Integracija robotskog sučelja osjetljivog na dodir

Šavrljuga, Demion

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Demion Šavrljuga

Zagreb, 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentori:

Izv. prof. dr. sc. Tomislav Stipančić, dipl. ing.

Student:

Demion Šavrljuga

Zagreb, 2024.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se mentoru dr. sc. Tomislavu Stipančiću na pruženim savjetima i financijskoj investiciji u ovaj rad.

Zahvaljujem se kolegi asistentu Leonu Korenu na pruženim savjetima i pomoći tokom izrade ovog rada.

Na kraju, posebnu zahvalu dobiva moj dugogodišnji prijatelj Aggin Etintrof iz Finske, zbog čijih ohrabrenja sam odlučio studirati ovo područje.

Demion Šavrljuga

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za završne i diplomske ispite studija mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 – 04 / 24 – 06 / 1**	
Ur.broj: **15 – 24 –**	

ZAVRŠNI ZADATAK

Student: **Demion Šavrljuga** JMBAG: **0035239902**

Naslov rada na hrvatskom jeziku: **Integracija robotskog sučelja osjetljivog na dodir**

Naslov rada na engleskom jeziku: **Integration of a robotic touch-screen interface**

Opis zadatka:

U radu je potrebno izraditi cjelovito rješenje komunikacijskog sučelja osjetljivog na dodir koje je dio sustava afektivnog robota PLEA čiji je prototip dostupan u sklopu Laboratorija za projektiranje izradbenih i montažnih sustava. S pomoću interakcijskog sučelja treba biti ostvaren izravan i intuitivan unos naredbi i parametara sustava od strane korisnika prema robotu. Razvijeno softversko i hardversko rješenje treba sadržavati:

- ekran osjetljiv na dodir s programskom podrškom interakcijskog sučelja koje sadrži grafičke elemente (virtualne tipke, klizače i druge elemente upravljanja) te koje prevodi dodire na ekranu u robotu shvatljive naredbe
- pozadinski sustav ili modul u robotu (hardverske komponente sustava) koji prima naredbe sa sučelja te izvršava akcije
- integracijski računalni sloj koji omogućuje komunikaciju između sučelja osjetljivog na dodir te pozadinskog sustava (npr. dio sloja za upravljanje podacima i sinkronizaciju podataka te dio sloja za upravljanje greškama).

Izvedeno hardversko i softversko rješenje treba biti izvedeno da omogućuje intuitivnu uporabu, učinkovitost s obzirom na integrirane funkcionalnosti, sigurnost u pogledu zaštite podataka, te da bude prilagodljivo s obzirom na ažuriranje softvera i dodavanje novih funkcionalnosti.

Razvijeno rješenje je potrebno eksperimentalno evaluirati. U radu je potrebno navesti korištenu literaturu te eventualno dobivenu pomoć.

Zadatak zadan: 24. 4. 2024.

Zadatak zadao: izv. prof. dr. sc. Tomislav Stipančić

Datum predaje rada:
2. rok (izvanredni): 11. 7. 2024.
3. rok: 19. i 20. 9. 2024.

Predviđeni datumi obrane:
2. rok (izvanredni): 15. 7. 2024.
3. rok: 23. 9. – 27. 9. 2024.

Predsjednik Povjerenstva: izv. prof. dr. sc. Petar Ćurković

SADRŽAJ

SADRŽAJ 1
POPIS SLIKA 3
POPIS OZNAKA 4
SAŽETAK 5
SUMMARY 6
1. UVOD 7
2. Grafika na sučelju 8
2.1. Rad grafike na sučelju 8
2.2. LovyanGFX 8
3. LVGL (Light and Versatile Embedded Graphics Library) 9
3.1. Struktura elemenata 10
3.2. Hijerarhija objekata 10
3.3. Widgeti 11
3.4. Style 12
3.5. Layers 12
3.6. Events 13
3.7. Flex flow 14
3.8. Alati za olakšavanje razvoja 15
3.8.1. SquareLine Studio 15
3.8.2. PC LVGL simulatori 16
4. BLE (Bluetooth Low Energy) 17
4.1. O protokolu 17
4.2. Način rada 17
4.3. Struktura protokola 17
4.4. Svojstva karakteristika 18
5. Odabir HMI hardvera 20
6. Radno okruženje 21
6.1. PlatformIO 21
6.2. Postavljanje biblioteka 21
6.2.1. LovyanGFX postavke 21
6.2.2. LVGL 8.3.6. 21
7. Osposobljavanje LVGL-a 22
7.1. LVGL handler 23
8. GUI (Graphical User Interface) 25
8.1. Općeniti dizajn sučelja 25
8.2. Connection_tab 26
8.3. Dizajn taba 26
8.4. Connection_tab dio programa 27
8.5. Mogućnosti nadogradnje 30

9. Dizajn BLE servisa 31
9.1. BLE inicijalizacija 31
9.2. BLE servisna inicijalizacija 33
9.2.1. Callback klasa za BLE_network_names_ch 35
9.2.2. Callback klasa za BLE_network_names_ch 36
10. Logika primanja podataka 37
10.1. Glavna petlja 37
10.1.1. Logika za primljeni string imena mreža 38
10.1.2. Logika za primljeni string poruke 38
11. Prikaz mrežnih podataka i odabir mreže 39
11.1. Procesiranje stringa mreža 39
11.2. Tablica mreža 40
11.3. Spajanje na mrežu 41
11.3.1. Upisivanje lozinke 44
11.3.2. „Connect“ i „Cancel“ gumbovi 45
11.3.3. Funkcija za spajanje na mrežu 46
12. Naredbe za klijenta 49
13. Poruke od klijenta 50
13.1. Proces primanja poruke 50
13.2. Procesiranje poruke 51
13.2.1. Prikaz statusa mrežne 53
13.2.2. Prikaz IP-eva i drugih poruka 53
14. Klijent 54
14.1. Spajanje klijenta na server 54
14.2. Procedura nakon spajanja 56
14.3. Spajanje na mrežu 57
14.4. Thread za slušanje notifikacija i slanje mrežnog statusa 59
14.4.1. Slanje statusa mrežne spojenosti HMI-u 60
14.5. Rad sa notifikacijama 62
14.5.1. Naredba „Search for networks“ 63
14.5.2. Naredba „Get IP“ 64
14.5.3. Naredba „Disconnect from networks“ 65
15. Zaključak 66
LITERATURA 67
PRILOZI 68

POPIS SLIKA

Slika 1. Prikaz rada HMI-a 7
Slika 2. LVGL [1] 9
Slika 3. Primjer hijerarhije jednog ekrana [2] 11
Slika 4. Slojevi prikaza na ekranu od LVGL-a 13
Slika 5. Primjer dizajna termostata u SquareLine Studiju [3] 15
Slika 6. Primjer LVGL simulatora u Visual Studiju [4] 16
Slika 7. BLE server-klijent komunikacija 18
Slika 8. Odabrani HMI hardverski modul [5] 20
Slika 9. Primjer tab strukture GUI-a 25
Slika 10. Željeni izgled connection taba 26
Slika 11. Primjer razmaka kod defaultnog objekta [6] 28
Slika 12. Dijagram približnog izgleda tablice 41
Slika 13. Željeni izgled popup-a za WiFi 43
Slika 14. Željeni izgled popup-a za WiFi 44
Slika 15. Tipkovnica za popup 45

