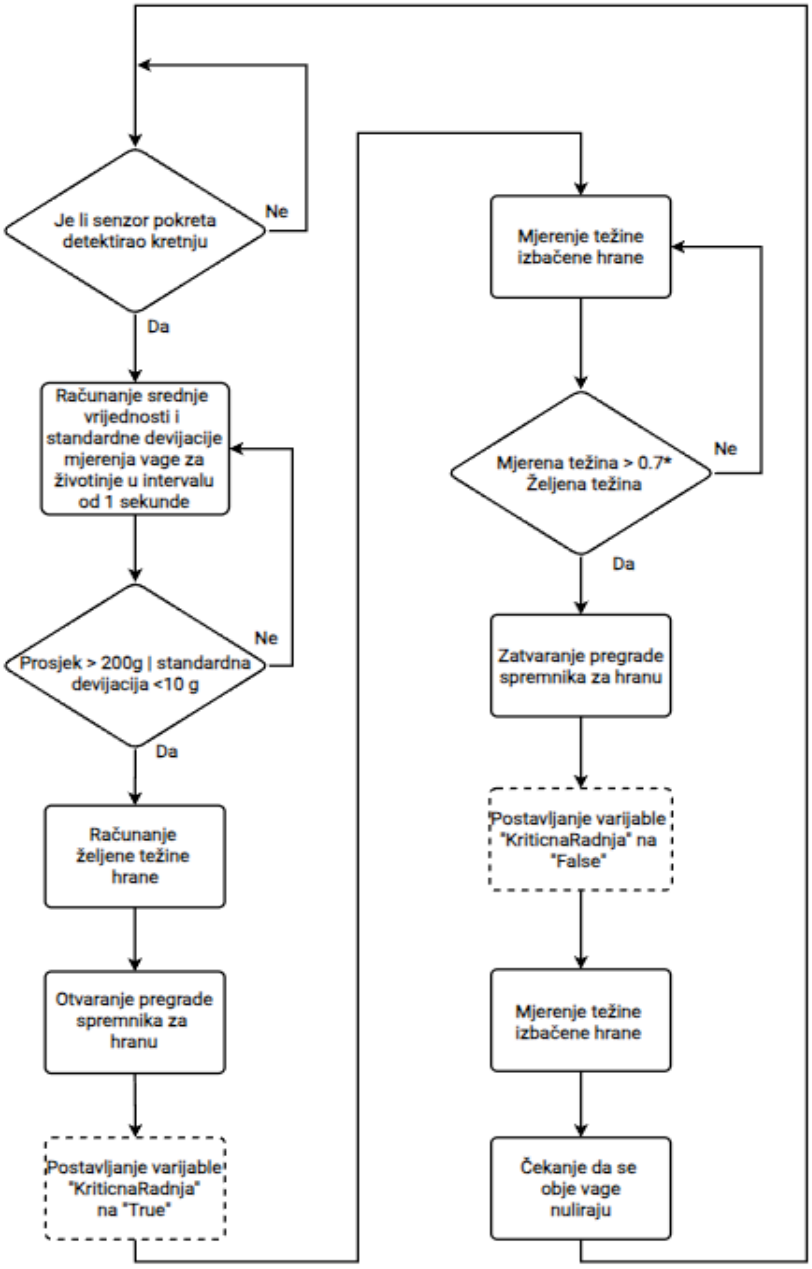


Slika 3.23. *Blokovski prikaz za kontrolu razine vode i upravljanje load-cell-ovima*

Upravljanje *load-cell* senzorima prikazano je desnim dijagramom na slici 3.23. Proces započinje čitanjem sirovih (neobrađenih) podataka s *load-cell* senzora. Signal koji dolazi sa senzora je ulazni argument ove funkcije. Ova funkcija vraća izlazni argument u obliku neobrađene vrijednosti, koja bez daljnje obrade nije korisna. Nakon očitavanja sirovih podataka, slijedi njihova kalibracija, koja podrazumijeva pretvaranje sirovih podataka u stvarne vrijednosti koje imaju smisla. Ova funkcija prima ulazni argument iz prethodnog koraka i prema poznatim koeficijentima kalibracije izračunava stvarnu vrijednost sile ili težine. Izlazni argument postaje rezultat kalibracije. Posljednji korak u petlji je spremanje kalibriranih vrijednosti. Program uzima i pohranjuje određene podatke u memoriju sustava. Ulazni argument je kalibrirana vrijednost iz prethodnog koraka, a izlazni argument je potvrda uspješno pohranjenih podataka. Nakon što su kalibrirane vrijednosti spremljene proces se vraća na početak.

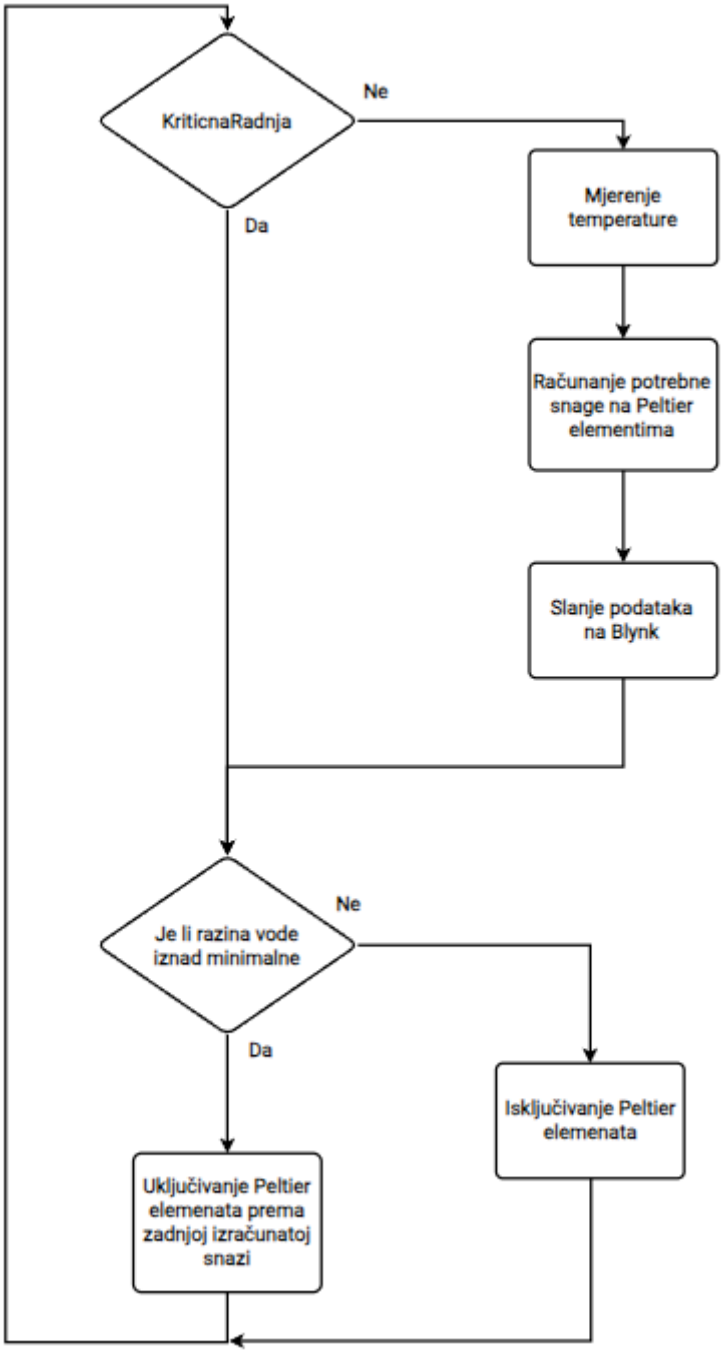
Blok dijagram prikazan slikom 3.24. predstavlja upravljanje hranilicom. Program započinje provjerom stanja senzora pokreta. Ako senzor nije detektirao kretnje, program se vraća na početak, u suprotnom program prelazi na sljedeći korak računanje srednje vrijednosti i standardne devijacije. U intervalu od jedne sekunde mjeri se srednja vrijednost i standardna devijacije podataka dobivenih sa senzora vage koja prati težinu životinje. Zatim je na redu provjera težine životinje. Ako je prosjek veći od 100 g i standardna devijacija manja od 10 g, program prelazi na sljedeći korak, računanje željene težine hrane, u suprotnom program konstantno računa ta dva parametra dok uvjet ne bude ispunjen. Zatim program računa kolika bi trebala biti željena težina hrane koju treba dati. Nakon izračunavanja željene količine hrane, program otvara pregradu spremnika i dozvoljava da se hrana počne izbacivati. Varijabla „KriticnaRadnja“ se postavlja na „*TRUE*“. Ova varijabla nam služi za signalizaciju da je u toku ključna radnja izdavanja hrane te da se program treba fokusirati na ispravno mjerenje i kontrolu tog procesa. „KriticnaRadnja“ se koristi zbog toga što mjerenje temperature i slanje podataka na blynk traje dugo, ali samo pokretanje peltiera je samo „uključi“ ili „isključi“ i traje jako kratko. Dok traje kritična radnja on ima vremena pokretat peltiere, ali nema vremena očitavati temperaturu i prilagoditi snagu. Nakon toga slijedi mjerenje težine izbačene hrane, a zatim provjera težine izbačene hrane. Ako je izbačena težina manja od 70% željene težine program nastavlja pratiti težinu hrane, dok ne dosegne ciljanu količinu. Ako je izbačena težina veća ili jednaka 70% željene težine prelazi se na zatvaranje pregrade spremnika za hranu. Kada je isporučeno dovoljno hrane, pregrada se zatvara i u sljedećem koraku se varijabla „KriticnaRadnja“ postavlja na „*FALSE*“. Sljedeći korak je mjerenje težine izbačene hrane gdje se mjeri konačna težina izbačene hrane, kako bi se potvrdilo da je ispravno izdana.

Posljednji blok predstavlja čekanje da se obje vage ponovno postave na nulu, što znači da program čeka dok se senzori vage ne stabiliziraju, odnosno dok obje vage ne pokažu nulu. Ovaj korak osigurava da je hranjenje završeno i da su sve vrijednosti postavljene u početno stanje za sljedeći ciklus.



Slika 3.24. *Blokovski prikaz upravljanja hranilice*

Slika 3.25. prikazuje blok dijagram upravljanja temperaturom vode i slanje podataka na platformu blynk. Blok kritična radnja predstavlja uvjetni blok, koji odlučuje hoće li se izvršiti mjerenje temperature vode i kontrola peltier elemenata. Ulazni argument ovdje je signal sa slike 3.24 „KriticnaRadnja“. Ako je uvjet „Ne“, tada se upravljanje peltier elementima i slanje podataka na blynk-u preskače i program se vraća na početak. Ako je uvjet „Da“, prelazi se na sljedeću operaciju, mjerenje temperature. Za blok mjerenje temperature ulazni argument je podatak sa senzora temperature, a izlazni argument je izmjerena temperature vode, potrebna za izračunavanje snage za peltier elemente. Sljedeći blok je računanje potrebne snage na peltier elementima koji služi za izračunavanje potrebne snage da peltier elementi osiguraju zagrijavanje ili hlađenje vode. Ulazni argument je izmjerena temperatura, a izlazni argument je izračunata snaga. Nakon toga slijedi blok za slanje podataka na *blynk*. Ulazni argument su podaci o trenutnoj temperaturi i izračunatoj snazi, a izlazni argument je potvrda da su podaci uspješno poslani. Blok s nazivom „Je li razina vode iznad minimalne“ je uvjetni blok koji provjerava razinu vode. Ulazni argument je mjerenje razine vode dobivene od senzora razine. Izlazni argument ima dva smjera, ako je razina vode ispod minimalne, program će isključiti peltier elemente, a ako je razina iznad minimalne, program nastavlja s pokretanjem peltier elementa. Nakon što su peltier elementi uključeni ili isključeni, program se vraća na početak i ponovno provjerava uvjete za kritičnu radnju.



Slika 3.25. *Blokovski prikaz upravljanja temperaturom vode i slanje podataka na Blynk*

Informacije se na *Blynk* prosljeđuju preko prethodno određenih *Blynk* virutalnih pinova. Na *blynk*-u je određeno 18 virtualnih pinova prikazanih u tablici 3.1.

Tablica 3.1. Pinovi korišteni za komunikaciju preko Blynk-a

Identifikcijski broj Blynk virutalnog pin-a	Naziv pina	Podatkovni tip	Minimalna vrijednost	Maksimalna vrijednost	Mjerna jedinica
0	*Current temperature*	*double*	-50	150	°C
1	*Unit 1 Animal weight*	*integer*	0	5000	g
2	*Unit 1 Wanted food weight*	*integer*	0	1000	g
3	*Unit 1 Measured food weight*	*integer*	0	1000	g
4	*Current power*	*double*	0	100	%
5	*Unit 2 Animal weight*	*integer*	0	5000	g
6	*Unit 2 Wanted food weight*	*integer*	0	1000	g
7	*Unit 2 Measured food weight*	*integer*	0	1000	g
8	*Heating OR Cooling*	*integer*	-1	1	N/A
9	*Unit 3 Animal weight*	*integer*	0	5000	g
10	*Unit 3 Wanted food weight*	*integer*	0	1000	g
11	*Unit 3 Measured food weight*	*integer*	0	1000	g
13	*Unit 4 Animal weight*	*integer*	0	5000	g

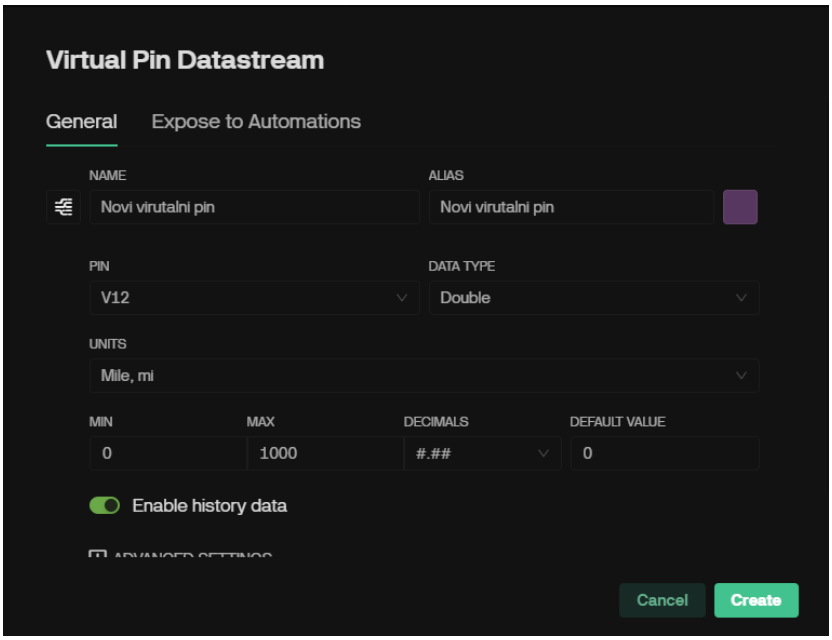
14	*Unit 4 Wanted food weight*	*integer*	0	1000	g
15	*Unit 4 Measured food weight*	*integer*	0	1000	g
17	*Unit 5 Animal weight*	*integer*	0	5000	g
18	*Unit 5 Wanted food weight*	*integer*	0	1000	g
19	*Unit 5 Measured food weight*	*integer*	0	1000	g

Svaka stanica ima 3 pina koji prikazuju rezultate sa posljednjeg ciklusa stanice. Rezultati pojedinačne stanice su izmjerena težina životinje, željena težina hrane i izmjerena težina hrane. Izmjerena težina životinje je prosječno mjerenje sa životinjske vage dane stanice. Željena težina hrane se računa kao 10% iznosa težine životinje. Po potrebi se taj izračun može prilagoditi. Izmjerena težina hrane je prosječno mjerenje sa vage za hranu dane stanice.

Težina životinje je parametar koji ne bi trebao biti veći od 5 kg za ovu maketu, stoga je vrijednost ograničena na 5000 g, a očekivana težina hrane i mjerena težina hrane je ograničena na 1000 g.

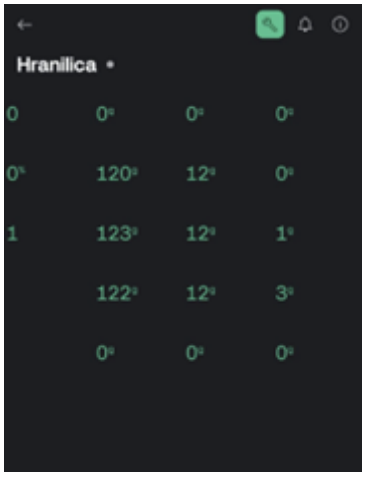
Preostala 3 pina predstavljaju kontrolu temperature vode, koji predstavljaju trenutnu snagu grijanja u odnosnu na maksimalnu moguću snagu peltier elementa, temperatura vode, te radi li se o grijanju ili hlađenju.