POPIS OZNAKA

Oznaka	Opis
HMI	Human Machine Interface
API	Application Programming interface
BLE	Bluetooth Low Energy
LVGL	Light and Versatile Graphics Library
GFX	Graphics

SAŽETAK

Rad opisuje proces dizajna HMI sustava za interakciju s afektivnim robotom PLEA. Prolaze se područja od odabira hardvera, dizajna grafičkog sučelja, uspostavljanje komunikacijskih protokola, programske implementacije na HMI-u te izrada API-a sa strane robota. Za rješavanje ovih zadataka objašnjava se korištenje LVGL biblioteke, LovyanGFX biblioteke i BLE komunikacijskog protokola. U sklopu rada na HMI se dodaje funkcionalnost kontrole mrežne spojenosti na robotu te se time prikazuje proces dodavanja novih funkcionalnosti u ovaj sustav. Poanta rada je da primjerom dodavanja jedne funkcionalnosti, objasni proces nadogradnje novih funkcionalnosti na programsko i hardversko rješenje izvedeno u ovom radu.

Ključne riječi: HMI, LVGL, BLE, programiranje, dizajn

SUMMARY

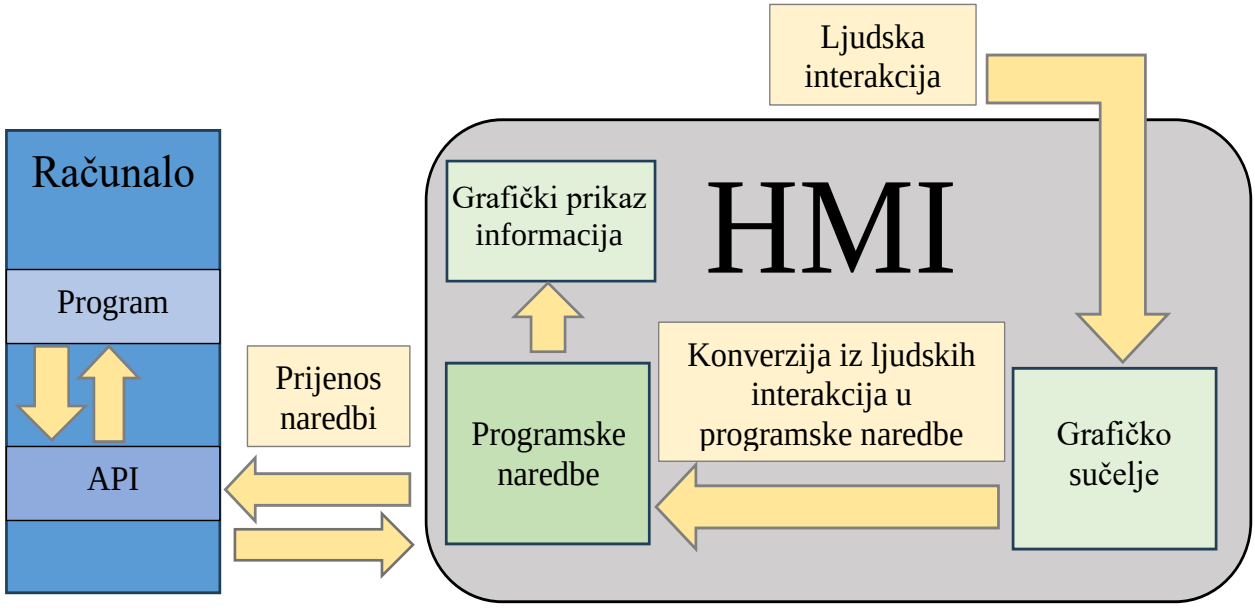
The paper describes the design process of an HMI system for interaction with the affective robot PLEA. It covers areas such as hardware selection, graphical interface design, establishment of communication protocols, software implementation on the HMI, and the development of an API on the robot's side. To accomplish these tasks, the use of the LVGL library, LovyanGFX library, and BLE communication protocol is explained. As part of the work, functionality for controlling the network connectivity of the robot is added to the HMI, demonstrating the process of adding new functionalities to this system. The purpose of the paper is to illustrate the process of upgrading new functionalities on the software and hardware solution implemented in this work by using the example of adding a single functionality.

Key words: HMI, LVGL, BLE, programming, design

.

1. UVOD

Problem komunikacije između korisnika i uređaja oduvijek je predstavljao izazov koji stvara prepreke u pristupačnosti i efikasnosti. Potreba za boljim načinom komunikacije s uređajem danas se rješava na mnogo različitih načina, od terminalnih sučelja do glasovnih komandi. U ovom radu prikazuje se primjer rješavanja ovog problema integracijom robotskog sučelja osjetljivog na dodir, odnosno HMI-a (Human-Machine Interface). Human-Machine Interface (HMI) je tehnologija koja omogućava interakciju između ljudi i strojeva. Danas su HMI sustavi prisutni u širokom rasponu primjena, od automatizacije u industriji do potrošačke elektronike i automobilske industrije. Postoji više vrsta HMI-a, no najčešće se pod tim pojmom misli na grafičko sučelje (GUI) kojim se upravlja putem tipki ili putem dodirnog zaslona. Prednost HMI-a leži u mogućnosti izrade sučelja koja su intuitivnija za korisnike u usporedbi s osnovnim načinima komunikacije s računalima, poput terminala. HMI sustavi također omogućuju nadzor i upravljanje složenim procesima u stvarnom vremenu, pružajući korisnicima mogućnost praćenja performansi, dijagnostike i upravljanja ključnim funkcijama sustava. U modernim HMI-ima često se implementiraju dodatne značajke kao što su vizualizacija podataka, alarmi i notifikacije, što dodatno poboljšava efikasnost i sigurnost u radu. Većina modernih HMI sustava funkcionira na sličan način: imaju programski sloj koji se bavi grafikom za korisnika, sloj koji konvertira korisnikove fizičke naredbe u programske naredbe, te sloj koji prenosi te programske naredbe računalu za koje je HMI namijenjen. Računalo koje koristi HMI mora imati implementiran odgovarajući API za primanje naredba od HMI-a.



Slika 1. Prikaz rada HMI-a

2. Grafika na sučelju

2.1. Rad grafike na sučelju

Ekran se sastoji od skupa piksela koji su namijenjeni prikazivanju grafika. Grafike na ekranu prikazuju se mijenjanjem boje piksela. Iako je ekran skupina vrlo velikog broja piksela, to ne znači da se programski mora mijenjati boja svakog piksela manualno. Zato postoje GFX biblioteke (GFX – skraćeno za grafike) koje se izravno bave interakcijom između programa i hardvera koji kontrolira ekran, pružajući nivo apstrakcije koji olakšava rad s grafikama na ekranu. GFX biblioteke obično imaju modularnu arhitekturu koja omogućuje prilagodbu različitim vrstama zaslona i mikrokontrolera. One omogućuju jednostavno crtanje osnovnih oblika na ekranu, poput crta, pravokutnika, krugova, teksta i slika, često putem jednostavnog API-a (Application Programming Interface). GFX biblioteke pojednostavljuju kompleksne operacije crtanja, omogućujući programerima da se fokusiraju na dizajn.

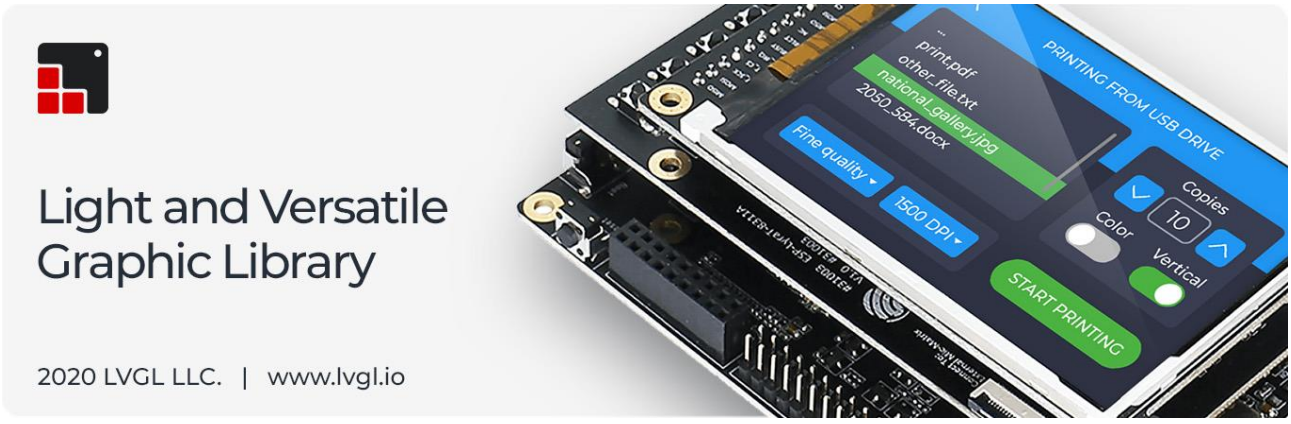
2.2. LovyanGFX

Za potrebe ovog HMI-a odabrana je LovyanGFX biblioteka, jer je dovoljno brza, relativno jednostavna za korištenje te podržava ekran odabran za ovaj projekt. LovyanGFX, osim mogućnosti crtanja po ekranu, omogućava čitanje pozicija dodira ako je ekran touchscreen. Ako ekran nije među onima koje biblioteka već podržava, potrebno je izraditi driver za taj ekran koji bi biblioteka koristila. Kako bi se LovyanGFX mogao koristiti na odabranom ekranu, potrebno je u konfiguracijskom fajlu biblioteke odkomentirati vrstu ekrana te u samom programu dodati nekoliko osnovnih naredbi kako bi se biblioteka pravilno konfigurirala za taj ekran:

```
#include "gfx_conf.h"
static LGFX_Sprite sprite(&gfx);
gfx.begin();            // započinje LovyanGFX
gfx.setRotation(1);     // okrenutost ekrana
gfx.setBrightnes(255);  // svjetlina ekrana
gfx.setColorDepth(16);  // bojna rezolucija piksela
```

3. LVGL (Light and Versatile Embedded Graphics Library)

LVGL (Light and Versatile Graphics Library)[1] je open-source grafička biblioteka dizajnirana za embedded sustave, s fokusom na učinkovitost, fleksibilnost i malu potrošnju resursa (RAM i memorije). LVGL je dizajnirana da bude neovisna o hardveru te da se može koristiti na bilo kojem embedded sistemu(ako zadovoljava neke osnovne tehničke specifikacije). Iako se ova biblioteka zove „Graphics Library“, ona nije ista vrsta biblioteke kao i LovyanGFX. LovyanGFX je biblioteka koja se bavi interakcijom između hardvera i softvera, dok LVGL omogućava laku izradu UI-a s već napravljenim widgetima(gumbi, slajderi, tipkovnice...) koji se lagano mogu modificirati. LVGL se nadovezuje na GFX biblioteku te koristi njene sposobnosti interakcije s hardverom. LVGL svaki frame crta ono što mu je zadano kao UI te ga šalje GFX biblioteci da crta kao sliku. LVGL postiže dobre performanse korištenjem buffera te tako optimizira broj frejmova po sekundi, ali zato je intenzivniji na RAM po tome što je ekran veći (po broju piksela). Na kraju je LovyanGFX potreban samo da može proslijediti LVGL-u funkciju za pisanje po ekranu te funkciju koja daje koordinate dodira na ekranu. LVGL podržava razne platforme mikrokontrolera poput ARM Cortex-M serije, ESP32, STM32, i mnoge druge. Također, podržava različite zaslone, od jednostavnih monokromatskih ekrana do složenih TFT zaslona s visokom rezolucijom. LVGL isto tako podržava rad u različitim operativnim sustavima i platformama, uključujući FreeRTOS, Zephyr, i RT-Thread, a osim toga može raditi i u samostalnom modu(rad bez operativnog sustava), što dodatno povećava njegovu fleksibilnost. LVGL je dosta zrela biblioteka s puno verzija, u ovom projektu je korištena verzija 8.3.6. te će izbor te verzije biti objašnjenu u kasnijem potpoglavlju.



Slika 2. LVGL [1]

3.1. Struktura elemenata

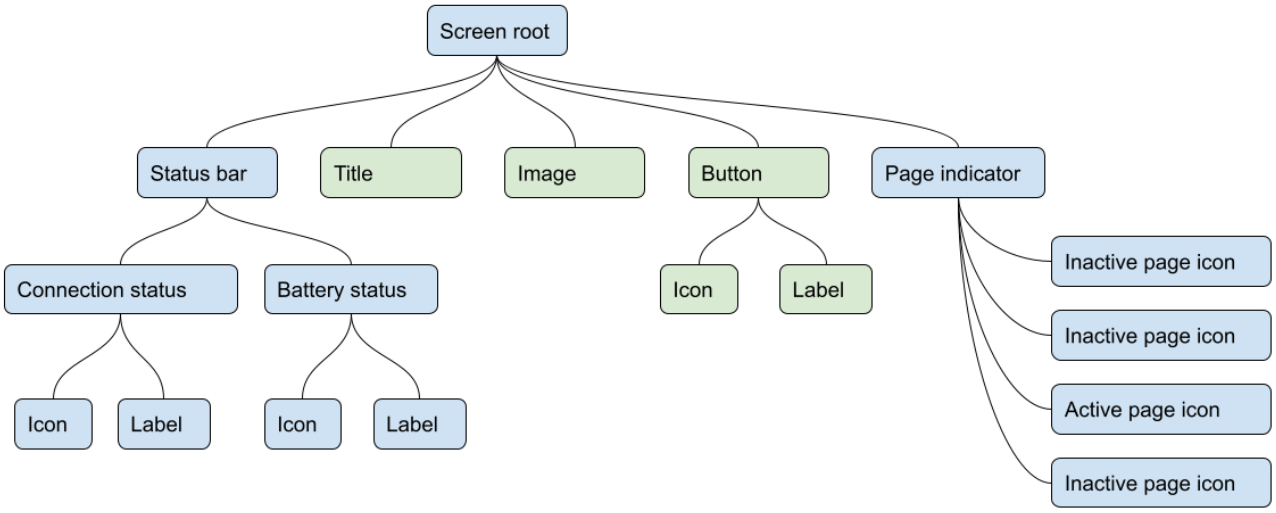
LVGL je baziran na hijerarhijskoj strukturi objekata. UI se sastoji od stabla objekata, gdje svaki objekt ima svojeg roditelja. Korijenski objekt je „screen“, to je posebna vrsta objekta koja nema roditelja i kao što mu ime govori namijenjen je da predstavlja jedan puni ekran (ne ekran u smislu fizičkog ekrana, već zasebnu jedinicu, npr. alegorično se može gledati na screen-ove poput kanala na televiziji, kada je daljinski pritisnut televizor se prebacuje na drugi ekran). Na „screen“ se stavljaju svi drugi objekti te je on najstariji roditelj. Objekti koji su već napravljeni za neku čestu funkciju zovu se „widget“, to su stvari poput gumbi, slajdera, tipkovnica, indikatora, kalendara, slika, itd. Kada se widget stvori mora mu se deklarirati roditelj, što može biti ili screen ili neki drugi objekt(widget). Moguće je napraviti više ekrana koji svaki ima svoje različite elemente te pozivanjem funkcije lv_scr_load(željeni_screen), može se mijenjati između toga koji ekran se prikazuje.