Novi virtualni pinovi se dodaju na *Blynk* preko *web*-sučelja. Potrebno je odrediti identifikacijski broj virtualnog pina, naziv pina, podatkovni tip pina (*integer, double ili string*), početnu vrijednost virtualnog pina, te ako je odabran brojčani tip podatka potrebno je odrediti minimalnu i maksimalnu vrijednost pina (*slika* 3.26.).



Slika 3.26. Dodavanje novog virtualnog pina.

Nakon što se odrede svi virtualni pinovi potrebno je odrediti predložak u koji se mogu dodati svi virtualni pinovi te odrediti kako će izgledati prikaz svih tih grafičkih elemenata na *web* i mobilnoj verziji aplikacije. Izgled mobilne aplikacije možemo vidjeti na slici 3.27.



Slika 3.27. *Mobilna aplikacija.*

Uređaj se autentificira na *Blynk* platformi pomoću privatnih ključeva pohranjenih u *privateData.h* datoteci. Ti ključevi su identifikacijski broj predloška, ime predloška te autentifikacijski token.

4. TESTIRANJE I REZULTATI

4.1. Metodologija testiranja

Nakon što je uređaj izrađen i proveden programski kod, izvedeno je testiranje kako bi se utvrdilo radi li sustav ispravno. Svaka komponenta sustava testirana je pojedinačno prilikom izrade rada, kako bi se osigurala njihova ispravnost i točnost u radu. Pumpa za vodu testirana je u kombinaciji sa senzorima razine. Pumpa se automatski uključuje kada senzor razine signalizira nisku razinu tekućine i isključuje se kada senzor gornje razine signalizira dostizanje maksimalne razine. Pumpa je napunila posudu s vodom za 47,66 s. Kada se postigla minimalna količina vode, koja se očitava senzorom razine pričvršćenim za dno posude, peltierov element kreće s radom ovisno o temperaturi. Testiran je u kombinaciji s temperaturnim senzorom kako bi se procijenila njegova učinkovitost u hlađenju i grijanu. Testiranje je odrađeno bez vode u posudi, iz sigurnosnih razloga. Kako bi se provjerila funkcionalnost peltierova elementa, u kodu je postavljeno uključivanje grijanja pri temperaturi ispod 35˚C, a hlađenje iznad 35˚C. Za testiranje zagrijavanja peltierova elementa je dovoljna sobna temperatura. Kada bi se ručno aktivirao senzor razine, senzor bi odmah pokazao sobnu temperaturu (oko 22 ˚C) i odmah bi se uključio peltierov element koji je zagrijavao posudu. Nakon nekog vremena na dnu posude osjetila bi se promjena temperature, posuda bi se zagrijala. Prilikom testiranja hlađenja, korišten je *upaljač* kojim se lagano zagrijavao vrh temperaturnog senzora (pazeći da ga se ne ošteti), te je potrebno pratiti trenutna temperatura pomoću aplikacije. Kada bi temperatura dostigla 40˚C, potrebno ju je održavati neko vrijeme iznad te granice, kako bi se osjetila promjena temperature dna posude, tj. da peltierov element posudu sve više hladi. Nekoliko sekundi nakon aktiviranja peltierova elementa koji hladi posudu, uključio bi se i ventilator koji hladi drugu stranu peltierovog elementa.

Testiranje je primarno provedeno na *load-cell* senzorima postavljenim u svakoj stanici, pri čemu su korištena dva senzora - jedan za mjerenje težine životinje, a drugi za mjerenje težine hrane. Svaka stanica testiran je nekoliko puta na način da se predmet poznate težine postavi na *load cell* predviđen za vaganje životinje. Kada bi se otvorila vratašca pomoću *stepper* motora, dosipavali bi hranu kroz otvor na vrhu plastične posude, predviđene za spremanje hrane, sve dok se vratašca ponovno ne zatvore. Hrana bi kroz otvor padala u plastičnu čašu koja se nalazila na drugom *load cell*-u predviđenom za vaganje hrane. Vrijednost koju životinja treba dobiti, računa se tako da se masa životinje podjeli s 10. Tijekom testiranja, svaki *load-cell* senzor bio je podvrgnut nizu

mjerenja kako bi se provjerila preciznost očitanja. Senzori su bili kalibrirani korištenjem poznatih referentnih težina, što je omogućilo usporedbu očitanih vrijednosti s očekivanim rezultatima. Predmet koji je korišten kao referentna težina tijekom testiranja prikazan je na slici 4.1.

Slika 4.1. *Referentni predmet*

Posljednji element koji se testirao su bili *PIR* senzori. Njih se testiralo na način da ih se u potpunosti prekrije, a zatim postavi predmet na *load cell* senzor za vaganje životinja. Nakon nekog vremena, može se primijetiti kako se vratašca ne otvaraju i ukloniti predmet sa senzora, te ponoviti postupak još nekoliko puta.

4.2. Rezultati testiranja

Kako bi se osigurala vjerodostojnost rezultata, testiranje je provedeno na način da se za svaku stanicu mjerenja ponove 20 puta pod što sličnijim uvjetima. Nakon svakog ciklusa mjerenja, izračunata je srednja vrijednost mjerenih veličina te je zabilježeno mjerno odstupanje. Rezultati su prikazani u tablicama, te grafički u obliku dijagrama. Srednja vrijednost se računa prema formuli:

$$\bar{x} = \frac{\sum xi}{n} \tag{4-1}$$

gdje je $\bar{x}$ srednja vrijednost mjerenih podataka, n ukupan broj mjerenja, a xi vrijednost pojedinačnog mjerenja.

Standardna devijacija, odnosno mjerno odstupanje računa se prema formuli:

$$\sigma = \sqrt{\frac{1}{n}\sum(x_i - \bar{x})^2} \tag{4-2}$$

gdje je σ mjerno odstupanje, n ukupan broj mjerenja, *xi* vrijednost pojedinačnog mjerenja, a $\bar{x}$ srednja vrijednost.

Tablica 4.1. *Testiranje load-cell senzora za stanicu-1.*

Broj mjerenja	*Load-cell* za životinje [g]	Izračunata hrana [g]	*Load-cell* za hranu [g]
1.	102	10	6
2.	108	10	8
3.	105	10	7
4.	103	10	7
5.	111	11	8
6.	102	10	8
7.	101	10	7
8.	111	11	7
9.	108	10	7
10.	101	10	7
11.	104	10	8
12.	103	10	7
13.	109	10	7
14.	111	11	7
15.	101	10	7
16.	107	10	7
17.	104	10	6
18.	105	10	7
19.	111	10	7
20.	113	10	8
Srednja vrijednost	105,3	10,15	7,15
Mjerno odstupanje	4,03	0,37	0,65

Tablica 4.2. *Testiranje load-cell senzora za stanicu-2.*

Broj mjerenja	*Load-cell* za životinje [g]	Izračunata hrana [g]	*Load-cell* za hranu [g]
1.	120	12	8
2.	121	12	10
3.	120	12	12
4.	120	12	9
5.	120	12	8
6.	119	11	10
7.	120	12	9
8.	120	12	9
9.	120	12	8
10.	118	11	10
11.	121	11	9
12.	120	11	10
13.	120	12	8
14.	121	12	9
15.	120	12	10
16.	120	12	10
17.	119	12	9
18.	118	12	12
19.	120	12	8
20.	119	11	9
Srednja vrijednost	119,8	11,75	9,35
Mjerno odstupanje	0,81	0,43	1,15

Tablica 4.3. *Testiranje load-cell senzora za stanicu -3.*

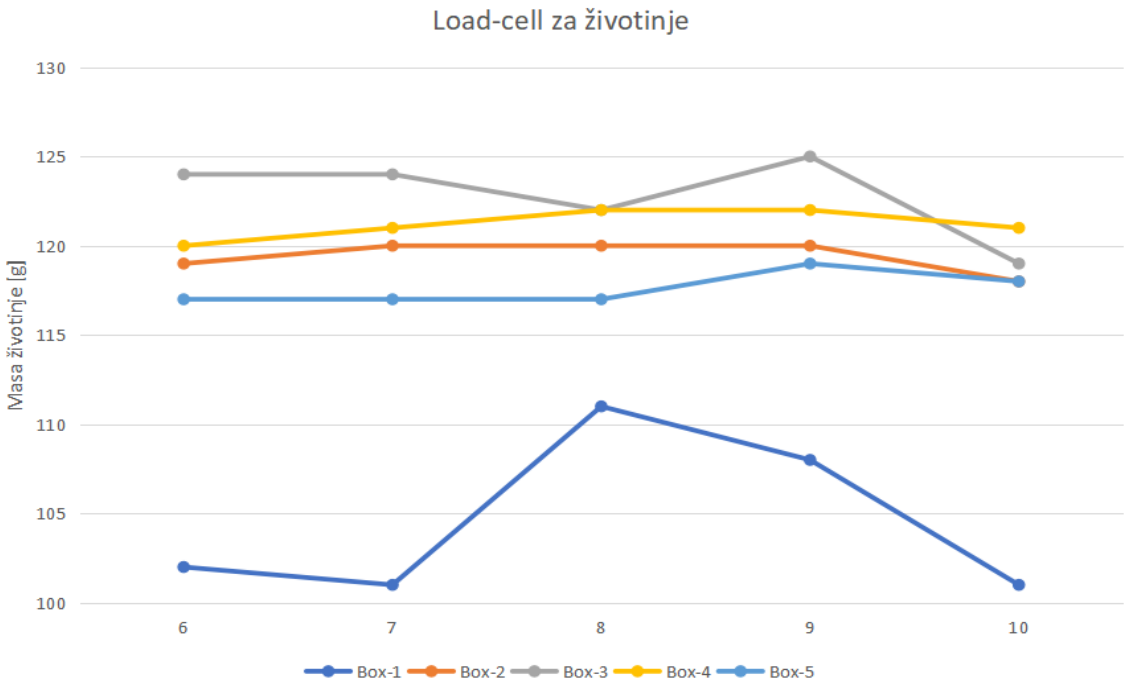
Broj mjerenja	*Load-cell* za životinje [g]	Izračunata hrana [g]	*Load-cell* za hranu [g]
1.	124	12	9
2.	124	12	10
3.	124	12	9
4.	122	12	9
5.	120	12	9
6.	124	12	9
7.	124	12	9
8.	122	12	8
9.	125	12	8
10.	119	11	10
11.	120	12	9
12.	123	12	9
13.	120	12	9
14.	122	11	9
15.	125	12	10
16.	124	12	9
17.	124	12	9
18.	124	12	9
19.	122	12	8
20.	119	11	8
Srednja vrijednost	122,55	11,85	8,95
Mjerno odstupanje	1,93	0,35	0,59

Tablica 4.4. *Testiranje load-cell senzora za stanicu -4.*

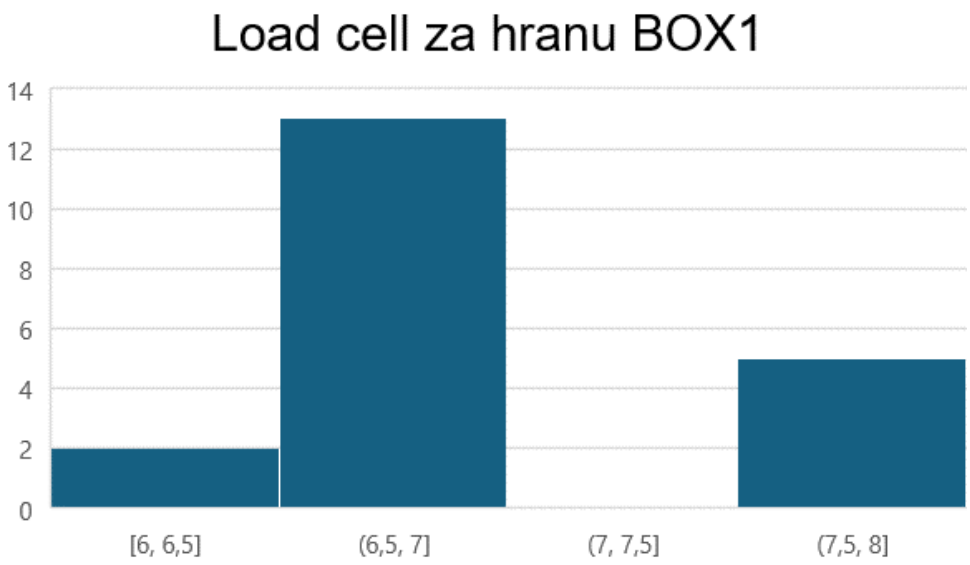
Broj mjerenja	*Load-cell* za životinje [g]	Izračunata hrana [g]	*Load-cell* za hranu [g]
1.	121	12	10
2.	123	12	10
3.	121	12	9
4.	121	12	9
5.	124	12	10
6.	120	12	10
7.	121	12	9
8.	122	12	8
9.	122	12	10
10.	121	12	11
11.	123	12	11
12.	122	12	10
13.	121	12	9
14.	121	12	9
15.	121	11	9
16.	123	12	11
17.	124	12	10
18.	122	12	9
19.	120	12	9
20.	121	12	8
Srednja vrijednost	121,5	11.95	9,55
Mjerno odstupanje	1,16	0.2	0,86

Tablica 4.5. *Testiranje load-cell senzora za stanicu -5.*

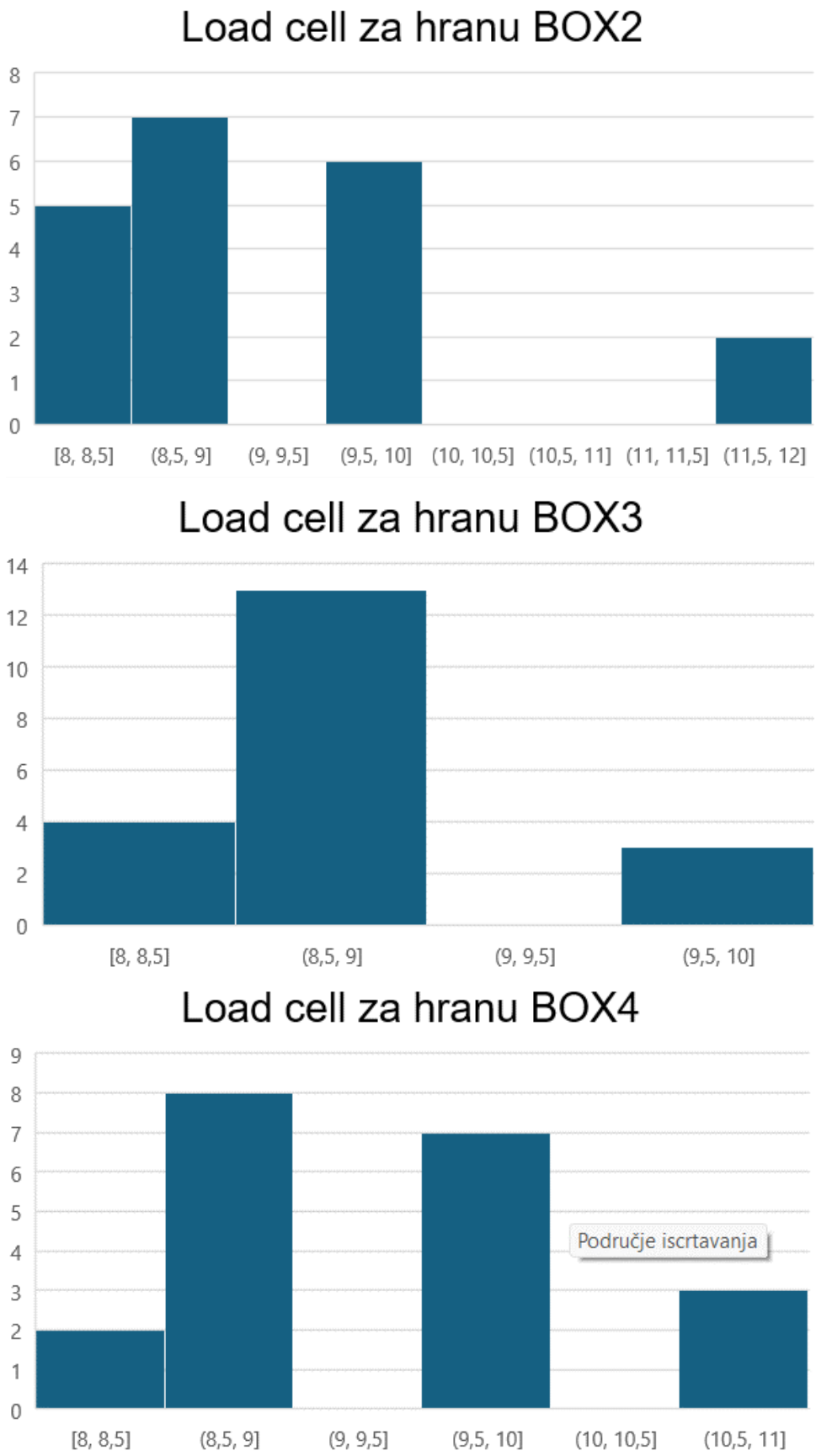
Broj mjerenja	*Load-cell* za životinje [g]	Izračunata hrana [g]	*Load-cell* za hranu [g]
1.	117	11	8
2.	118	11	7
3.	118	11	8
4.	116	11	8
5.	118	11	9
6.	117	11	7
7.	117	11	10
8.	117	11	9
9.	119	11	9
10.	118	11	8
11.	117	11	9
12.	117	11	8
13.	116	11	8
14.	117	11	9
15.	118	11	8
16.	118	11	7
17.	118	11	8
18.	117	11	10
19.	117	11	10
20.	119	11	8
Srednja vrijednost	117,45	11	8,4
Mjerno odstupanje	0,6	0	0,91