Primjer gumba na ekranu:

```
lv_obj_t* default_screen;                   // deklaracija objekta
lv_obj_t* button;                           // deklaracija objekta
default_screen = lv_obj_create(NULL);       // objekt je screen, bez roditelja
button = lv_btn_create(default_screen);     // objekt je gumb, roditelj je screen
```

3.2. Hijerarhija objekata

Poanta ove hijerarhije je to što utjecanjem na roditelja se utječe na svu njegovu djecu. Pozicija objekta se određuje ili x, y koordinatama ili ključnim riječima u funkciji za poziciji (sredina, lijevo gore, desno dole, itd.). Pozicija djeteta je uvijek relativna na roditelja, tako da ako se pomakne roditelja dijete se automatski pomiče s njime. (npr. ako je na ekranu gumb s tekstom na sebi koji je centriran na gumbu, pomicanjem gumba tekst na gumbu se automatski pomiče s njime, tako da će u novoj poziciji i dalje biti u centru gumba). Brisanjem roditelja brišu se sva njegova djeca [2](a brisanjem djece brišu se i njihova djeca, itd.). Isto tako svi objekti uz svoju poziciju imaju i svoju veličinu (širinu i visinu). Veličina objekta može biti čvrsto zadana, ali može i biti relativna na roditelja, te ako je tako relativno zadana, skaliranjem roditelja se skalira i dijete. To olakšava ne samo lakše mijenjanje UI-a, već i mogućnost rađenja UI-a koji sam sebe skalira ovisno o veličini ekrana, tako da je moguća izrada programa koji se može koristiti na ekranima različitih veličina, bez dodatne modifikacije.



Slika 3. Primjer hijerarhije jednog ekrana [2]

Velika većina widgeta se ne može nalaziti izvan granica roditelja već moraju biti unutar granica roditelja. Ako dijete prijeđe neku od granica roditelja, ono neće izaći izvan granica roditelja već će postati „scrollable“, što znači da će se na roditelju pokazivati samo dio djeteta, a povlačenjem prsta po tom djetetu će ga scrollati te pomicati na druge dijelove koje se trenutno ne vide.

3.3. Widgeti

Widgeti su osnovna prednost LVGL-a naspram korištenja samo GFX biblioteke za manualno crtanje. To su česti grafički elementi koji dolaze s LVGL bibliotekom. Oni predstavljaju vizualne i interaktivne komponente, poput gumbi, tekstualnih polja, slika, klizača i sličnih elemenata.. Widgeti su napravljeni tako da se mogu lagano podešavati kako bi mogli pasati bilo kojim dizajnerskim temama i potrebama. Widgeti su inače rađeni od više osnovnih objekata iz biblioteke, što omogućuje podešavanje samo dijela widgeta (npr. kod tipkovnice svaki gumb je dodatni podwidget). Svi widgeti se kreiraju na isti način:

```
// Općenita deklaracija:
lv_obj_t* željeni_widget;                              // deklaracija objekta
željeni_widget = lv_(vrst widgeta)_create(roditelj);   // općenita deklaracija

// Primjeri nekih deklaracija widgeta:
lv_obj_t* gumb;
gumb = lv_btn_create(roditelj);                        // objekt je gumb

lv_obj_t* slajder;
slajder = lv_slider_create(roditelj);                  // objekt je slajder
```

3.4. Style

„Style“, to jest stilovi, su LVGL-ov način za podešavanje widgeta. Korištenjem stilova se kontroliraju ponašanje i izgled widgeta. Stilom se mogu odrediti svojstva objekta poput boje, oblika, fontova, sjenčanja, animacije, oblika, itd. Programski gledano, stil je struktura podataka koja se koristi za pohranu različitih svojstva koje se mogu pridati widgetu. Još jedna velika prednost stilova je to što se stil može jednom napraviti te pridati više widgeta, umjesto da se svakom widgetu posebno zadaju ista svojstva. Svaki widget u LVGL-u može imati više stilova, od kojih svaki može biti primijenjen na različite dijelove widgeta, kao što su pozadina, rubovi, tekst i slike. Stilovi se kreiraju i konfiguriraju s pomoću „lv_style_t“ objekta, nakon čega se primjenjuju na widgete s pomoću funkcije „lv_obj_add_style()“. Stilovi također podržavaju „pseudo-stanja“, što omogućava promjenu izgleda widgeta ovisno o njegovom trenutnom stanju. (npr. moguće je napraviti da gumb bude drugačije boje kada je pritisnut i otpušten).

Primjer korištenja stila.

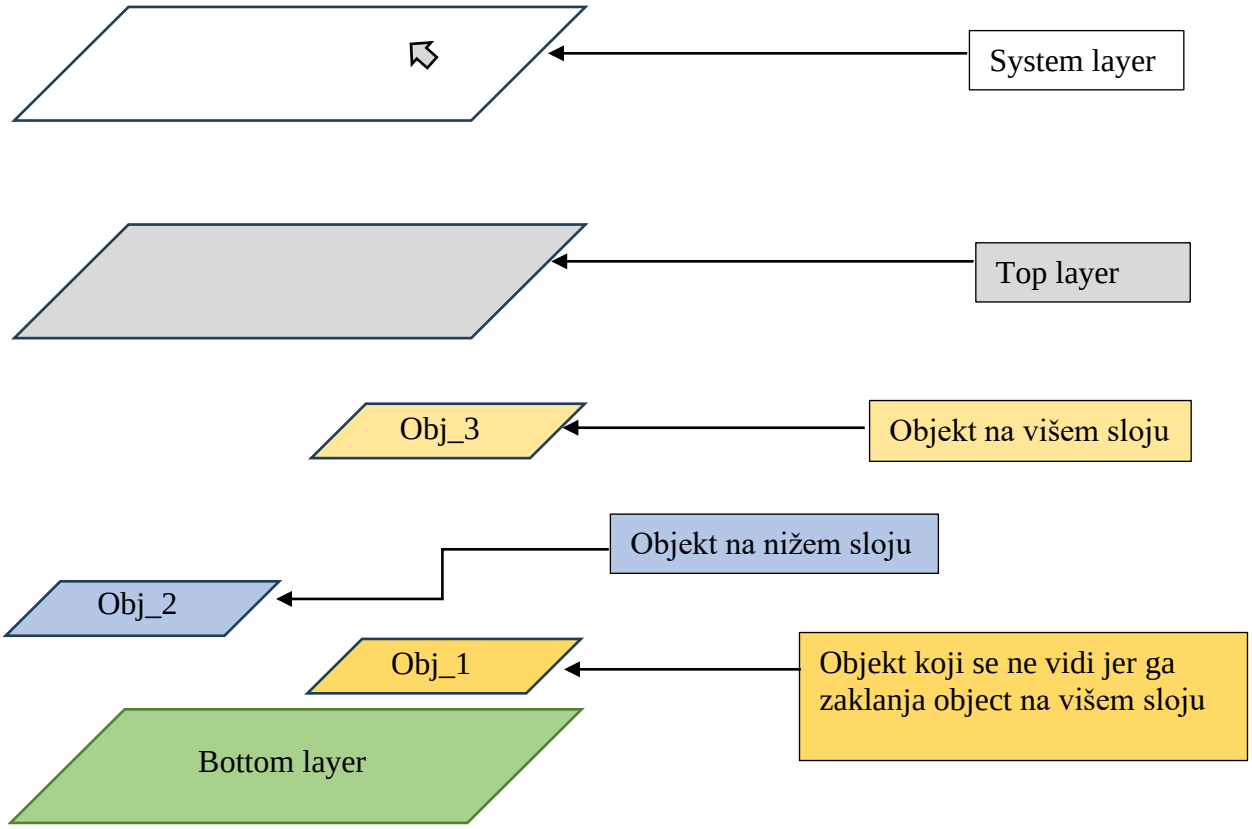
```
// Kreiranje novog stila
lv_style_t style;           // Kreiranje stila
lv_style_init(&style);      // Stil je potrebno inicijalizirati

// Definiranje osnovnih svojstava stila
lv_style_set_bg_color(&style, lv_color_hex(0xFF0000));      // Pozadine je crvena
lv_style_set_radius(&style, 10);                            // Zaobljenost kutova
lv_style_set_border_width(&style, 2);                       // Debljina obruba
lv_style_set_border_color(&style, lv_color_hex(0x000000));  // Crna boja obruba
```

3.5. Layers

„Layers“, to jest slojevi, su funkcionalnost u LVGL-u koja omogućava upravljanje višestrukim razinama grafičkog prikaza unutar korisničkog sučelja. Svaki sloj je površina po kojoj se mogu crtati widgeti, tekst ili neke druge grafike. Slojevi mogu prekrivati jedni druge, što omogućava parcijalno zakrivanje određenih elemenata. Kada se napravi neki objekt, on automatski ide u prvi viši sloj nego u kojemu je roditelja, ali sloj na kojemu se objekt nalazi se može i manualno mijenjati. Izvedba ovakve slojne hijerarhije omogućava fleksibilnost upravljanja prikaza, to miče potrebu razmišljanja o tome koji objekt se crta preko kojega, jer je to određeno slojevima. Slojevi su posebno korisni za implementaciju efekata poput prozirnosti, gdje gornji slojevi mogu djelomično otkrivati ono što se nalazi ispod njih, kao i za implementaciju dinamičkih sučelja gdje različiti slojevi mogu biti prikazani ili skriveni prema potrebi. Postoje i dva posebna sloja koja su „top_layer“ i „sys_layer“. Top layer je sloj u koji se stavljaju stvari za koje se želi da budu iznad svega drugoga, to je najčešće za nekakve

popup-ove, notifikacije ili tipkovnice. Iako bi „top_layer“ po imenu trebao biti najviši u hijerarhiji, zapravo najviši sloj je „sys_layer“, to je sloj namijenjen za stvari poput kursora miša ili sistemskih poruka, to jest stvari koje su najvažnije da uvijek budu na vrhu. Da bi se objekt stavio u određeni sloj, može mu se pri deklaraciji umjesto nekog widget ili screen roditelja samo staviti ime tog sloja kao roditelja.

Hijerarhija slojeva:



Slika 4. Slojevi prikaza na ekranu od LVGL-a

3.6. Events

Kako bi program mogao reagirati na interakciju s widgetima, u LVGL-u postoje „events“, to jest eventi. Kada god se nešto promijeni s widgetom, on generira event, a event sam po sebi je kod koji govori što se s widgetom dogodilo. Svaki widget može generirati različite vrste eventa, poput pritiska na gumb, pomicanja klizača, unosa teksta ili promjene veličine. Eventi mogu biti uzrokovani korisničkom interakcijom, ali neki eventi i s nekim drugim procesima (npr. repozicioniranje widgeta daje event). Eventi zapravo omogućavaju strukturu u kodu: „kada se nešto dogodi, napravi nešto“.

Da bi događanje određenog eventa uzrokovalo pozivanjem neke funkcije potrebno je povezati widget s „callback“ funkcijom pomoću „lv_obj_add_event_cb(widget, callback, EVENT, user_data)“. Callback funkciji je moguće proslijediti i „user_data“, ako je potrebno u callback-u. Jedan widget može imati više od jednog callbacka, gdje je svaki postavljen za jednu vrstu eventa, te tako raditi različite stvari ovisno o vrsti interakcije (npr. gumb se drži stisnutim i gumb je kliknut). Isto tako moguće je napraviti i da se u jedan callback može ući s više evenata tako da se koristi „or“ operator za pridodavanja dodatnih evenata u „lv_obj_add_event_cb()“ (npr. LV_EVENT_PRESSED | LV_EVENT_CLICKED). Ovaj sistem eventova i callbacka omogućava fleksibilno i responzivno korisničko sučelje, gdje aplikacija može dinamično reagirati na širok spektar interakcija i promjena, prilagođavajući svoje ponašanje u stvarnom vremenu. Korištenjem evenata, programeri mogu učinkovito upravljati složenim interakcijama.

Primjer eventa gdje se mijenja tekst na gumbu kada se klikne

```
lv_obj_t * btn = lv_btn_create(lv_scr_act());       // Kreiranje gumba
// Dodavanje labele unutar gumba sa tekstom
lv_obj_t * label = lv_label_create(btn);
lv_label_set_text(label, "Klikni me!");

lv_obj_add_event_cb(btn, event_handler, LV_EVENT_CLICKED, NULL); // Povezivanje
event handlera s klik eventom

// Callback funkcija koja se poziva kada se gumb klikne
void event_handler(lv_event_t * e) {
    lv_label_set_text(label, "Bio sam kliknut!");    // Promjena teksta na gumbu
}
```

3.7. Flex flow

Flex flow u LVGL-u je rasporedni mehanizam koji omogućava dinamičko postavljanje i upravljanje položajem widgeta unutar kontejnera koristeći fleksibilno poravnanje. Sličan je CSS Flexbox modelu u web programiranju, gdje se elementi mogu automatski rasporediti vodoravno ili okomito unutar kontejnera, a da se ne mora ručno postavljati svaki widget. Ovaj mehanizam je posebno koristan kada želite da vaši elementi budu responzivni, odnosno da se automatski prilagođavaju veličini i obliku kontejnera ili ekrana. Flex flow se postavlja na objekt ili widget pomoću „lv_obj_set_flex_flow()“ funkcije, flex flow može raditi kao stupac ili redak ovisno o zadanim parametrima.