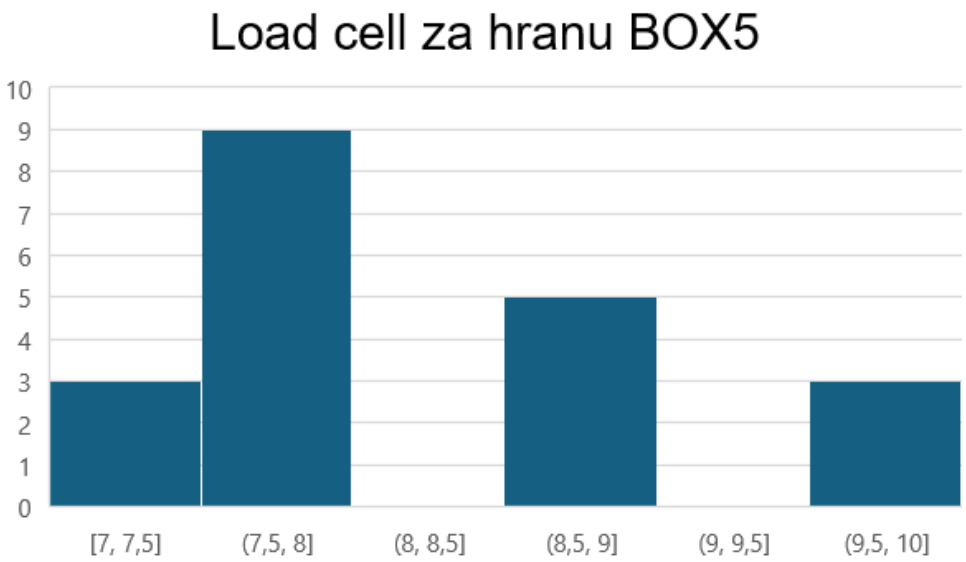
Slika 4.2. *Grafički prikaz rezultata mjerenja load-cell senzora za životinje.*



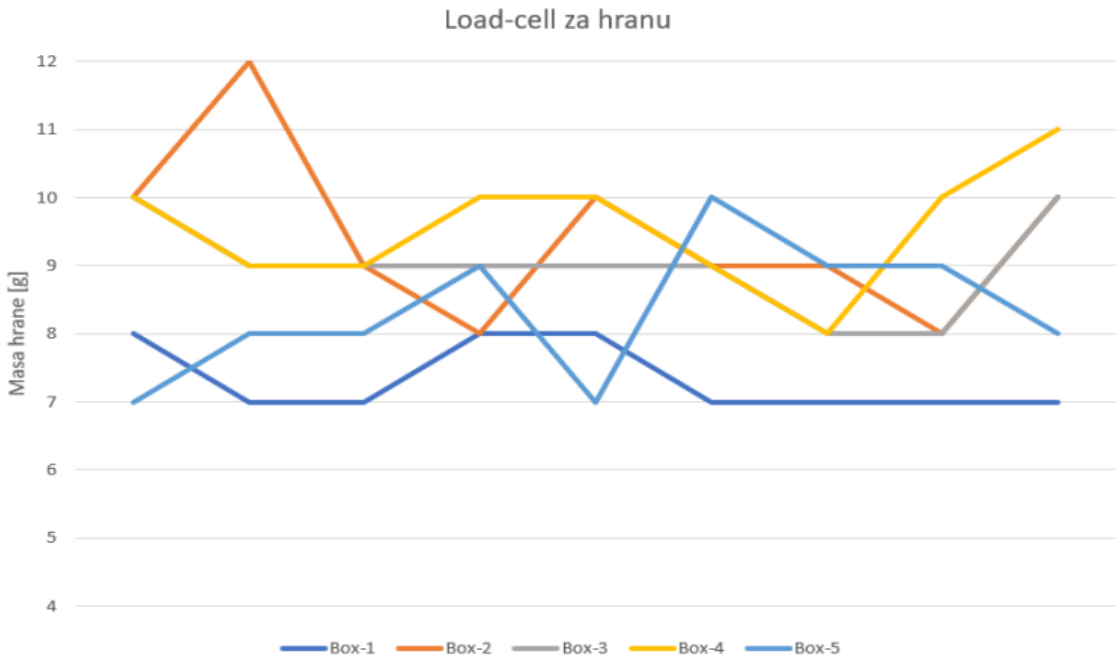
Slika 4.3. *Prikaz rezultata mjerenja load-cell senzora za životinje za stanicu 1 pomoću histograma*



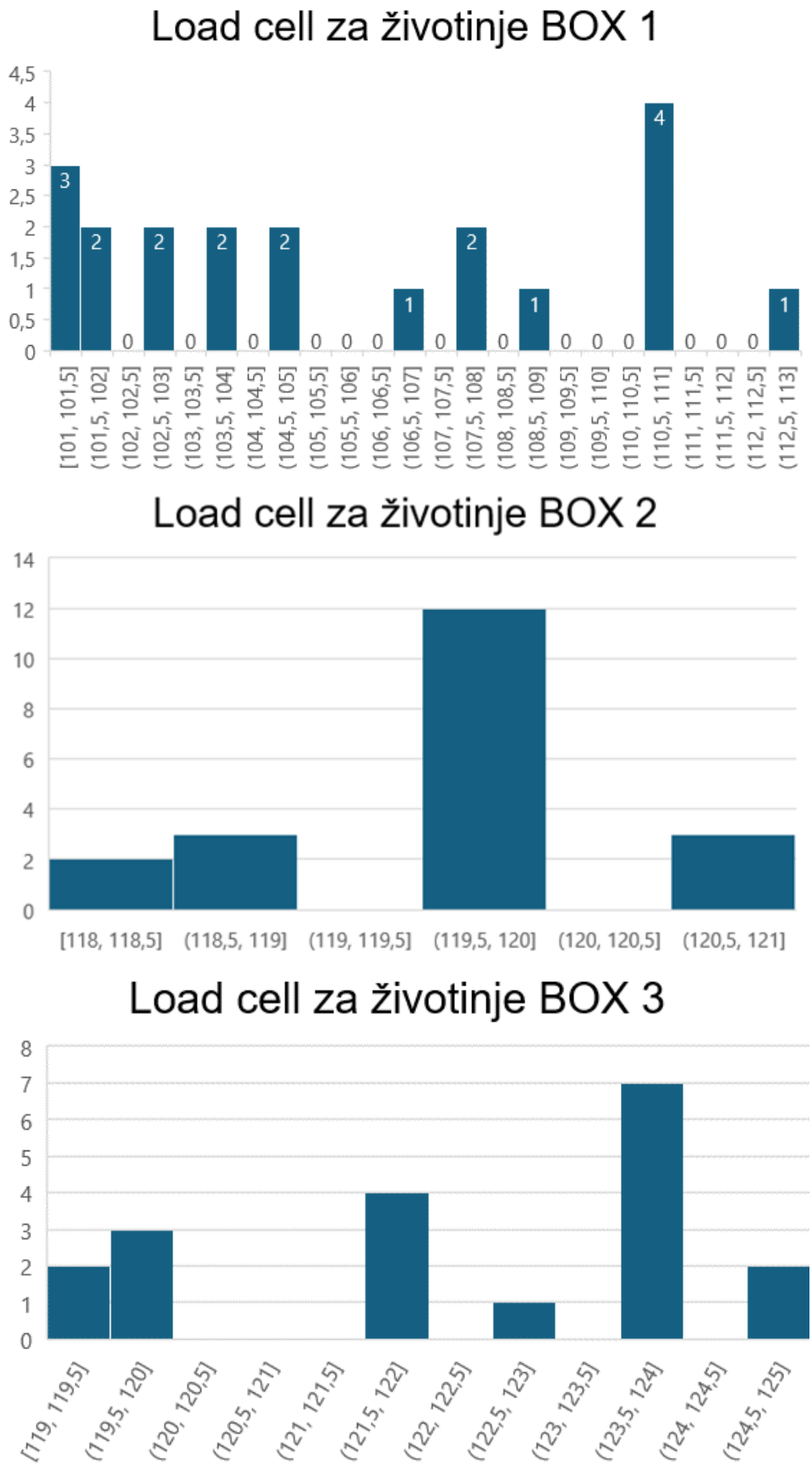
Slika 4.4. *Prikaz rezultata mjerenja load-cell senzora za životinje za stanicu 2, 3 i 4, pomoću histograma*



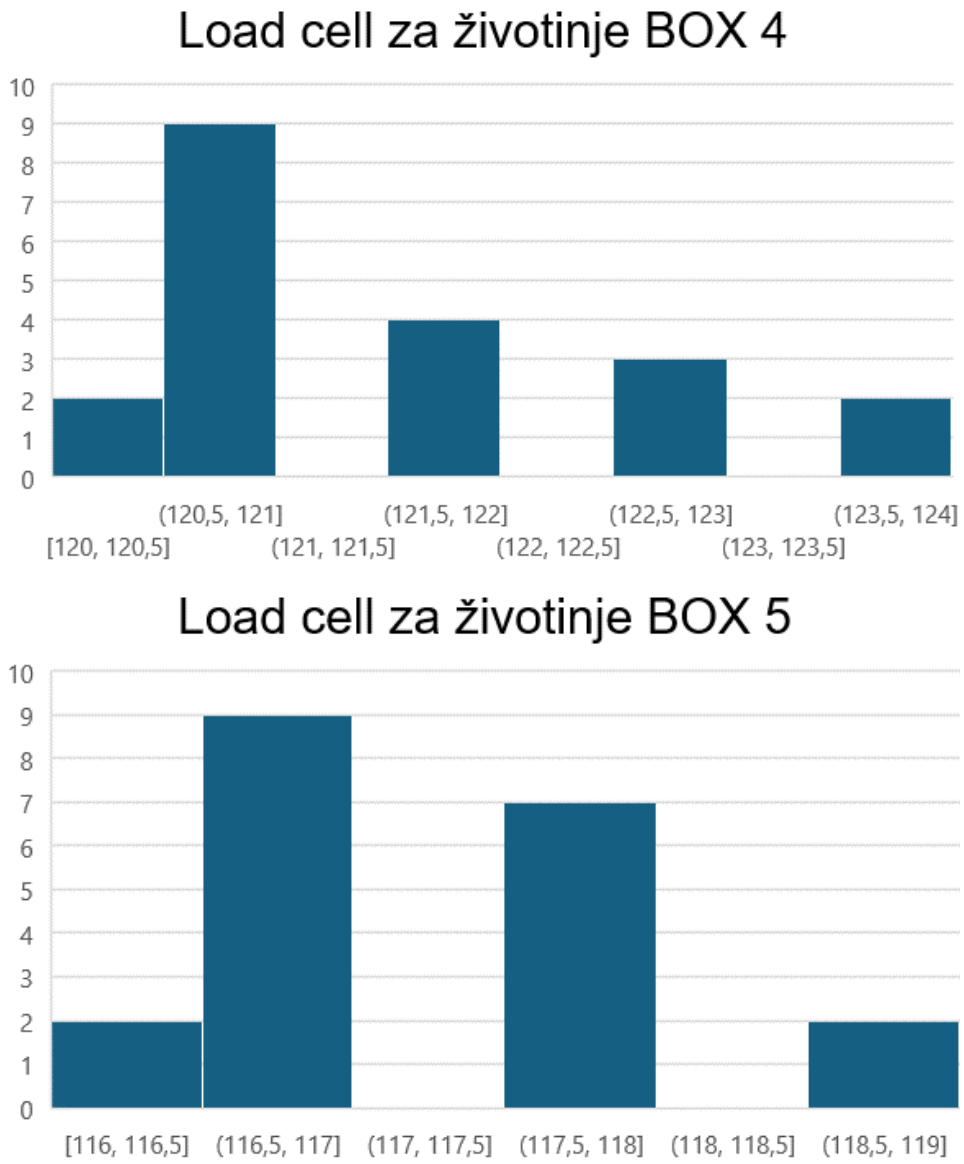
Slika 4.5. *Prikaz rezultata mjerenja load-cell senzora za životinje za stanicu 5, pomoću histograma*



Slika 4.6. *Grafički prikaz rezultata mjerenja load-cell senzora za hranu.*



Slika 4.7. *Prikaz rezultata mjerenja load-cell senzora za hranu za stanicu 1, 2 i 3, pomoću histograma*



Slika 4.8. *Prikaz rezultata mjerenja load-cell senzora za životinje za stanicu 4 i 5, pomoću histograma*

Rezultati mjerenja su prikazani tablicama 4.1. - 4.5, te pomoću histograma sa slika 4.3-4.5 i 4.7-4.8 . Slika 4.2. prikazuje rezultate mjerenja za svih pet *load-cell* senzora postavljenih u stanicama za mjerenje težine životinja. Svi *load-cell* senzori za životinje pokazuju relativno ujednačene rezultate s minimalnim odstupanjima između ponovljenih mjerenja. Najveća srednja vrijednost je zabilježena za stanicu-3, s prosjekom od 122,8 g, dok je najmanja srednja vrijednost zabilježena u stanici-1, s prosjekom od 105,2 g. Upravo je u stanici-1 zabilježeno najveće mjerno odstupanje od 3,8 g, što pokazuje da su u toj stanici mjerenja bila nešto varijabilnija i da kalibracija *load-cell* senzora nije najbolje obavljena.

Na slici 4.6. prikazani su rezultati mjerenja za *load-cell* senzore za hranu, za svaku od testiranih stanica. Vrijednosti mjerenja su većinom postojane i bliske izračunatoj količini hrane. U većini slučajeva, odstupanja su minimalna. Ipak, u stanici-2 primijećeno je nešto veće mjerno odstupanje (1,18 g), što i dalje predstavlja prihvatljivo odstupanje.

5. ZAKLJUČAK

Cilj ovoga rada je automatizirati postupak hranjenja i pojenja životinja. Glavni pokretač ovoga rada je *ESP32* mikroupravljač i *Arduino* razvojno okruženje. Osim toga za realizaciju su korišteni *load-cell* senzori, *PIR* senzori, *stepper* motori, pumpa za vodu, senzori razine, senzor temperature i Peltier pločice. Konstrukcija je sastavljena od OSB ploče i boca koje predstavljaju spremnike za hranu, te spremnike za vodu. Složena maketa u konačnici radi bez ikakvih problema, no prilikom izrade uočeni su neki problemi.

Prvi problem koji se pojavio je spajanje 5 *stepper* motora od kojih svaki ima 4 pina za spajanje, koji je riješen preko *shift* registara. Korištena su tri *shift* registra koji pamte podatke serijski, a predaju ih paralelno. Osim toga problem je predstavljala pumpa za vodu. Pumpa kao istosmjerni *DC* motor predstavljala je smetnju mikrokontroleru i svim ostalim komponentama. Ako motor ima četkice, gotovo sigurno će generirati male iskrice ili lukove između četkica i segmenata komutatora. Lukovi su glavni izvor radiofrekvencijskih smetnji (RFI). Te smetnje mogu biti prenesene žicama motora te doći do osjetljive elektronike unutar kontrolera. Ako su smetnje dovoljne jačine, mogu uzrokovati blokadu mosfeta, što će kratko spojiti napajanje i dovesti do eksplozije *mosfeta*. Prvo rješenje za rješavanje ovog problema je ugradnja kondenzatora za suzbijanje smetnji, te njegovo postavljanje što je moguće bliže četkicama motora. Drugo rješenje koje je korišteno je uvrtanje žica motora koje sprječava da se smetnje emitiraju u zrak.