3.8. Alati za olakšavanje razvoja

Dizajn i programiranje HMI-a je često iterativni proces, te kod čistog programiranja je problem to što se rezultat promjena u kodu ne vidi sve dok se program ne pošalje. Za ove probleme u razvijanju osmišljeni su alati za pomoć. U ovom radu se nisu koristili ovi alati jer su potrebe dizajna sučelja bile relativno jednostavne, ali je odlučeno samo opisno proći kroz ova dva softvera u pod točkama ispod, jer je jedan od zadataka ovog rada bio demonstracija razvoja sučelja te bi ovi softveri mogli biti od velike pomoći u nadogradnji novih funkcija na ovaj HMI ili u nekim drugim projektima.

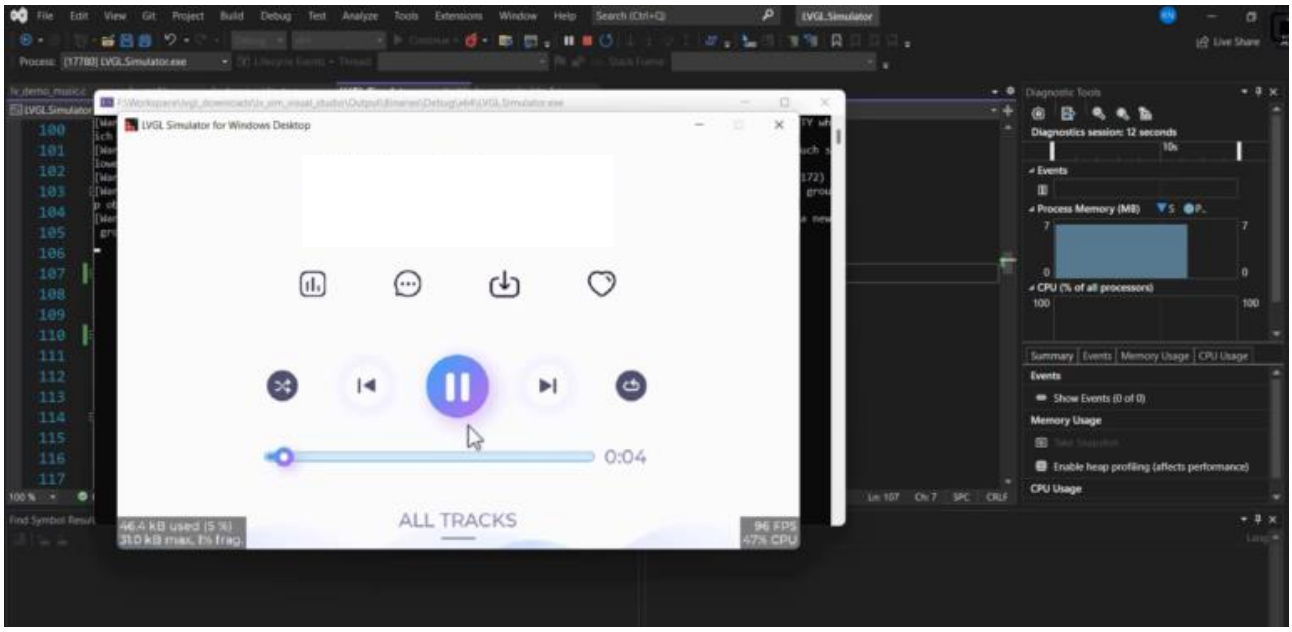
3.8.1. SquareLine Studio

SquareLine Studio je moćan grafički alat za razvoj korisničkog sučelja (HMI) temeljenog na LVGL biblioteci. Ovaj alat omogućava dizajnerima i programerima da vizualno kreiraju sučelja bez potrebe za pisanjem velikih količina koda. Korištenjem „drag-and-drop“ elemenata, korisnici mogu lako dodavati widgete, definirati njihove stilove, te ih pozicionirati unutar različitih ekrana. SquareLine Studio također omogućava stvaranje interakcija između različitih elemenata putem eventa, što olakšava definiranje ponašanja sučelja bez potrebe za direktnim kodiranjem. Nakon što je sučelje dizajnirano, SquareLine Studio daje opciju generiranja koda, tako da izlaz dizajna sučelja u SquareLine Studiju je kod koji se može poslati na fizički HMI te tako dobiti isto sučelje (primjer jednog takvog sučelja termostata [3]). Ovo značajno ubrzava proces razvoja, jer promjene u dizajnu mogu odmah biti vizualizirane, a generirani kod se može direktno integrirati u projekt. Važno je napomenuti da SquareLine Studio podržava LVGL biblioteku samo do verzije 8.3.6. te je zato ta verzija LVGL-a korištena u ovom radu kako bi dopustila korištenje SquareLine Studija za lakše dodavanje novih funkcionalnosti HMI-u.

Slika 5. Primjer dizajna termostata u SquareLine Studiju [3]

3.8.2. PC LVGL simulatori

PC LVGL simulatori su alati koji omogućavaju simulaciju „LVGL-based“ sučelja na osobnom računalu, bez potrebe za učitavanjem koda na stvarni uređaj. Razvojni tim LVGL-a je napravio simulatore koji su nadogradnje na IDE-eve te simulatori pokazuju u zasebnom prozoru IDE-a izgled GUI-a koji će biti generiran od strane koda u IDE-u. Simulatori trenutno postoje za par najpopularnijih standardnih IDE-a poput VSCode-a, Visual Studija[4], Eclipse-a i drugih. Simulatori omogućavaju testiranje i debugiranje sučelja u stvarnom vremenu, što je izuzetno korisno u ranim fazama razvoja, jer omogućava brzu iteraciju i otkrivanje grešaka prije nego što se kod prenese na hardver. Ovi simulatori podržavaju sve funkcionalnosti LVGL biblioteke, uključujući widgete, stilove, animacije i evente, omogućavajući potpunu simulaciju korisničkog sučelja. Također, omogućavaju razvijanje i testiranje različitih scenarija interakcije bez potrebe za fizičkim uređajem, što može značajno ubrzati razvojni proces. Iako nije bio potreban u ovom radu, LVGL simulator može biti izuzetno koristan alat u kompleksnijim projektima, omogućavajući efikasniji razvoj i testiranje HMI sustava.