Ovaj rad se može poboljšati povećavanjem preciznosti *load-cell* senzora na način da se oni pričvrste za policu na kojoj se nalaze, te preciznijom izvedbom konstrukcijskog dijela. Rad se također može nadograditi dodavanjem senzora u spremnike sa hranom i vodom kako bi uvijek imali informaciju o stanju unutar spremnika. Osim toga može se dodati i kamera za nadzor svake stanice.

LITERATURA

[1] „Lønholm Agro“ Denmark https://www.delaval.com/en-gb/learn/news/the-worlds-largest-delaval-vms-batch-milking-farm/, pristup: 19.9.2024.

[2] Farma „Topolik“ https://www.belje.hr/otvorena-najveca-robotizirana-farma-muznih-krava-u-europi/, pristup: 19.9.2024.

[3] Automatski sistemi za hranjenje https://www.mdpi.com/2076-2615/13/21/3382, pristup 19.9.2024.

[4] ESP-WROOM-32 Development Board, https://www.amazon.com/ESP-WROOM-32-Development-Microcontroller-Integrated-Compatible/dp/B08D5ZD528?th=1,pristup 20.08.2024.

[5] 28BYJ-48 - 5V Stepper Motor, https://components101.com/motors/28byj-48-stepper-motor, pristup: 21.08.2024.

[6] Stepper motor s driverom, https://soldered.com/hr/proizvod/stepper-motor-s-driverom/, pristup: 21.08.2024.

[7] *PIR* motion sensor, https://hub360.com.ng/product/pir-motion-sensor/, pristup: 21.08.2024.

[8] PIR motion sensor, https://hub360.com.ng/product/pir-motion-sensor/, pristup: 21.08.2024.

[9] Strain Gauge Load Cell - 4 Wires - 5Kg, https://www.adafruit.com/product/4541, pristup: 21.08.2024.

[10] Load Cell Weight Sensor 5Kg, https://www.twinschip.com/Load_Cell_Weight_Sensor_5Kg, pristup: 21.08.2024.

[11] Adafruit HX711 - 24-Bit Analog-to-Digital Converter, https://learn.adafruit.com/adafruit-hx711-24-bit-adc, pristup: 21.08.2024.

[12] Load-cell amplifier HX711 board, https://soldered.com/product/load-cell-amplifier-hx711-board/, pristup: 21.08.2024.

[13] Water Level Sensor Float Switch - P43 for Industrial, https://www.indiamart.com/proddetail/water-level-sensor-float-switch-p43-22215039748.html, pristup: 26.08.2024

[14] White Wired Liquid Water Level Sensor Float Switch for Aquarium, https://www.amazon.in/DS-Robotics%C2%AE-Sensor-Liquid-Aquarium/dp/B0D45T4HN8, pristup: 26.08.2024.

[15] DS18B20 Datasheet, https://www.analog.com/media/en/technical-documentation/data-sheets/ds18b20.pdf, pristup: 26.08.2024.

[16] Waterproof DS18B20-compatible Temperature Sensor, https://shop.pimoroni.com/products/ds18b20-programmable-resolution-1-wire-digital-thermometer?variant=32127344640083, pristup: 26.08.2024.

[17] What is a Thermo-electric module (Peltier element)?, https://ft-mt.co.jp/en/product/electronic_device/thermo/, pristup: 26.08.2024.

[18] TEC1-12705 Thermoelectric Cooler Heat Sink Cooling, https://www.amazon.com/uxcell-TEC1-12705-Thermoelectric-Cooling-Peltier/dp/B07MTYSSCS, pristup: 26.08.2024.

[19] KKM: MINI PUMPA ZA VODU, https://soldered.com/hr/learn/kkm-mini-pumpa-za-vodu/, pristup: 09.09.2024.

[20] KKM: MINI PUMPA ZA VODU, https://soldered.com/hr/learn/kkm-mini-pumpa-za-vodu/, pristup: 09.09.2024.

[21] About KiCad, https://codeinstitute.net/global/blog/what-is-vs-code/, pristup 11.9.2024.

[22] ESP32-WROOM-32 Datasheet, https://www.espressif.com/sites/default/files/documentation/ esp32-wroom-32_datasheet_en.pdf, pristup: 20.08.2024.

POPIS OZNAKA I KRATICA

PIR – Passive Infraread

PCB – Printed Circuit Bord

IoT – Internet of Things

ATX – Advanced Technology Extended

AC – Alternating Current

DC – Direct Current

OSB – oriented Strand Bord

CPU – Central Processing Unit

SPS – Samples Per Second

SD – Secure Digital

SPI – Serial Peripheral Interface

UART – Universal Asynchronous Reciever/Transmitter

I2S – Integrated Interchip Sound

I2C – Inter-Integrated Circuit

SAŽETAK

Automatska hranilica koja omogućava precizno doziranje hrane na temelju prisutnosti i težine životinje, uz automatsko punjenje vodom i regulaciju njezine temperature, tema su ovoga završnog rada. Sustav je sastavljen od raznih senzora, uključujući *PIR* senzori za detekciju pokreta, *load-cell* senzori za mjerenje težine životinje i hrane, temperaturni senzor za kontrolu temperature vode, te senzori razine za nadzor razine vode. Rad je realiziran korištenjem *ESP32* mikroupravljača i Arduino razvojnog okruženja. Prilikom kreiranja makete pojavile su se neki problemi, poput smetnji uzrokovanih pumpom, koji su uspješno otklonjeni. Automatska hranilica i pojilica ključne su komponente suvremenih farmi, jer ne samo da smanjuju potrebu za fizički radom, nego značajno doprinose optimizaciji resursa i povećanju produktivnosti u stočarstvu. Ovaj sustav također osigurava bolju brigu za životinje, posebno u zahtjevnim uvjetima, čime se osigurava stabilan prinos i održivost.

Ključne riječi: Arduino IDE, Automatska hranilica, automatska pojilica, *ESP32* mikroupravljač

ABSTRACT

The automatic feeder, which enables precise food dispensing based on the presence and weight of the animal, along with automatic water replenishment and temperature regulation, is the focus of this final project. The system consists of various sensors, including *PIR* sensors for motion detection, *load-cell* sensors for measuring the weight of animals and food, a temperature sensor for water temperature control, and level sensors for monitoring water levels. The project was realized using the *ESP32* microcontroller and Arduino development environment. During the creation of the prototype, some issues, such as interference caused by the water pump were encountered, but successfully resolved. The automatic feeder and water dispenser are essential components of modern farms, as they not only reduce the need for manual labor but also significantly contribute to resource optimization and increased productivity in livestock farming. This system also ensures better care for animals, particularly in demanding conditions, thereby securing stable yields and sustainability.

Keywords: Arduino IDE, Automatic feeder, Automatic water dispenser, *ESP32* microcontroller

PRILOZI I DODACI

P1. Hardverske komponente

Esp-32

Prema [22], ploča sadrži dvije CPU (engl. *Central Processing Unit*) jezgre koje se mogu zasebno kontrolirati, a frekvencija CPU-a može se podesiti od 80 MHz do 240 MHz. Čip također uključuje niskopotrošni koprocesor koji se može koristiti umjesto CPU-a kako bi se uštedjela energija prilikom izvršavanja zadataka koji ne zahtijevaju mnogo računalne snage, poput nadzornih perifernih uređaja. *ESP32* integrira bogat skup perifernih uređaja, uključujući kapacitivne dodirne senzore, SD kartično sučelje, Ethernet, brzi SPI, UART, I2S i I2C. Struja u stanju mirovanja *ESP32* čipa manja je od 5 μA, što ga čini pogodnim za primjene u uređajima koji se napajaju iz baterije i nosivim elektroničkim uređajima. Modul podržava brzinu prijenosa podataka od 150 Mbps i izlaznu snagu od 20 dBm na anteni, čime se osigurava maksimalni fizički domet.

Koračni motor 28BYJ-48 s driverom

Napajanje: Motor radi pri naponu od 5 V DC, što ga čini kompatibilnim s većinom osnovnih mikroupravljača.

Kut koraka: Kut koraka iznosi 5.625° po koraku, što znači da motor obavi 64 koraka za potpunu rotaciju, omogućujući visoku preciznost u kretanju.

Moment držanja: Ovaj moment je veći od 34.3 mNm, što označava maksimalnu snagu koju motor može održati kada je potpuno zaustavljen i u stabilnom položaju.

Broj faza: Motor ima 4 faze, što omogućuje precizno upravljanje rotacijom

Upravljanje: Za upravljanje ovim motor preporučuje se korištenje *drivera*, koji omogućuje preciznu kontrolu rotacije. Upravljački signal treba se provoditi prema specifičnom redoslijedu aktivacije namota kako bi motor rotirao u željenom smjeru.

Struja drivera: *Driver* može isporučiti do 800 mA struje.

Napon drivera: *Driver* može raditi unutar naponskog raspona od 3,3 V do 14 V

PIR senzor

Napajanje: *PIR* senzor pokreta u ovom radu, radi u rasponu napona od 2.7 do 12 V.

Kašnjenje: Senzor ima zadano vrijeme kašnjenja (engl. *Delay*) od približno 2 sekunde nakon detekcije pokreta, što znači da će signal za aktivaciju biti poslan tek nakon ovog kratkog intervala.

Detekcija pokreta: Senzor može detektirati pokrete na udaljenosti od 3 do 5 metara. To znači da senzor ima sposobnost da prepozna kretanje unutar ovog raspona udaljenosti

Detekcijski kut: Senzor je dizajniran za detekciju pokreta unutar kuta od 100°. Ovo omogućuje senzoru da pokrije relativno široko područje.

HX711 breakout

Pojačanje: Kanal A nudi pojačanje od 128 puta, dok kanal B nudi pojačanje od 64 puta, što omogućuje prilagodbu osjetljivosti prema potrebama senzora i primjeni.

Brzina očitanja: Uređaj podržava dvije brzine očitanja, 10 uzoraka u sekundi (engl. *Samples per second* - SPS) ili 80 SPS, omogućujući izbor između bržeg uzorkovanja ili dužeg vremena stabilizacije za točnija mjerenja.

Senzor temperature DS18B20

Raspon mjerenja: DS18B20 digitalni termometar može mjeriti temperaturu u širokom rasponu od -55°C do +125°C.

Pogreška mjerenja: Senzor nudi pogrešku mjerenja od $\pm$0.5°C unutar radnog raspona od -10°C do +85°C.

Rezolucija mjerenja: Senzor omogućuje usklađivanje rezolucija između 9 i 12 bita, što omogućava prilagodbu između točnosti i brzine očitanja temperature.

Pumpa za vodu

Napon: Ova pumpa radi na niskom naponu od 3 do 6V.

Maksimalna visina na koju može doći tekućina: Pumpa može podići vodu na visinu između 40 i 110 cm.

Brzina protoka: Maksimalni protok vode koji pumpa može postići iznosi 120 litara po satu.

P2. Program

FullModel

FullModel.cpp

```
#include "fullModel.h"

#include "../Setup/BlynkPinConfiguration.h"

#include <WiFi.h>
#include <WiFiClient.h>
#include <BlynkSimpleEsp32.h>
#define BLYNK_PRINT Serial
// Konstruktor klase FullModel, prima pokazivače na WaterContainer i
MultipleLoadCells objekte, te broj jedinica (n).
FullModel::FullModel(
    WaterContainer *mWaterContainer,
    MultipleLoadCells *mMultipleLoadCells,
    int n,
    ...
):
    waterContainer(mWaterContainer),
    // Inicijalizira pokazivač na spremnik vode.
    multipleLoadCells(mMultipleLoadCells),
    // Inicijalizira pokazivač na senzore težine.
    numberOfUnits(n)     // Sprema broj jedinica.
{
    Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass);
    // Pokreće Blynk povezanost s WiFi mrežom.
    for(int i = 0; i < 20; i++){
    Blynk.virtualWrite(i,0);
    // Postavlja sve Blynk virtualne pinove (0-19) na 0.

    }
//Dinamički alocira memoriju za niz pokazivača na jedinice, veličine n.
    units = (Unit**) calloc(n, sizeof(Unit*));
    va_list arguments;  // Inicijalizacija varijabilnog broja argumenata.
    va_start(arguments, n); // Početak pristupa varijabilnim argumentima.
    for (int x = 0; x < n; x++)
        units[x] = va_arg(arguments, Unit*);
// Pohranjuje svaki varijabilni argument (pokazivač na Unit) u niz units.
    va_end(arguments);                  // Zatvara varijabilne argumente.
}
```

```
bool FullModel::loop(){
    multipleLoadCells->loop();
    // Neprekidno poziva loop metodu za senzor težine.
    waterContainer->waterLevelLoop();
    // Neprekidno poziva metodu za provjeru razine vode.

    bool availableToUseBlynk = true;
    // Označava da je Blynk dostupan za korištenje.

// Provjera stanja za prvih 5 jedinica.
    for(int i = 0; i < 5; i++){

        if(false == units[i]->loop()){
        // Ako je jedan od jedinica aktivna, Blynk nije dostupan.
            availableToUseBlynk = false;
        }
        if(units[i]->isMotorActive()){
        // Ako je motor aktivan, Blynk nije dostupan.
            availableToUseBlynk = false;
        }
    }
    // Ako je Blynk dostupan, izvršava se Blynk loop.
    if(availableToUseBlynk){
        blynkLoop();
    }
    waterContainer->waterTemperatureLoop(availableToUseBlynk);
      // Poziva funkciju za provjeru temperature vode i prosljeđuje je Blynku,
      samo kada nije nijedna stanica aktivna i pumpa ne radi.
    return true;
}

int FullModel::getUnitActive(int unitId)
// Funkcija vraća aktivnost određene jedinice.
{
    return units[unitId]->getUnitActive();
}

int FullModel::getAnimalWeight(int unitId)
// Vraća težinu životinje iz određene jedinice.
{
    return units[unitId]->getAnimalWeight();
}

int FullModel::getWantedFoodWeight(int unitId)
// Vraća željenu težinu hrane za određenu jedinicu.
{
    return units[unitId]->getWantedFoodWeight();
}
```

```
int FullModel::getMeasuredFoodWeight(int unitId)
// Vraća izmjerenu težinu hrane za određenu jedinicu.
{
    return units[unitId]->getMeasuredFoodWeight();
}

void FullModel::resetUnitActive(int unitId)
// Resetira aktivnost određene jedinice.

{
    units[unitId]->resetUnitActive();
}

void FullModel::resetAnimalWeight(int unitId)
// Resetira težinu životinje za određenu jedinicu.
{
    units[unitId]->resetAnimalWeight();
}

void FullModel::resetWantedFoodWeight(int unitId)
// Resetira željenu težinu hrane za određenu jedinicu.
{
    units[unitId]->resetWantedFoodWeight();
}

void FullModel::resetMeasuredFoodWeight(int unitId)
// Resetira izmjerenu težinu hrane za određenu jedinicu.
{
        units[unitId]->resetMeasuredFoodWeight();
}
// Funkcija za slanje podataka na Blynk server.
void FullModel::blynkLoop(){
    double temperature = waterContainer->getCurrentTemperatureForBlynk();
    // Dohvaća trenutnu temperaturu.
    double power = waterContainer->getCurrentPowerForBlynk();
    // Dohvaća trenutnu snagu.
    int sign = waterContainer->getCurrentSignForBlynk();
    // Dohvaća trenutni znak.
    // Resetira spremljene vrijednosti nakon slanja.
    waterContainer->resetCurrentPowerForBlynk();
    waterContainer->resetCurrentSignForBlynk();
    waterContainer->resetCurrentTemperatureForBlynk();
```

```
// Ako je temperatura veća od 0, šalje na Blynk stanje sa pina 0.
    if(temperature > 0){
        Blynk.virtualWrite(0, temperature);
    }
    // Ako snaga nije -1, šalje na Blynk stanje sa pina 4.
    if(power != -1){
        Blynk.virtualWrite(4, power * 100);
    }
    // Ako znak nije 0, šalje na Blynk stanje sa pina 8.
    if(sign != 0){
        Blynk.virtualWrite(8, sign);
    }

//Petlja koja prolazi kroz sve jedinice za slanje njihovih podataka na Blynk.
    for(int i = 0; i < numberOfUnits; i++){
        int animalWeight = units[i]->getAnimalWeight();
            // Dohvaća težinu životinje.
        int wantedFoodWeight = units[i]->getWantedFoodWeight();
            // Dohvaća željenu težinu hrane.
        int measuredFoodWeight = units[i]->getMeasuredFoodWeight();
            // Dohvaća izmjerenu težinu hrane.

        if(animalWeight > 0){
            Blynk.virtualWrite(4 * i + 1, animalWeight);
            units[i]->resetAnimalWeight();
        }   // Ako je težina životinje veća od 0, šalje je na odgovarajući Blynk
pin i resetira vrijednost.
        if(wantedFoodWeight > 0){
            Blynk.virtualWrite(4 * i + 2, wantedFoodWeight);
            units[i]->resetWantedFoodWeight();
        }   // Ako je željena težina hrane veća od 0, šalje je na odgovarajući
Blynk pin i resetira vrijednost.
        if(measuredFoodWeight > 0){
            Blynk.virtualWrite(4 * i + 3, measuredFoodWeight);
            units[i]->resetMeasuredFoodWeight();
        }   // Ako je izmjerena težina hrane veća od 0, šalje je na odgovarajući
Blynk pin i resetira vrijednost.
        Blynk.run();
    }
}
```

FullModel.h

```
#ifndef FULL_MODEL_H
#define FULL_MODEL_H

#include "../loadCellLib/multipleLoadCells/multipleLoadCells.h"
#include "../Unit/Unit.h"
#include "../WaterContainer/waterContainer.h"

class FullModel {
private:
    int numberOfUnits;          // Pohranjuje broj jedinica (Unit) u sustavu.
    WaterContainer *waterContainer;             // Pokazivač na spremnik vode.
    MultipleLoadCells *multipleLoadCells;  // Pokazivač na više senzora težine
    Unit **units;                   // Polje pokazivača na objekte klase Unit.
public:
      // Konstruktor klase FullModel koji prima pokazivače na waterContainer,
      multipleLoadCells, te broj jedinica u sustavu (n).
    FullModel(
        WaterContainer *waterContainer,
        MultipleLoadCells *multipleLoadCells,
         int n,
        ...);

    bool loop();

    void blynkLoop();    // Funkcija koja se koristi za rad s Blynk platformom

    int getUnitActive(int unitId);
    // Dohvaća aktivnost određene jedinice i označava je li jedinica trenutno
    aktivna ili ne.
    int getAnimalWeight(int unitId);
    // Dohvaća težinu životinje u određenoj jedinici.
    int getWantedFoodWeight(int unitId);
    // Dohvaća željenu težinu hrane za životinju u određenoj jedinici.
    int getMeasuredFoodWeight(int unitId);
    // Dohvaća izmjerenu težinu hrane u određenoj jedinici.

    void resetUnitActive(int unitId);
    // Resetira status aktivnosti  za određenu jedinicu.
    void resetAnimalWeight(int unitId);
    // Resetira težinu životinje u određenoj jedinici.
    void resetWantedFoodWeight(int unitId);
    // Resetira željenu težinu hrane u određenoj jedinici.
    void resetMeasuredFoodWeight(int unitId);
    // Resetira izmjerenu težinu hrane u određenoj jedinici.

};

#endif
```

LoadCellLib

LoadCell.cpp

```
#include "./loadCell.h"
#include "loadCell.h"

#include<Arduino.h>

LoadCell::LoadCell(
    PinConfiguration::OutputPins mDataOutPin,
    double mCalibrationOffset,
    double mCalibrationConstant
):
    dataOutPin(mDataOutPin),
    calibrationOffset(mCalibrationOffset),
    calibrationConstant(mCalibrationConstant),
    scaleActive(false)
{
}

bool LoadCell::loop(){
    if(scaleActive){

    }
    return true;
}

void LoadCell::setToActive() {
    scaleActive = true;
}

bool LoadCell::isActive(){
    return scaleActive;
}

void LoadCell::cleanAllMeasurements()   // Funkcija koja briše sva prethodna
mjerenja postavljanjem njihovih vrijednosti i vremenskih oznaka na 0.
{
    for(int i = 0; i < LOAD_CELL_HISTORY_LENGTH; i++){
        measurements[i].value = 0;
        measurements[i].timestamp = 0;
    }
}
```

```
void LoadCell::cleanOldMeasurements()
{
    for(int i = 0; i < LOAD_CELL_HISTORY_LENGTH; i++){
        if(measurements[i].age() > 5000 && measurements[i].timestamp != 0) {
            measurements[i].value = 0;
            measurements[i].timestamp = 0;
        }
    }
}
// Funkcija koja briše stara mjerenja nakon 5 sekundi.
// Također provjerava jesu li ta mjerenja validna (vremenska oznaka nije 0).

void LoadCell::setNewRawReading(long rawReading){
    LoadCellMeasurement newReading(getCalibratedValue(rawReading));
    cleanOldMeasurements();
    for(int i = 0; i < LOAD_CELL_HISTORY_LENGTH; i++){
        if(measurements[i].value == 0) {
            measurements[i].value = newReading.value;
            measurements[i].timestamp = newReading.timestamp;
            break;
        }
    }
}
// Funkcija za postavljanje novog očitavanja (rawReading).
// Mjerenje se kalibrira, stara mjerenja se čiste, a novo mjerenje se
postavlja na prvo slobodno mjesto u povijesti mjerenja.


PinConfiguration::OutputPins LoadCell::getDataPin(){
    return dataOutPin;  // Funkcija koja vraća pin koji se koristi za
očitavanje podataka s vage.
}

double LoadCell::getCalibratedValue(long rawReading)
{
    return (rawReading - this->calibrationOffset) / this->calibrationConstant;
}
// Funkcija koja vraća kalibrisanu vrijednost očitavanja.
```

```
void LoadCell::averageAndStandardDeviationLastNSeconds(
    int seconds,
    double &average,
    double &standardDeviation
) {
    long timeThreshold = millis() - (seconds * 1000);   // Izračunavanje
vremenskog praga za zadnjih N sekundi.
    double sum = 0.0;
    double sumSquares = 0.0;
    int count = 0;
    for (int i = 0; i < LOAD_CELL_HISTORY_LENGTH; i++) {
        // Provjera jeli mjerenje unutar zadnjih N sekundi.
        if (measurements[i].timestamp >= timeThreshold) {
            double value = measurements[i].value;
            sum += value;  // Sumira se vrijednost
            sumSquares += value * value;  // Sumira se kvadrat vrijednosti
            count++;  // Povećava se broj validnih mjerenja
        }
    }
    if (count > 0) {
        average = sum / count;  // Izračunavanje prosjeka
        if (count > 1) {
            standardDeviation = sqrt((sumSquares / count) - (average *
average)); // Izračunavanje standardne devijacije
        } else {
            standardDeviation = 0.0;  // Ako je samo jedno mjerenje,
standardna devijacija je 0
        }
    } else {
        average = 0.0;  // Ako nema mjerenja, vraća se 0 kao prosjek
        standardDeviation = 0.0; // I standardna devijacija je također 0
    }
}
// Funkcija koja računa prosječnu vrijednost i standardnu devijaciju za
mjerenja unutar zadnjih N sekundi.

double LoadCell::getMaxValue() {
    double maxValue = measurements[0].value; // Inicijalizira se maxValue
prvom vrijednošću u nizu mjerenja.

    // Prolazi kroz niz mjerenja da bi se pronašla maksimalna vrijednost
    for (int i = 1; i < LOAD_CELL_HISTORY_LENGTH; i++) {
        if (measurements[i].value > maxValue) {
            maxValue = measurements[i].value;
        }
    }

    return maxValue; // Vraća se maksimalna vrijednost.
}
```

LoadCell.h

```
#ifndef LOAD_CELL_H
// Ovaj dio sprječava višestruko uključivanje istog header file-a.
#define LOAD_CELL_H
// Ako LOAD_CELL_H nije definirano, definira ga i uključuje sadržaj.
#include "../LoadCellMeasurement/loadCellMeasurement.h"
#include "../../Setup/pinconfiguration.h"
#define LOAD_CELL_HISTORY_LENGTH 1000
// Definira duljinu povijesti mjerenja s load cell-a (1000 mjerenja).
class LoadCell{
private:
    PinConfiguration::OutputPins dataOutPin;
    double calibrationOffset;
    // Offset kalibracije koji se koristi za kompenzaciju nule (tare).
    double calibrationConstant;
    // Konstanta kalibracije koja pretvara sirovo očitanje u pravu masu.
    bool scaleActive;   // Oznaka je li vaga aktivna (spremna za očitavanje).
    LoadCellMeasurement measurements[LOAD_CELL_HISTORY_LENGTH] = {0};
    // Polje za spremanje povijesti mjerenja.
public:     // Konstruktor klase koji prima pin, offset i konstantu za
kalibraciju.
LoadCell(PinConfiguration::OutputPins dataOutPin, double calibrationOffset,
double calibrationConstant);
    bool loop();
//Funkcija koja se poziva u glavnoj petlji programa za ažuriranje stanja vage.
    void setToActive();
    // Aktivira vagu i postavlja ju u stanje spremno za mjerenje.
    bool isActive();              // Provjerava je li vaga trenutno aktivna.
    void cleanAllMeasurements();           // Briše sva spremljena mjerenja.
    void cleanOldMeasurements();
    // Briše stara mjerenja kako bi se napravilo mjesta za nova.
    void setNewRawReading(long rawReading);
    // Sprema novo očitanje s load cell-a.
    PinConfiguration::OutputPins getDataPin();
    // Vraća pin na kojem se nalazi izlazni signal load cell-a.
    double getCalibratedValue(long rawReading);     // Izračunava i vraća
    kalibriranu vrijednost mase na temelju "sirovog" očitanja.
    void averageAndStandardDeviationLastNSeconds(   // Izračunava prosječnu
    vrijednost i standardnu devijaciju mjerenja u zadanim sekundama.
        int seconds,
        double &average,
        double &standardDeviation
    );
    double getMaxValue();       // Vraća maksimalnu vrijednost izmjerene mase
    u povijesti mjerenja.
};
#endif
```

LoadCellMeasurement.cpp

```
#include "loadCellMeasurement.h"

#include <Arduino.h>

// Konstruktor LoadCellMeasurement klase bez argumenata.
// Inicijalizira varijable 'value' na 0 i 'timestamp' na 0.
LoadCellMeasurement::LoadCellMeasurement(): value(0), timestamp(0){}

// Konstruktor LoadCellMeasurement klase s jednim argumentom (mValue).
// Postavlja vrijednost 'value' na predanu vrijednost (mValue),
// dok 'timestamp' postavlja na trenutni broj milisekundi od pokretanja
sistema.
LoadCellMeasurement::LoadCellMeasurement(
    double mValue
):
    value(mValue),
    timestamp(millis())
{

}

// Konstruktor LoadCellMeasurement klase s dva argumenta (mValue i
mTimestamp).
// Postavlja 'value' na predanu vrijednost (mValue) i 'timestamp' na predanu
vrijednost (mTimestamp).
LoadCellMeasurement::LoadCellMeasurement(
    double mValue,
    long mTimestamp
):
    value(mValue),
    timestamp(mTimestamp)
{

}
// Starost se računa kao razlika između trenutnog vremena (millis()) i
trenutka kad je mjerenje zabilježeno (timestamp).
long LoadCellMeasurement::age(){
    return millis() - timestamp;
}

void LoadCellMeasurement::print(){
    Serial.print("Time: "); // Ispisuje oznaku za vrijeme
    Serial.println(timestamp);  // Ispisuje vrijednost 'timestamp'
    Serial.print("Value:"); // Ispisuje oznaku za vrijednost
    Serial.println(value);  // Ispisuje vrijednost 'value'
}
```

LoadCellMeasurement.h

```
#ifndef LOAD_CELL_MEASUREMENT_H
// Ako nije definirana konstanta 'LOAD_CELL_MEASUREMENT_H', definiraj je
#define LOAD_CELL_MEASUREMENT_H
// Definiranje klase 'LoadCellMeasurement'
class LoadCellMeasurement {
public:
    double value;        // Varijabla za spremanje izmjerene vrijednosti
    long timestamp;     // Varijabla za spremanje vremenskog oznake
    LoadCellMeasurement();
    LoadCellMeasurement(double mValue);
    // Konstruktor koji prima izmjerenu vrijednost
    LoadCellMeasurement(double mValue, long mTimestamp);
    // Konstruktor koji prima izmjerenu vrijednost i vremensku oznaku
    long age();     // Varijabla koja vraća starost mjerenja (razliku između
    trenutnog vremena i timestamp-a)
    void print();       // Varijabla koja ispisuje podatke objekta (vrijednost
    i vremenski oznaku)
};
#endif
```

MultipleLoadCells.h

```
#ifndef MULTIPLE_LOAD_CELLS_H
#define MULTIPLE_LOAD_CELLS_H
#include "../LoadCell/loadCell.h"
#include "HX711-multi.h"
#define MULTIPLE_LOADCELL_LOOP_TIMER 100
//Definira konstantu koja određuje interval za funkciju loop()
class MultipleLoadCells {
private:
    byte clockPin;       // Clock pin koji se koristi za komunikaciju s HX711
    int n;               // Broj senzora (load cells) koji se koriste
    LoadCell **loadCells; // Niz pokazivača na objekte LoadCell
    byte *dOuts;
    //Niz pinova za podatke (data out pins) koji se koriste za svaki load cell
    long *results;      // Niz za pohranu rezultata mjerenja s load cell-ova
    long loopTimer;
    // Varijabla za praćenje vremena između izvršavanja funkcije loop()
    HX711MULTI *scales; // Objekt za komunikaciju s HX711 čipom koji omogućava
    rad s više load cell-ova
public:
    MultipleLoadCells(byte clockPin, int n, ...);
    // Konstruktor klase MultipleLoadCells koji prihvaća pin za clock, broj
    senzora i broj dodatnih argumenata
    void loop();
};
#endif
```

MultipleLoadCells,cpp

```
#include "./multipleLoadCells.h"

MultipleLoadCells::MultipleLoadCells(byte clockPin, int n, ...):
clockPin(clockPin), n(n) {
    loadCells = (LoadCell**) calloc(n, sizeof(LoadCell*));
    dOuts = (byte*) calloc(n, sizeof(byte));
    results = (long*) calloc(n, sizeof(long));
    pinMode(clockPin, OUTPUT);

    va_list arguments;        // Mjesto za pohranu popisa argumenata
    va_start(arguments, n);
    // Pokretanje argumenata za pohranu svih vrijednosti nakon n
    for (int x = 0; x < n; x++){        // Petlja koja prolazi kroz svaki
     LoadCell i pohranjuje ga u loadCells i dOuts
        loadCells[x] = va_arg(arguments, LoadCell*);
        dOuts[x] = loadCells[x]->getDataPin();
    }
    va_end(arguments);

    scales = new HX711MULTI(n, dOuts, clockPin);
    // Kreira novi HX711MULTI objekt za upravljanje više senzora
}

void MultipleLoadCells::loop(){         // Ako nije prošlo dovoljno vremena
(određeno MULTIPLE_LOADCELL_LOOP_TIMER), funkcija se prekida
    if(millis() - loopTimer <= MULTIPLE_LOADCELL_LOOP_TIMER) return;
    scales->read(results);
    // Čita podatke iz svih LoadCell senzora i sprema ih u results
    for (int i=0; i<scales->get_count(); ++i) {
    // Ako je LoadCell aktivan (provjerava isActive()), ažurira novi rezultat
     čitanja za LoadCell
        // Serial.println(results[i]);
        // Serial.println(loadCells[i]->getCalibratedValue(results[i]));
        if(loadCells[i]->isActive()){
            loadCells[i]->setNewRawReading(results[i]);
        }
    }
    // Serial.println("---");

    loopTimer = millis();
}
```

PIRSensor

PIRSensor.h

```
#ifndef PEER_SENSOR_H   // Provjerava je li PEER_SENSOR_H već definiran
#define PEER_SENSOR_H   // Definira PEER_SENSOR_H kako bi se spriječilo
višestruko uključivanje ovog zaglavlja

class PeerSensor{
public:
    virtual bool readSensor() = 0;      // Čista virtualna funkcija koja se
mora implementirati u izvedenim klasama

    virtual ~PeerSensor() {};
};

#endif
```

PIRSensorDigital.cpp

```
#include "peerSensorDigital.h"
#include "Arduino.h"

PeerSensorDigital::PeerSensorDigital(
    PinConfiguration::OutputPins pin
): pin(pin) // Inicijalizira član 'pin' klase s proslijeđenom vrijednošću.
{
    pinMode(pin, INPUT_PULLDOWN);
}
bool PeerSensorDigital::readSensor()
// Funkcija za očitavanje stanja senzora povezanog na određeni pin.
{
    return digitalRead(pin);     // Vraća trenutno stanje pina (HIGH ili LOW).
}
```

PIRSensorDigital.h

```
#ifndef PEER_SENSOR_DIGITAL_H
#define PEER_SENSOR_DIGITAL_H
#include "../Setup/pinconfiguration.h"
#include "peerSensor.h"

class PeerSensorDigital: public virtual PeerSensor {
    // Definira klasu PeerSensorDigital koja nasljeđuje klasu PeerSensor
private:
    PinConfiguration::OutputPins pin;
public:     // Konstruktor klase koji inicijalizira pin s predanom vrijednošću
    PeerSensorDigital(PinConfiguration::OutputPins pin);
    bool readSensor();      // Metoda koja čita podatke sa senzora i vraća
    rezultat u obliku boolean vrijednosti
};
#endif
```