Slika 6. Primjer LVGL simulatora u Visual Studiju [4]

4. BLE (Bluetooth Low Energy)

4.1. O protokolu

Komunikacija između HMI-a i robota može se uspostaviti preko fizičke konekcije ili preko bežičnih protokola poput Wi-Fi, Bluetooth Classic ili BLE-a. Bluetooth je za te potrebe prilično dobar odabir, s efektivnim dometom od oko 10 metara i brzom komunikacijom. Bluetooth Classic je namijenjen za kontinuirano slanje podataka (npr. Bluetooth slušalice), dok je Bluetooth Low Energy (BLE) namijenjen za brze i kratke prijenose informacija. BLE se često koristi u uređajima kao što su nosivi uređaji, senzori, pametni kućni uređaji i IoT uređaji, gdje je ključna niska potrošnja energije uz održavanje zadovoljavajućih performansi u prijenosu podataka. Budući da za zahtjeve ovog HMI-a nije potrebno kontinuirano slanje podataka, koristit će se BLE za cijeli protokol komunikacije. Također, budući da je BLE dizajniran za malu potrošnju energije, to će omogućiti dulje trajanje baterije HMI-a.

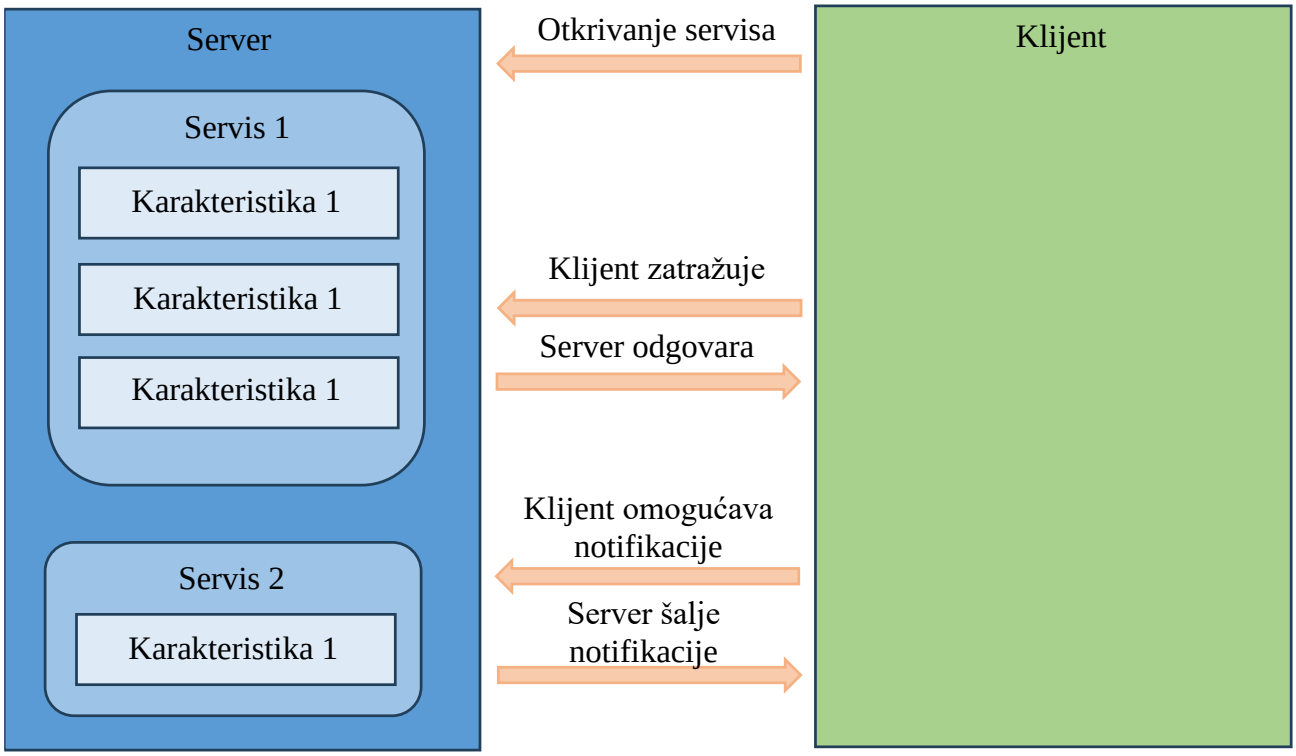
4.2. Način rada

BLE poput i klasičnog Bluetooth-a je bežični protokol koji radi u rasponu od 2.400 do 2.4835 GHz, ali se razlikuje u načinu prijenosa podataka, zbog čega je efikasniji u pogledu potrošnje energije. BLE koristi tehnologiju nazvanu „frequency hopping spread spectrum“ (FHSS), koja omogućava stabilniji prijenos podataka u okruženjima s visokom razinom elektromagnetske interferencije. Standard BLE-a omogućava brze prijenose podataka, s brzinama prijenosa do 2 Mbps (u verziji 5.0 i kasnijima), dok istovremeno omogućava uređajima da veći dio vremena ostanu u stanju mirovanja kako bi se štedjela energija.

4.3. Struktura protokola

BLE komunikacija koristi server-klijent sistem, pri čemu je server uređaj koji kontrolira komunikaciju i pohranjuje informacije koje klijent uređaj zatražuje. Server može biti povezan s više klijenata, a BLE protokol omogućava različite načine rada, kao što su broadcast način rada, koji omogućuje slanje podataka svim uređajima unutar dometa, i connection-oriented način rada, koji omogućava stabilniji i sigurniji prijenos podataka između dva uređaja. Slanje i primanje podataka u protokolu određeno je posebnom strukturom. Svi podaci u skupu sličnih informacija organizirani su u servise, a servis se sastoji od jedne ili više karakteristika koje djeluju kao kanali komunikacije (npr. za sportske satove može postojati servis „tjelesne informacije“ koji sadrži karakteristike poput „BPM“, „Krvni kisik“, „Tjelesna temperatura“ itd.). Klijent ima mogućnost povezivanja na servise i

karakteristike koje su mu potrebne, tako da koristi sa servera samo ono što želi. Ova struktura postoji kako bi se standardizirale komunikacije među proizvodima iste vrste. Svaki servis i svaka karakteristika imaju svoju jedinstvenu adresu, koja se zove UUID (npr. bc2f672f-3008-4d4b-9e22-baf0004d3a56), a postoje i adrese servisa koje su rezervirane za određene vrste uređaja (npr. svi proizvedeni BLE termostati su napravljeni da se povezuju na isti UUID servisa za termostate, te svi termostati imaju iste standardizirane karakteristike na koje se spajaju). To omogućava da se svi uređaji iste vrste povezuju na isti način i da su međusobno potpuno zamjenjivi. Bez obzira na to je li heart rate monitor povezan s mobilnim uređajem proizveden od jednog ili drugog proizvođača, zbog standardizacije mobilni uređaj neće imati problema s interakcijom.