Peltier

Peltier.h

```
#ifndef PELTIER_H
#define PELTIER_H
#include "../Setup/PinConfiguration.h"
#include "../shiftRegLib/shiftReg.h"

class Peltier{  // Definicija klase Peltier koja kontrolira Peltierov element
private:
    PinConfiguration::ShiftRegisterOutputPins peltierPin;
    // Pin konfiguriran za upravljanje Peltierom kroz shift register
    ShiftRegister *shiftRegister;
    // Pokazivač na shift register koji kontrolira odgovarajući pin
    double power;
    // Varijabla koja drži trenutnu snagu Peltierovog elementa (0-off, 1-on)
    long timer;
public: // Konstruktor klase koji inicijalizira Peltierov element s
odgovarajućim pinom i shift registrom
    Peltier(
        PinConfiguration::ShiftRegisterOutputPins peltierPin,
        // Parametar za pin na shift registru koji kontrolira Peltier
        ShiftRegister *shiftRegister
        // Pokazivač na shift register koji se koristi za slanje komandi
    );
    void setNewState(double newState);  // newState je vrijednost u rasponu
    [0, 1] koja određuje snagu Peltierovog elementa
    void loop();    // Koristi se za redovito ažuriranje stanja Peltiera
};
#endif
```

Peltier.cpp

```
#include "peltier.h"
#include "Arduino.h"

namespace {

#define PeltierPeriod 60000
// Definira trajanje ciklusa za Peltier uređaj u mikrosekundama (1 minuta)

}

// Konstruktor za klasu Peltier, postavlja inicijalne vrijednosti za pinove i
druge varijable
Peltier::Peltier(
    PinConfiguration::ShiftRegisterOutputPins mPeltierPin,
    //Pin za Peltier modul
    ShiftRegister *mShiftRegister   // Shift register koji kontrolira pinove
):
peltierPin(mPeltierPin),    // Inicijalizira peltierPin sa zadanim pinom
shiftRegister(mShiftRegister),  // Inicijalizira shiftRegister pokazivač
power(0),                // Početna vrijednost snage je postavljena na 0
timer(millis())
{ }

void Peltier::setNewState(double newState)
// Funkcija za postavljanje nove vrijednosti snage (power) za Peltier uređaj
{
    power = newState;           // Postavlja novu vrijednost snage
}

void Peltier::loop()
{
    double percentage = 1.0 * (millis() - timer) / PeltierPeriod;  // Računa
  postotak vremena koje je prošlo u odnosu na PeltierPeriod (trajanje ciklusa)
    if(millis() - timer > PeltierPeriod){
    //Ako je prošlo više od jednog ciklusa, resetira timer na trenutno vrijeme
        timer = millis();
    }
    shiftRegister->setPin(peltierPin, percentage <= power, true);
    // Postavlja stanje pina: ako je postotak manji ili jednak vrijednosti
    snage (power), uključi pin
}
```

Setup

BlynkPinConfiguration.h

```
#ifndef BLYNK_CONFIGURATION_H
#define BLYNK_CONFIGURATION_H
#include "../../privateData.h"

// Definira se enum koji se zove BlynkPinConfiguration.
// Enum sadrži pinove za različite konfiguracije box-a (UNIT_1 do UNIT_5)
// Koriste se za praćenje stanja uređaja i težine hrane ili životinje.
enum BlynkPinConfiguration {
    UNIT_1_ONGOING,
    //Signal koji označava da je proces za box 1 u toku.
    UNIT_1_LAST_ANIMAL_WEIGHT,
    // Pin za čuvanje posljednje izmjerene težine životinje za box 1.
    UNIT_1_LAST_FOOD_WEIGHT_WANTED,
    // Pin koji označava željenu težinu hrane za box 1.
    UNIT_1_LAST_FOOD_MEASURED,
    // Pin za čuvanje posljednje izmjerene težine hrane za box 1.

    UNIT_2_ONGOING,
    UNIT_2_LAST_ANIMAL_WEIGHT,
    UNIT_2_LAST_FOOD_WEIGHT_WANTED,
    UNIT_2_LAST_FOOD_MEASURED,

    UNIT_3_ONGOING,
    UNIT_3_LAST_ANIMAL_WEIGHT,
    UNIT_3_LAST_FOOD_WEIGHT_WANTED,
    UNIT_3_LAST_FOOD_MEASURED,

    UNIT_4_ONGOING,
    UNIT_4_LAST_ANIMAL_WEIGHT,
    UNIT_4_LAST_FOOD_WEIGHT_WANTED,
    UNIT_4_LAST_FOOD_MEASURED,

    UNIT_5_ONGOING,
    UNIT_5_LAST_ANIMAL_WEIGHT,
    UNIT_5_LAST_FOOD_WEIGHT_WANTED,
    UNIT_5_LAST_FOOD_MEASURED,
};

#endif
```

PinConfiguration.h

```
#ifndef PIN_CONFIGURAITON_H
#define PIN_CONFIGURAITON_H
namespace PinConfiguration {
// Definira skup izlaznih pinova (OutputPins) koji će se koristiti u
aplikaciji
enum OutputPins {
    ShiftRegLatchPin,    ,   // Pin za kontrolu kašnjenja u shift registru
    NC_1,                    // NC označava "Not Connected" - ovaj pin nije
povezan
    ShiftRegClockPin,        // Pin za kontrolu takta u shift registru
    NC_3,
    LoadCellData6,           // Pin za očitavanje podataka s senzora težine
(load cell)
    LoadCellData7,
    NC_6,
    NC_7,
    NC_8,
    NC_9,
    NC_10,
    NC_11,
    LoadCellClock,           // Pin za upravljanje taktom load cell senzora
    LowerWaterLevel,         // Pin za očitavanje razine niže vode
    LoadCellData0,
    ShiftRegDataPin,         // Pin za prijenos podataka u shift registru
    LoadCellData1,
    LoadCellData3,
    LoadCellData2,
    LoadCellData9,
    NC_20,
    LoadCellData4,
    DallasSensor,            // Pin za povezivanje s Dallas temperaturom
senzorom
    LoadCellData8,
    NC_24,
    PeerSensor4,             // Pin za očitavanje podataka s jednog od peer
senzora
    HigherWaterLevel,        // Pin za očitavanje razine više vode
    LoadCellData5,
    NC_28,
    NC_29,
    NC_30,
    NC_31,
    PeerSensor2,
    PeerSensor3,
    PeerSensor0,
    PeerSensor1,
    NC_36,
    NC_37,
```

```
    NC_38,
    NC_39,
    NC_40,
    NC_41,
    NC_42,
    NC_43,
    NC_44,
    NC_45,
    NC_46,
    NC_47,
    NC_48,
};
// Definira skup izlaznih pinova u shift registru
enum ShiftRegisterOutputPins {
    MOTOR_1_O1,     // Pin za prvu motornu kontrolu 1. motora
    MOTOR_1_O2,     // Pin za drugu motornu kontrolu 1. motora
    MOTOR_1_O3,     // Pin za treću motornu kontrolu 1. motora
    MOTOR_1_O4,     // Pin za četvrtu motornu kontrolu 1. motora

    MOTOR_2_O1,
    MOTOR_2_O2,
    MOTOR_2_O3,
    MOTOR_2_O4,

    MOTOR_3_O1,
    MOTOR_3_O2,
    MOTOR_3_O3,
    MOTOR_3_O4,

    MOTOR_4_O1,
    MOTOR_4_O2,
    MOTOR_4_O3,
    MOTOR_4_O4,

    MOTOR_5_O1,
    MOTOR_5_O2,
    MOTOR_5_O3,
    MOTOR_5_O4,

    NO_CONNECTION,          // Oznaka za nepovezani pin u shift registru
    WATER_PUMP,             // Pin za upravljanje pumpom za vodu
    PELTIER_COOLING_PIN,    // Pin za kontrolu Peltier hlađenja
    PELTIER_HEATING_PIN,    // Pin za kontrolu Peltier grijanja

    NUMBER_OF_SHIFT_REGISTER_PINS   // Ukupni broj pinova u shift registru
};
}
#endif
```

SetupData.h

```
#ifndef SETUP_DATA_H
#define SETUP_DATA_H

#include "../shiftRegLib/shiftReg.h"
#include "../loadCellLib/LoadCell/loadCell.h"
#include "../loadCellLib/multipleLoadCells/multipleLoadCells.h"
#include "../Unit/Unit.h"
#include "../FullModel/fullModel.h"
#include "pinconfiguration.h"
namespace SetupData {
// Definicija konstante za brzinu stepper motora
const double myStepperMotorSpeed{0.6};
ShiftRegister *shiftRegister;

LoadCell *LoadCell0_animal;
LoadCell *LoadCell1_animal;
LoadCell *LoadCell2_animal;
LoadCell *LoadCell3_animal;
LoadCell *LoadCell4_animal;
LoadCell *LoadCell0_food;
LoadCell *LoadCell1_food;
LoadCell *LoadCell2_food;
LoadCell *LoadCell3_food;
LoadCell *LoadCell4_food;
MultipleLoadCells *multipleLoadCells;
Unit *unit0;
Unit *unit1;
Unit *unit2;
Unit *unit3;
Unit *unit4;
WaterContainer *waterContainer;
FullModel *fullModel;

void runSetup(){
    Serial.begin(115200);
    shiftRegister = new ShiftRegister{
    // Kreira objekt za rad sa shift registrom, koristeći unaprijed definirane
    pinove iz PinConfiguration
        PinConfiguration::OutputPins::ShiftRegDataPin,
        PinConfiguration::OutputPins::ShiftRegClockPin,
        PinConfiguration::OutputPins::ShiftRegLatchPin,
        PinConfiguration::ShiftRegisterOutputPins::NUMBER_OF_SHIFT_REGISTER_PI
NS
    };
```

```
    // Inicijalizacija LoadCell za mjerenje životinja, svaki
    LoadCell ima svoj pin i kalibracijske vrijednosti
    LoadCell0_animal = new
LoadCell{PinConfiguration::OutputPins::LoadCellData0, 100000, 8000};
    LoadCell1_animal = new
LoadCell{PinConfiguration::OutputPins::LoadCellData1, -619000, 452};
    LoadCell2_animal = new
LoadCell{PinConfiguration::OutputPins::LoadCellData2, -142000, 349};
    LoadCell3_animal = new
LoadCell{PinConfiguration::OutputPins::LoadCellData3, -345850, -779};
    LoadCell4_animal = new
LoadCell{PinConfiguration::OutputPins::LoadCellData4, 22960, 808};

    // Inicijalizacija LoadCell za mjerenje hrane, svaki
    LoadCell ima svoj pin i kalibracijske vrijednosti
    LoadCell0_food = new LoadCell{PinConfiguration::OutputPins::LoadCellData5,
-9100, 363};
    LoadCell1_food = new LoadCell{PinConfiguration::OutputPins::LoadCellData6,
-292800, -710};
    LoadCell2_food = new LoadCell{PinConfiguration::OutputPins::LoadCellData7,
-47000, 352};
    LoadCell3_food = new LoadCell{PinConfiguration::OutputPins::LoadCellData8,
367000, 2000};
    LoadCell4_food = new LoadCell{PinConfiguration::OutputPins::LoadCellData9,
-18600, 1794};

    // Kreira objekt MultipleLoadCells koji povezuje sve senzore težine i
    definira njihov zajednički pin za clock signal
    multipleLoadCells = new MultipleLoadCells{
        PinConfiguration::OutputPins::LoadCellClock,
        10,
        LoadCell0_animal,
        LoadCell1_animal,
        LoadCell2_animal,
        LoadCell3_animal,
        LoadCell4_animal,
        LoadCell0_food,
        LoadCell1_food,
        LoadCell2_food,
        LoadCell3_food,
        LoadCell4_food
    };
```

```
// Inicijalizacija jedinica (box-a/unita), koje uključuju stepper motor i
senzore za težinu
unit0 = new Unit{
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_1_O1,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_1_O2,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_1_O3,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_1_O4,
    LoadCell0_animal,
    LoadCell0_food,
    shiftRegister,
    myStepperMotorSpeed,
    PinConfiguration::OutputPins::PeerSensor0,
};

unit1 = new Unit{
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_2_O1,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_2_O2,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_2_O3,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_2_O4,
    LoadCell1_animal,
    LoadCell1_food,
    shiftRegister,
    myStepperMotorSpeed,
    PinConfiguration::OutputPins::PeerSensor1
};

unit2 = new Unit{
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_3_O1,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_3_O2,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_3_O3,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_3_O4,
    LoadCell2_animal,
    LoadCell2_food,
    shiftRegister,
    myStepperMotorSpeed,
    PinConfiguration::OutputPins::PeerSensor2
};
unit3 = new Unit{
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_4_O1,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_4_O2,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_4_O3,
    PinConfiguration::ShiftRegisterOutputPins::MOTOR_4_O4,
    LoadCell3_animal,
    LoadCell3_food,
    shiftRegister,
    myStepperMotorSpeed,
    PinConfiguration::OutputPins::PeerSensor3
};
```

```
    unit4 = new Unit{
        PinConfiguration::ShiftRegisterOutputPins::MOTOR_5_O1,
        PinConfiguration::ShiftRegisterOutputPins::MOTOR_5_O2,
        PinConfiguration::ShiftRegisterOutputPins::MOTOR_5_O3,
        PinConfiguration::ShiftRegisterOutputPins::MOTOR_5_O4,
        LoadCell4_animal,
        LoadCell4_food,
        shiftRegister,
        myStepperMotorSpeed,
        PinConfiguration::OutputPins::PeerSensor4
    };
    // Inicijalizacija spremnika za vodu, koji koristi Peltier uređaji za
     grijanje i hlađenje, senzore razine vode, temperature i pumpu.
    waterContainer = new WaterContainer{
        PinConfiguration::ShiftRegisterOutputPins::PELTIER_COOLING_PIN,
        PinConfiguration::ShiftRegisterOutputPins::PELTIER_HEATING_PIN,
        PinConfiguration::OutputPins::DallasSensor,
        PinConfiguration::ShiftRegisterOutputPins::WATER_PUMP,
        PinConfiguration::OutputPins::LowerWaterLevel,
        PinConfiguration::OutputPins::HigherWaterLevel,
        shiftRegister
    };

    //Povezuje sve jedinice i senzore unutar sustava
    fullModel = new FullModel{
        waterContainer,
        multipleLoadCells,
        5,
        unit0,
        unit1,
        unit2,
        unit3,
        unit4
    };
}

}

#endif
```

ShiftRegLib

ShiftReg.cpp

```
#ifndef SHIFT_REGISTER_CC
#define SHIFT_REGISTER_CC
#include "./shiftReg.h"
#include <arduino.h>

ShiftRegister::ShiftRegister(
    PinConfiguration::OutputPins dataPin,   // Pin za slanje podataka
    PinConfiguration::OutputPins clockPin,  // Pin za slanje clock signala
    PinConfiguration::OutputPins latchPin,  // Pin za slanje Latch signala
    int numberOfOutputPins                  // Broj izlaznih pinova (broj
bita)
):  dataPin(dataPin),
    clockPin(clockPin),
    latchPin(latchPin),
    numberOfOutputPins(numberOfOutputPins)
{
    registerState = (bool*) calloc(numberOfOutputPins, sizeof(bool));

    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(latchPin, OUTPUT);

        setDataToOutput();

void ShiftRegister::setPin( // Funkcija koja postavlja određeni pin u registar
na određenu vrijednost (true/false)
    PinConfiguration::ShiftRegisterOutputPins pin,
    bool data,
    bool set
){
    if(pin < 0 || pin >= this->numberOfOutputPins) return;
    // Ako je broj pina van granica, funkcija se odmah vraca
    if(this->registerState[pin] == data) return;
    //Ako je trenutna vrijednost pina ista kao nova, nema potrebe za promjenom
    this->registerState[pin] = data;
    // Postavlja novu vrijednost za taj pin u memoriji
    if(set) this->setDataToOutput();
    // Ako je 'set' true, odmah šalje promjene na izlazne pinove
}
```

```
// Funkcija koja generira clock puls (neophodno za pomicanje podataka kroz
shift registar)
void ShiftRegister::clockPulse(){
    digitalWrite(this->clockPin, HIGH);
    delayMicroseconds(30);
    digitalWrite(this->clockPin, LOW);
    delayMicroseconds(30);
}

// Funkcija koja generira latch puls (koristi se za prihvaćanje trenutnog
stanja u shift registru)
void ShiftRegister::latchPulse(){
    digitalWrite(this->latchPin, HIGH);
    delayMicroseconds(30);
    digitalWrite(this->latchPin, LOW);
    delayMicroseconds(30);
}

// Funkcija koja šalje podatke iz lokalne memorije na izlazne pinove shift
registra
void ShiftRegister::setDataToOutput(){
    for(int i = this->numberOfOutputPins - 1; i >= 0; i--){
        digitalWrite(dataPin, this->registerState[i]);
        clockPulse();
    }
    latchPulse();
// Generira latch puls kako bi shift registar prihvatio promjene i proslijedio
ih dalje
}

#endif
```

ShiftReg.h

```
#ifndef SHIFT_REGISTER_H
#define SHIFT_REGISTER_H

#include "../Setup/pinconfiguration.h"

class ShiftRegister {
// Definira klasu ShiftRegister koja upravlja shift registrom
private:
    PinConfiguration::OutputPins dataPin;
    // Pin koji se koristi za prijenos podataka u shift register (Data Pin).
    PinConfiguration::OutputPins clockPin;
    // Pin koji se koristi za sinkronizaciju pomicanja podataka kroz shift
    register (Clock Pin).
    PinConfiguration::OutputPins latchPin;
    //Pin koji se koristi za učitavanje podataka u izlazni registar
    int numberOfOutputPins;         // Broj izlaznih pinova na shift registru.
    bool *registerState;
    // Polje koje drži trenutno stanje svakog bita unutar shift registra.

    void clockPulse();
    void latchPulse();

public:
// Konstruktor klase ShiftRegister. Inicijalizira data, clock i latch pinove
te broj izlaznih pinova.
    ShiftRegister(
        PinConfiguration::OutputPins dataPin,
        PinConfiguration::OutputPins clockPin,
        PinConfiguration::OutputPins latchPin,
        int numberOfOutputPins
    );
    void setPin(PinConfiguration::ShiftRegisterOutputPins pin, bool data, bool
set);
    void setDataToOutput();
};


#endif
```

StepperMotorLib

StepperMotor.cpp

```
#ifndef MY_STEPPER_MOTOR_CC
#define MY_STEPPER_MOTOR_CC
#include<Arduino.h>
#include<math.h>
#include "./myStepperMotor.h"
#include "myStepperMotor.h"

// Konstruktor za klasu myStepperMotor, inicijalizira motor s četiri pina i
postavlja osnovne parametre
myStepperMotor::myStepperMotor(
    PinConfiguration::ShiftRegisterOutputPins pin1,
    PinConfiguration::ShiftRegisterOutputPins pin2,
    PinConfiguration::ShiftRegisterOutputPins pin3,
    PinConfiguration::ShiftRegisterOutputPins pin4,
    ShiftRegister *shiftRegister,
    double rotationPeriod
    ):  pins{pin1, pin2, pin3, pin4},       // Pohrana pinova u niz
        shiftRegister(shiftRegister),       // Inicijalizacija registra
        cycleState(0),                      // Početno stanje ciklusa
        cycleTimer(0),                      // Početni timer ciklusa
        leftoverSteps(0)                    // Broj preostalih koraka je nula
    {
        if(CYCLE_SIZE != 4 && CYCLE_SIZE != 8){
        // Provjerava je li veličina ciklusa valjana (mora biti 4 ili 8)
            throw("Incorrect CYCLE_SIZE");
        }
        timerPeriod = rotationPeriod / 200 / CYCLE_SIZE * 1000;
        // Računanje vremena između koraka motora, ovisno o periodi rotacije
    }

bool myStepperMotor::loop()
{
    if(leftoverSteps == 0){     // Ako nema preostalih koraka, zaustavi motor
        cycleState = 0;
        for(int i = 0; i < 4; i++){
            shiftRegister->setPin(pins[i], false, false);
        }
        shiftRegister->setDataToOutput();   // Ažuriraj izlaz registra
        return true;
    }
    if(millis() - cycleTimer <= timerPeriod) return true;
    // Ako nije prošlo dovoljno vremena, čekaj
```

```
    for(int i = 0; i < 4; i++){
    // Postavi odgovarajuće pinove za trenutno stanje ciklusa
        shiftRegister->setPin(pins[i], getPinValueAtStep(cycleState,i),
false);
    }

    shiftRegister->setDataToOutput();
    incrementCycleState();
    cycleTimer = millis();
    return true;
}
void myStepperMotor::incrementCycleState()
{
    cycleState += (leftoverSteps > 0) - (leftoverSteps < 0);
    // Uvećava ili smanjuje stanje ciklusa, ovisno o smjeru rotacije
    leftoverSteps -= (leftoverSteps > 0) - (leftoverSteps < 0);
    if(cycleState >= CYCLE_SIZE) cycleState = 0;
    // Resetiraj stanje ciklusa ako pređe granice
    if(cycleState < 0) cycleState = CYCLE_SIZE - 1;
}
bool myStepperMotor::getPinValueAtStep(int cycleState, int pin)
{
    if(pin < 0 || pin >= 4) throw("Unvalid pin for stepper motor value");
    // Provjera je li pin i stanje ciklusa unutar valjanog raspona
    if(cycleState < 0 || cycleState >= CYCLE_SIZE) throw("Unvalid step for
stepper motor value");
    if(CYCLE_SIZE == 4){
    // Ako je CYCLE_SIZE 4, koristi set uzoraka za upravljanje motorom
        bool CYCLE_SIZE_4[4][4] = {
            {1, 0, 0, 1},    // Prvi korak: aktiviraj pinove 1 i 4
            {1, 1, 0, 0},    // Drugi korak: aktiviraj pinove 1 i 2
            {0, 1, 1, 0},    // Treći korak: aktiviraj pinove 2 i 3
            {0, 0, 1, 1},    // Četvrti korak: aktiviraj pinove 3 i 4
        };
        return CYCLE_SIZE_4[cycleState][pin];
    } else if(CYCLE_SIZE == 8){
      // Ako je CYCLE_SIZE 8, koristi prošireni set uzoraka
        bool CYCLE_SIZE_8[8][4] = {
            {1, 0, 0, 1},      // Prvi korak
            {1, 0, 0, 0},      // Drugi korak
            {1, 1, 0, 0},      // Treći korak
            {0, 1, 0, 0},      // Četvrti korak
            {0, 1, 1, 0},      // Peti korak
            {0, 0, 1, 0},      // Šesti korak
            {0, 0, 1, 1},      // Sedmi korak
            {0, 0, 0, 1},      // Osmi korak
        };
        return CYCLE_SIZE_8[cycleState][pin];
```

```
} else{
        throw("Unvalid Cycle size for stepper motor");
        // Greška ako je veličina ciklusa nevažeća
    }
}
bool myStepperMotor::isMotorActive()
// Funkcija koja provjerava je li motor aktivan (ima li preostalih koraka)
{
    return leftoverSteps != 0;
}
void myStepperMotor::setRotations(double rotations){
// Funkcija za postavljanje broja rotacija motora
    leftoverSteps = rotations * 200 / 4 * CYCLE_SIZE;
    // Izračunavanje preostalih koraka za zadani broj rotacija
}
#endif
```

StepperMotor.h

```
#ifndef MY_STEPPER_MOTOR_H
#define MY_STEPPER_MOTOR_H
#include "../shiftRegLib/shiftReg.h"
#include "../Setup/pinconfiguration.h"
#define CYCLE_SIZE 8
// Definiraj konstantu koja predstavlja broj koraka u ciklusu motora
class myStepperMotor {
// Definicija klase myStepperMotor koja upravlja stepper motorom
private:
    PinConfiguration::ShiftRegisterOutputPins pins[4];
    // Polje od 4 pina povezanih s motorom preko shift registra
    ShiftRegister *shiftRegister;
    // Pokazivač na shift registar, koristi se za kontrolu signala prema motoru
    int cycleState;
    long cycleTimer;
    int timerPeriod;
    int leftoverSteps;
    void incrementCycleState();
    // Povećava stanje ciklusa (za prelazak na sljedeći korak u rotaciji)
    bool getPinValueAtStep(int cycleState, int pin);
    // Vraća vrijednost (logički 1 ili 0) koju treba postaviti na pin za
     određeni korak ciklusa
```

```
public: // Konstruktor klase, inicijalizira stepper motor s odgovarajućim
pinovima, shift registrom i periodom rotacije
    myStepperMotor(
        PinConfiguration::ShiftRegisterOutputPins pin1,
        PinConfiguration::ShiftRegisterOutputPins pin2,
        PinConfiguration::ShiftRegisterOutputPins pin3,
        PinConfiguration::ShiftRegisterOutputPins pin4,
        ShiftRegister *shiftRegister,
        double rotationPeriod
    );
    bool loop();
    void setRotations(double rotations);
    bool isMotorActive();
};
#endif
```

TemperatureSensor

TemperatureSensor.cpp

```
#include "temperatureSensor.h"

TemperatureSensor::TemperatureSensor(PinConfiguration::OutputPins pin)
// Konstruktor TemperatureSensor klase, prima pin kao parametar
{
    oneWire = new OneWire(pin);
    // Kreira novi OneWire objekt koji je zadužen za komunikaciju preko
    određenog pina
    sensors = new DallasTemperature(oneWire);
    // Kreira novi DallasTemperature objekt za rad s temperaturnim senzorom
    sensors->begin();             // Inicijalizira komunikaciju sa senzorom
}

double TemperatureSensor::readTemperature()
// Funkcija koja čita temperaturu sa senzora
{
    sensors->requestTemperatures();
    //Šalje zahtjev senzoru da mjeri temperaturu
    double tempC = sensors->getTempCByIndex(0);
    // Dohvaća temperaturu u Celzijusima s prvog senzora na OneWire sabirnici
    return tempC;         // Vraća očitanu temperaturu
}
```

TemperatureSensor.h

```
#ifndef TEMPERATURE_SENSOR_H
#define TEMPERATURE_SENSOR_H
#include <OneWire.h>
#include <DallasTemperature.h>
#include "../Setup/PinConfiguration.h"

class TemperatureSensor{    // Definira klasu TemperatureSensor koja će
služiti za očitavanje temperature s senzora.
private:
// Ovdje se definiraju pokazivači na objekte za OneWire i DallasTemperature.
    OneWire *oneWire;   // Pokazivač na objekt OneWire klase, koristi se za
komunikaciju sa senzorima putem OneWire protokola.
    DallasTemperature *sensors;     // Pokazivač na objekt DallasTemperature
klase, koristi se za rad s DS18B20 senzorima.
public:  // Konstruktor klase koji prihvaća pin konfiguraciju (koji pin na
mikrokontroleru koristi senzor).
    TemperatureSensor(PinConfiguration::OutputPins pin);
    double readTemperature();
    // Čita temperaturu sa senzora i vraća je kao double vrijednost.
};

#endif
```

Unit

Unit.cpp

```cpp
#include "Unit.h"
#include "../PeerSensor/peerSensorDigital.h"

Unit::Unit( // Konstruktor klase Unit. Inicijalizira motore, vage, te senzor.
    PinConfiguration::ShiftRegisterOutputPins motorPin1,
    PinConfiguration::ShiftRegisterOutputPins motorPin2,
    PinConfiguration::ShiftRegisterOutputPins motorPin3,
    PinConfiguration::ShiftRegisterOutputPins motorPin4,
    LoadCell *animalScale,
    LoadCell *foodScale,
    ShiftRegister *shiftRegister,
    double rotationPeriod,
    PinConfiguration::OutputPins peerSensorPin
):
    motor{
        motorPin1,
        motorPin2,
        motorPin3,
        motorPin4,
        shiftRegister,
        rotationPeriod
    },
    animalScale(animalScale),
    foodScale(foodScale)
{
    peerSensor = new PeerSensorDigital{peerSensorPin};
    // Stvara novi PeerSensorDigital objekt koristeći ulazni pin.
    state = UnitState::WAITING_FOR_PEER_SENSOR_DETECTION;
   //Postavlja početno stanje jedinice na "WAITING_FOR_PEER_SENSOR_DETECTION".
}
bool Unit::checkPeerSensor(){
// Funkcija provjerava je li senzor detektirao prisutnost pomoću peerSensora.
    return peerSensor->readSensor();
}
void Unit::turnOnAnimalScale(){
// Funkcija aktivira vagu za mjerenje težine životinje.
    animalScale->setToActive();
}
void Unit::turnOnFoodScale(){
// Funkcija aktivira vagu za mjerenje težine hrane.
    foodScale->setToActive();
}
```

```
void Unit::checkAnimalScale(){
}
void Unit::checkFoodScale(){
}

void Unit::setMotorRotations(double rotations){
// Postavlja broj rotacija za motor.
    motor.setRotations(rotations);
}
// Funkcija koja upravlja glavnim stanjem uređaja i vraća informaciju može li
se Blynk aplikacija koristiti.
bool Unit::loop(){
    bool availableToUseBlynk = true;
    switch(state){
        case WAITING_FOR_PEER_SENSOR_DETECTION:
            waitingForInputLoop();
            break;
        case MEASURING_ANIMAL_WEIGHT:
            measuringAnimalWeightLoop();
            break;
        case DISPENCING_FOOD:
            dispencingFoodLoop();
            availableToUseBlynk = false;
            break;
        case MEASURING_TOTAL_FOOD_WEIGHT:
            // Pričekaj da sve vage budu 0
            // Pričekaj da se PIR senzor ugasi
            motor.loop();
            if(measuredFoodWeight == -2 && false == motor.isMotorActive()){
                double average;
                double standardDeviation;
                foodScale->averageAndStandardDeviationLastNSeconds(1, average,
standardDeviation);    // Izračunava prosjek i standardnu devijaciju težine
hrane u zadnjoj sekundi.
                if(standardDeviation < 4){
                    measuredFoodWeight = average;
                    state = WAITING_FOR_DATA_RESET;
                }
            }
            break;
       case WAITING_FOR_DATA_RESET:    // Resetira podatke kada su prosjek i
         standardna devijacija težina unutar zadanih granica.
            double average1;
            double standardDeviation1;
            double average2;
            double standardDeviation2;
```

```
animalScale->averageAndStandardDeviationLastNSeconds(3, average1,
standardDeviation1);
            foodScale->averageAndStandardDeviationLastNSeconds(3, average2,
standardDeviation2);
            if(average1 < 50 && average2 < 10 && standardDeviation1 < 10 &&
standardDeviation2 < 10){
                state = WAITING_FOR_PEER_SENSOR_DETECTION;
            }

            break;
    }
    return availableToUseBlynk;
}

void Unit::waitingForInputLoop(){
    // Funkcija koja se izvršava dok jedinica čeka na unos s peerSensora.
    if(checkPeerSensor()){
        unitActive = 1;
        turnOnAnimalScale();
        animalScaleActivatedTimestamp = millis();
        state = MEASURING_ANIMAL_WEIGHT;
    }
}
void Unit::measuringAnimalWeightLoop(){
    // Funkcija za mjerenje težine životinje.
    if(millis() - animalScaleActivatedTimestamp > 1000){
        double average;
        double standardDeviation;
        animalScale->averageAndStandardDeviationLastNSeconds(1, average,
standardDeviation);

        bool decition = standardDeviation < 10 && average > 100;
        // Odluka o ispuštanju hrane na temelju težine životinje.
        if(decition){
            wantedFoodWeightForRef = calculateFoodWeight(average);
            animalWeight = average; // set blynk data
            wantedFoodWeight = wantedFoodWeightForRef;
            motor.setRotations(FULL_OPENED_STEPPER_MOTOR);
            state = DISPENCING_FOOD;
            turnOnFoodScale();
        }

        else{
            animalScaleActivatedTimestamp = millis();
        }
    }
}
```

```
void Unit::dispencingFoodLoop(){
    // Funkcija za ispuštanje hrane.
    motor.loop();
    double average, standardDeviation;

    double maxValue = foodScale->getMaxValue();
    if(maxValue > wantedFoodWeightForRef * PERCENTAGE_OF_WEIGHT_TO_STOP_FOOD){
        motor.setRotations(-1 * FULL_OPENED_STEPPER_MOTOR);
        state = MEASURING_TOTAL_FOOD_WEIGHT;
        measuredFoodWeight = -2;
    }
}

double Unit::calculateFoodWeight(double animalWeight)
// Funkcija koja računa potrebnu količinu hrane na temelju težine životinje.
{
    return animalWeight / 10;
}

bool Unit::isMotorActive()  // Provjerava je li motor aktivan.
{
    return motor.isMotorActive();
}

int Unit::getUnitActive()   // Vraća status jedinice (aktivna ili ne).
{
    return unitActive;
}

int Unit::getAnimalWeight()   // Vraća izmjerenu težinu životinje.
{
    return animalWeight;
}

int Unit::getWantedFoodWeight()    // Vraća potrebnu količinu hrane.
{
    return wantedFoodWeight;
}

int Unit::getMeasuredFoodWeight()   // Vraća izmjerenu količinu hrane.
{
    return measuredFoodWeight;
}

void Unit::resetUnitActive()    // Resetira status jedinice na neaktivno.
{
    unitActive = 0;
}
```

```
void Unit::resetAnimalWeight()       // Resetira težinu životinje.
{
    animalWeight = 0;
}

void Unit::resetWantedFoodWeight()    // Resetira željenu količinu hrane.
{
    wantedFoodWeight = 0;
}

void Unit::resetMeasuredFoodWeight()    // Resetira izmjerenu količinu hrane.
{
    measuredFoodWeight = 0;
}
```

Unit.h

```
#ifndef UNIT_H
#define UNIT_H
#include "../StepperMotorLib/myStepperMotor.h"
#include "../loadCellLib/LoadCell/loadCell.h"
#include "../shiftRegLib/shiftReg.h"
#include "../PeerSensor/peerSensor.h"
#include "Arduino.h"

#define FULL_OPENED_STEPPER_MOTOR -1.275
 // Vrijednost za potpuno otvoren motor
#define PERCENTAGE_OF_WEIGHT_TO_STOP_FOOD 0.7 // [0,1]

enum UnitState {
    WAITING_FOR_PEER_SENSOR_DETECTION,
    MEASURING_ANIMAL_WEIGHT,
    DISPENCING_FOOD,
    MEASURING_TOTAL_FOOD_WEIGHT,
    WAITING_FOR_DATA_RESET
};

class Unit {
    // Klasa Unit koja upravlja cjelokupnim sustavom dispenzora hrane
private:
    UnitState state;
    myStepperMotor motor;
    LoadCell *animalScale;
    LoadCell *foodScale;
    PeerSensor *peerSensor;
```

```
int unitActive = -1;
      // Varijabla koja prati aktivnost jedinice (-1 znači neaktivno)
    double animalWeight = 0;
    double wantedFoodWeight = 0;
    double measuredFoodWeight = 0;
    long animalScaleActivatedTimestamp;
    double wantedFoodWeightForRef;

public:
    // Konstruktor koji inicijalizira sve potrebne komponente
    Unit(
        PinConfiguration::ShiftRegisterOutputPins motorPin1,
        PinConfiguration::ShiftRegisterOutputPins motorPin2,
        PinConfiguration::ShiftRegisterOutputPins motorPin3,
        PinConfiguration::ShiftRegisterOutputPins motorPin4,
        LoadCell *animalScale,
        LoadCell *foodScale,
        ShiftRegister *shiftRegister,
        double rotationPeriod,
        PinConfiguration::OutputPins peerSensorPin
    );
    // Funkcije koje provjeravaju stanja svake komponente, uključuju ih.
    bool checkPeerSensor();
    void checkAnimalScale();
    void turnOnAnimalScale();
    void turnOnFoodScale();
    void checkFoodScale();
    void setMotorRotations(double rotations);
    // Funkcija za postavljanje broja rotacija motora

    bool loop();     // Glavna funkcija petlje koja upravlja cijelim procesom
    void waitingForInputLoop();
    // Petlja koja se izvršava kada se čeka unos podataka
    void measuringAnimalWeightLoop();
    // Petlja koja se izvršava kada se mjeri težina životinje
    void dispencingFoodLoop();
    // Petlja koja se izvršava tijekom dispenziranja hrane
    double calculateFoodWeight(double animalWeight);
    // Funkcija za izračunavanje težine hrane na temelju težine životinje
    bool isMotorActive();    // Funkcija koja provjerava je li motor aktivan
    // Funkcije za dobivanje vrijednosti trenutne aktivnosti jedinice, težine
životinje, željene i izmjerene težine hrane
    int getUnitActive();
    int getAnimalWeight();
    int getWantedFoodWeight();
    int getMeasuredFoodWeight();
```

```
    // Funkcije za resetiranje varijabli stanja jedinice, težine životinje,
željene težine hrane i izmjerene težine hrane
    void resetUnitActive();
    void resetAnimalWeight();
    void resetWantedFoodWeight();
    void resetMeasuredFoodWeight();
};

#endif
```

WaterContainer

WaterContainer.cpp

```
#include "waterContainer.h"
// Konstruktor klase WaterContainer - inicijalizira objekte i varijable
WaterContainer::WaterContainer(
    PinConfiguration::ShiftRegisterOutputPins peltierCoolingPin,
      // Pin za hlađenje
    PinConfiguration::ShiftRegisterOutputPins peltierHeatingPin,
      // Pin za grijanje
    PinConfiguration::OutputPins temperatureSensorPin,
      // Pin za senzor temperature
    PinConfiguration::ShiftRegisterOutputPins waterPumpPin,
      // Pin za vodenu pumpu
    PinConfiguration::OutputPins lowerWaterLevelPin,
      // Pin za senzor niže razine vode
    PinConfiguration::OutputPins higherWaterLevelPin,
      // Pin za senzor više razine vode
    ShiftRegister *shiftRegister) : peltierCooling{
      // Shift register za kontrolu pinova
                                        peltierCoolingPin,
                                        shiftRegister
                                    },
                                     // Inicijalizacija hlađenja i grijanja
                                    peltierHeating{
                                        peltierHeatingPin,
                                        shiftRegister
                                    },
```

```
                                    waterLevel{
                                     // Inicijalizacija razine vode
                                        lowerWaterLevelPin,
                                        higherWaterLevelPin,
                                        waterPumpPin,
                                        shiftRegister
                                    },
                                    tempSensor{
                                     // Inicijalizacija senzora temperature
                                        temperatureSensorPin
                                    },
                                    temperatureSetting{
                                        35
                            // Početna željena temperatura (35°C zbog testiranja)
                                    },
                                    currentPower{
                                     // Početna snaga je 0
                                        0
                                    }
{}

void WaterContainer::setTemperature(double temperature){
    temperatureSetting = temperature;   // Sprema zadanu temperaturu
}

void WaterContainer::waterLevelLoop(){  // Petlja za kontrolu razine vode
    waterLevel.loop();
}

void WaterContainer::waterTemperatureLoop(bool freeToBlock){
// Petlja koja kontrolira temperaturu vode (hlađenje/grijanje)
    if(waterLevel.readLowerSensor()){
     // Provjerava je li niža razina vode aktivna
        if(freeToBlock){
            currentPower = calculatePeltierPower();
            // Izračunava potrebnu snagu za Peltier
        }
        if(currentPower < 0){
            peltierCooling.setNewState(fabs(currentPower));
            peltierHeating.setNewState(0);
        }
        else if(currentPower > 0){
            peltierCooling.setNewState(0);
            peltierHeating.setNewState(fabs(currentPower));
        } else {
            peltierCooling.setNewState(0);
            peltierHeating.setNewState(0);
        }
```

```
    } else {                    // Ako nema dovoljno vode, isključuje sustav
      peltierCooling.setNewState(0);
      peltierHeating.setNewState(0);
      powerForReport = 0;
      signForReport = 0;
    }
    peltierCooling.loop();
    peltierHeating.loop();
}

double WaterContainer::getCurrentTemperatureForBlynk()
// Vraća trenutnu temperaturu za prikaz preko Blynk platforme
{
    return currentTemperatureForReport;
}

void WaterContainer::resetCurrentTemperatureForBlynk()
// Resetira trenutnu temperaturu za Blynk izvještaj
{
    currentTemperatureForReport = 0;
}


double WaterContainer::getCurrentPowerForBlynk()
// Vraća trenutnu snagu za prikaz preko Blynk platforme
{
    return powerForReport;
}

void WaterContainer::resetCurrentPowerForBlynk()
// Resetira trenutnu snagu za Blynk izvještaj
{
    powerForReport = -1;
}

int WaterContainer::getCurrentSignForBlynk()
// Vraća trenutni znak snage za prikaz preko Blynk platforme (1 = grijanje, -1
= hlađenje)
{
    return signForReport;
}

void WaterContainer::resetCurrentSignForBlynk()
// Resetira trenutni znak snage za Blynk izvještaj
{
    signForReport = 0;
}
```

```
double WaterContainer::calculatePeltierPower()
// Izračunava snagu za Peltier
{
    double temperature = tempSensor.readTemperature();

    currentTemperatureForReport = temperature;
    double diff = temperatureSetting - temperature;
    double absoluteDiff = fabs(diff);

    if(absoluteDiff < TEMP_DIFF_FOR_NO_PELTIER_DRIVE) {
      // Ako je razlika premala, Peltier ne radi
        powerForReport = 0;
        return 0;
    }
    double power = absoluteDiff / TEMP_DIFF_FOR_MAX_PELTIER_DRIVE;

    if(power > 1){
        power = 1;
    }
    int sign = diff > 0? 1: -1;
    // Postavlja znak: 1 za grijanje, -1 za hlađenje

    powerForReport = power;
    signForReport = sign;

    return power * sign;
}
```

WaterContainer.h

```
#ifndef WATER_CONTAINER_H
#define WATER_CONTAINER_H

#include "../Setup/PinConfiguration.h"
#include "../shiftRegLib/shiftReg.h"
#include "../Peltier/peltier.h"
#include "../WaterLevel/waterLevel.h"
#include "../TemperatureSensor/temperatureSensor.h"

#define TEMP_DIFF_FOR_NO_PELTIER_DRIVE 1
// Kada je temperaturna razlika manja od 1, Peltier ne radi
#define TEMP_DIFF_FOR_MAX_PELTIER_DRIVE 10
// Kada je temperaturna razlika 10 ili više, Peltier radi punom snagom

class WaterContainer{
// Definicija klase WaterContainer (spremnik vode) koja upravlja temperaturom
vode i razinom vode
```

```
private:
    Peltier peltierHeating;     // Peltier uređaj za grijanje
    Peltier peltierCooling;     // Peltier uređaj za hlađenje
    WaterLevel waterLevel;      // Objekt za praćenje razine vode
    TemperatureSensor tempSensor;   // Senzor za mjerenje temperature vode
    double temperatureSetting;  // Željena postavka temperature
    double currentPower;
    // Trenutna snaga koja se koristi za upravljanje Peltierovim uređajem
    double currentTemperatureForReport;
    double powerForReport;
    int signForReport;

    double calculatePeltierPower();
    // Funkcija za izračunavanje snage koju treba dati Peltieru, ovisno o
    trenutnoj i ciljanoj temperaturu
public:
    WaterContainer(
        PinConfiguration::ShiftRegisterOutputPins peltierPin1,
        // Pin za grijanje Peltiera
        PinConfiguration::ShiftRegisterOutputPins peltierPin2,
        // Pin za hlađenje Peltiera
        PinConfiguration::OutputPins temperatureSensorPin,
        // Pin za povezivanje senzora temperature
        PinConfiguration::ShiftRegisterOutputPins waterPumpPin,
        // Pin za pumpu za vodu
        PinConfiguration::OutputPins lowerWaterLevelPin,
        // Pin za senzor niže razine vode
        PinConfiguration::OutputPins higherWaterLevelPin,
        // Pin za senzor više razine vode
        ShiftRegister *shiftRegister
        // Pokazivač na pomični registar koji kontrolira uređaje
    );
    void setTemperature(double temperature);
    // Postavlja željenu temperaturu spremnika vode
    void waterLevelLoop();
    // Funkcija za kontrolu razine vode, vrši provjere razine vode i upravlja
     pumpom
    void waterTemperatureLoop(bool freeToBlock);
    // Funkcija za kontrolu temperature vode, ovisno o tome je li slobodno
       blokirati operaciju
    double getCurrentTemperatureForBlynk();
    // Funkcija koja vraća trenutnu temperaturu za Blynk aplikaciju
    void resetCurrentTemperatureForBlynk();
    // Resetira temperaturu za Blynk nakon što je pročitana
```

```
    double getCurrentPowerForBlynk();
    // Funkcija koja vraća trenutnu snagu Peltier uređaja za Blynk aplikaciju
    void resetCurrentPowerForBlynk();
    // Resetira vrijednost trenutne snage nakon što je pročitana za Blynk
    int getCurrentSignForBlynk();
    // Vraća trenutni znak za Blynk aplikaciju
    void resetCurrentSignForBlynk();
    // Resetira znak nakon što je pročitana vrijednost za Blynk
};

#endif
```

WaterLevel

WaterLevel.cpp

```
#include "waterLevel.h"
#include "Arduino.h"

// Konstruktor klase WaterLevel koji prima pinove za senzore i pumpu te
pokazivač na registrator pomaka (shift register)
WaterLevel::WaterLevel(
    PinConfiguration::OutputPins mLowerWaterLevelPin,
     // Pin za donji senzor razine vode
    PinConfiguration::OutputPins mHigherWaterLevelPin,
     // Pin za gornji senzor razine vode
    PinConfiguration::ShiftRegisterOutputPins mPumpPin,
     // Pin za upravljanje pumpom preko shift registra
    ShiftRegister *mShiftRegister
):
    lowerWaterLevelPin(mLowerWaterLevelPin),
    higherWaterLevelPin(mHigherWaterLevelPin),
    pumpPin(mPumpPin),
    shiftRegister(mShiftRegister)
{
    pinMode(lowerWaterLevelPin, INPUT_PULLUP);
    pinMode(higherWaterLevelPin, INPUT_PULLUP);
}

void WaterLevel::setPump(bool state){
// Funkcija koja postavlja stanje pumpe, koristi shift registar za kontrolu
    shiftRegister->setPin(pumpPin, state, true);
}
```

```
bool WaterLevel::readLowerSensor()
// Funkcija za čitanje stanja donjeg senzora razine vode
{
    return digitalRead(lowerWaterLevelPin);
}

bool WaterLevel::readHigherSensor()
// Funkcija za čitanje stanja gornjeg senzora razine vode
{
    return digitalRead(higherWaterLevelPin);
}

void WaterLevel::loop()
{
    bool lowerLevel = readLowerSensor();
    // Čitanje stanja donjeg senzora i spremanje u varijablu lowerLevel
    bool higherLevel = readHigherSensor();
    // Čitanje stanja gornjeg senzora i spremanje u varijablu higherLevel

    switch (higherLevel) {
        case false:
// Ako je gornji senzor u "false" stanju (razina vode ispod gornjeg senzora)
            setPump(true);  // Uključuje pumpu
            break;
        case true:
// Ako je gornji senzor u "true" stanju (razina vode iznad gornjeg senzora)
            setPump(false); // Isključuje pumpu
    }
}
```

WaterLevel.h

```
#ifndef WATER_LEVEL_H
#define WATER_LEVEL_H

#include"../Setup/PinConfiguration.h"
#include "../shiftRegLib/shiftReg.h"

class WaterLevel{
// Definicija klase WaterLevel koja upravlja senzorima za razinu vode i pumpom
private:
//konfiguracija pinova i rad s shift registrom
    PinConfiguration::OutputPins lowerWaterLevelPin;
    PinConfiguration::OutputPins higherWaterLevelPin;
    PinConfiguration::ShiftRegisterOutputPins pumpPin;
    ShiftRegister *shiftRegister;
    void setPump(bool state);
    //Funkcija koja postavlja stanje pumpe (uključeno ili isključeno)

public:
  // Konstruktor klase koji prima konfiguraciju pinova i objekt shift registra
    WaterLevel(
        PinConfiguration::OutputPins lowerWaterLevelPin,
        PinConfiguration::OutputPins higherWaterLevelPin,
        PinConfiguration::ShiftRegisterOutputPins pumpPin,
        ShiftRegister *shiftRegister
    );
    void loop();
    bool readLowerSensor();      // Funkcija za očitanje statusa senzora niže
    razine vode (vraća true/false ovisno o stanju)
    bool readHigherSensor();        // Funkcija za očitanje statusa senzora
    više razine vode (vraća true/false ovisno o stanju)
};

#endif
```

Hranilica.ino

```
#include "src/FullModel/fullModel.h"
// Uključujemo datoteku koja sadrži deklaracije i definicije za klasu
'FullModel'
#include "src/Setup/SetupData.h"
// Uključujemo datoteku koja sadrži deklaracije i definicije za klasu
'SetupData'

void setup() {
// Funkcija 'setup' se izvršava samo jednom prilikom pokretanja ili
resetiranja uređaja
  delay(2000);
  // Kašnjenje od 2000 milisekundi (2 sekunde) kako bi se omogućilo da se
uređaj inicijalizira
  SetupData::runSetup();
}
void loop() {
  // Funkcija 'loop' se izvršava kontinuirano u petlji nakon 'setup' funkcije
  // Ova metoda se ponavlja stalno, dok uređaj radi, i obavlja zadatke koje je
definirala klasa
  SetupData::fullModel->loop();
}
```

PrivateData.h

```
#ifndef PRIVATE_DATA_H
#define PRIVATE_DATA_H

char* ssid = "Hranilica";
char* pass = "**************";

#define BLYNK_TEMPLATE_ID "*******"
#define BLYNK_TEMPLATE_NAME "******"
#define BLYNK_AUTH_TOKEN "C**********************"

#endif
```