Slika 7. BLE server-klijent komunikacija

4.4. Svojstva karakteristika

Karakteristike se mogu gledati kao kanali komunikacije, karakteristika je zapravo memorijska adresa koja ukazuje na memorijsku lokaciju koja je veličine određenog broja bajtova(inače je 20). Na tu memorijsku lokaciju se stavljaju informacije koje se žele slati ili primati. Server određuje kako se mogu koristiti informacije iz karakteristike tako da im pridaje svojstva. Moguća svojstva koja karakteristika može imati su:

- „read“ - iz karakteristike se može čitati

- „write“ - u karakteristiku se može upisivati
- „broadcast“ - omogućava slanje prema više klijenata odjednom
- „write without response“ - u karakteristiku se može upisivati bez odziva
- „notify“ - karakteristika koristi notifikacije (u toku programa se omogućuje i onemogućuje uz pomoć deskriptora kako bi smanjili potrošnju struje)
- „indicate“ - omogućava notifikacije za koje se treba potvrditi primanje
- „authenticated signed writes“ – omogućava upisivanje signed vrijednosti u karakteristiku
- „extended properties“ - omogućava druga svojstva koja se moraju dodatno definirati

5. Odabir HMI hardvera

Odabir fizičkog hardvera koje će služiti kao HMI uvijek treba biti orijentirano tako da što bolje zadovolji potrebe korisnika (npr. ako se radi o nekim teškim uvjetima rada, preferiraju se ili fizički gumbi ili otpornički touchscreen kao sustav interakcije s ljudima). Zahtjevi za ovaj HMI su bili: TFT touchscreen, minimalno 6 inča dijagonalna veličina displeja, WiFi ili Bluetooth mogućnost komunikacije i pristupačna cijena. Ovi zahtjevi otvaraju dva moguća puta razvoja, pronalazak zasebnih mikrokontrolera, ekrana i ostalog hardvera ili pronalazak već gotovog postojećeg rješenja na tržištu. Razvojem vlastitog hardverskog rješenja se ide kada tržište ne daje ponude koje zadovoljavaju tražene uvjete, no nakon istraživanja tržišta ustvrđeno je da postoje zadovoljavajuća rješenja te nakon selekcije odlučeno je koje će se komercijalno dostupan proizvod koristiti. Odabran je proizvod dizajniran da bude gotovo hardversko rješenje za HMI-e, proizvod je Elecrow-ov TFT 7 inčni displej[5] na bazi ESP32 mikrokontrolera, zvan „CrowPanel 7.0". Ovaj displej je odabran jer zadovoljava sve zatražene uvjete (veličina, TFT, WiFi i Bluetooth 5.0 te je vrlo pristupačne cijene) uz to ima i na sebi integrirani BMS(Battery Management System), port za micro SD karticu i port za zvučnik. Ovo rješenje je omogućilo fokusiranje razvoja samo na programiranju jer je cijeli hardverski dio u ovom proizvodu riješen. Zadnji važan razlog za odabir ovog proizvoda bila je dobra dokumentacija, proizvođač je napravio dokumentaciju za rad s ovim proizvodom u najčešćim frameworkovima(Arduino, Espressif IDF, Micropython) te za osposobljavanje i korištenje LVGL-a s ovim proizvodom.

Slika 8. Odabrani HMI hardverski modul [5]

6. Radno okruženje

6.1. PlatformIO

Za svrhu ovog projekta korišten je PlatformIO dodatak na VScode IDE za rad s fajlovima i bibliotekama. PlatformIO je višestruko platformski, višestruko arhitekturni, višestruki framework, profesionalni alat za „embedded“ sustave. U ovom projektu PlatformIO je korišten putem ekstenzije za VScode. Kao što je prije navedeno „CrowPanel 7.0“ podržava više framework-ova, ovdje je izabrano koristiti Arduino framework jer se proizvođačeva dokumentacija najviše bazira na Arduino framework-u, ali sami framework koji se koristi nije toliko važan jer se velika većina projekta svodi na rad s LVGL-bibliotekom koja je univerzalna po framework-ovima.

6.2. Postavljanje biblioteka

Biblioteke korištene u ovom projektu su:

- LVGL 8.3.6.
- LovyanGFX
- regex
- BLEDevice
- BLEServer
- BLEUtils
- BLE2902

Od ovih biblioteka potrebno je prilagoditi LVGL 8.3.6. i LovyanGFX kako bi mogle raditi sa izabranim ekranom.

6.2.1. LovyanGFX postavke

GFX biblioteka koja je korištena u ovom projektu podržava brojne postojeće ekrane, ali ne i „CrowPanel 7.0“ te je zato potrebno dodati dodatan fajl koji će biti driver za ekran. Proizvođač nudi već napravljeni driver za ekran putem cpp fajla koji se može skinuti.

6.2.2. LVGL 8.3.6.

LVGL neće raditi sam po sebi kada se doda u projekt, potrebno je napraviti promjene u konfiguracijskom fajlu. Potrebno je konfigurirati postavke poput: memory management opcije, clang formata, izvor tikova itd. Sve promjene potrebne za napraviti su dokumentirane sa strane proizvođača.

7. Osposobljavanje LVGL-a

Kao što je prije bilo rečeno, LVGL koristi postojeću GFX biblioteku te preko nje crta po ekranu i čita dodire touchscreena, da bi to moglo raditi potrebno je LVGL-u dati dvije funkcije.

Funkciju za čitanje dodira touchscreena:

```
void touchpad_read(lv_indev_drv_t *indev_driver, lv_indev_data_t *data)
{
    // This function gives LVGL the touch cordinates
    uint16_t touchX, touchY;
    bool touched = gfx.getTouch(&touchX, &touchY);
    if (!touched)
    {
        data->state = LV_INDEV_STATE_REL;
    }
    else
    {
        // set the coordinates
        data->state = LV_INDEV_STATE_PR;
        data->point.x = touchX;
        data->point.y = touchY;
    }
}
```

Funkcija za crtanje po ekranu:

```
void disp_flush(lv_disp_drv_t *disp, const lv_area_t *area, lv_color_t *color_p)
{
    // This function is used by LVGL to push graphics to the screen
    uint32_t w = (area->x2 - area->x1 + 1);
    uint32_t h = (area->y2 - area->y1 + 1);
    gfx.pushImageDMA(area->x1, area->y1, w, h, (lgfx::rgb565_t *)&color_p->full);
    lv_disp_flush_ready(disp);
}
```

Te dvije funkcije omogućavaju rad LVGL-a, ali za osposobljavanje LVGL-a potrebno je još definirati veličinu buffera koji će LVGL koristiti za driver za displej te definirati touchscreen driver za dodire.

```
void init_LVGL()  // Function that sets up LVGL
   // gfx setup
   pinMode(BL_PIN, OUTPUT);   // backlight initialization
   digitalWrite(BL_PIN, 1);   //
   gfx.begin();               // start the LovyanGFX
   gfx.fillScreen(TFT_BLACK); //
    //
```

```
    lv_init(); // LVGL initialize

    // display setup
    lv_disp_draw_buf_init(&draw_buf,disp_draw_buf1,NULL,screenWidth*screenHeight/8);
    lv_disp_drv_init(&disp_drv); // display driver initilize
    disp_drv.hor_res = screenWidth;
    disp_drv.ver_res = screenHeight;
    disp_drv.flush_cb = disp_flush; // flush function for LVGL
    disp_drv.full_refresh = 1;
    disp_drv.draw_buf = &draw_buf;
    lv_disp_drv_register(&disp_drv);

    // touchscreen setup
    static lv_indev_drv_t indev_drv;
    lv_indev_drv_init(&indev_drv);
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    indev_drv.read_cb = touchpad_read; // touchpad reed function
    lv_indev_drv_register(&indev_drv);
}
```

Važno je za napomenuti da LVGL koristi buffere kako bi se moglo pretprocesirati idući frame-ovi, to znači da ako se LVGL-u da veći buffer potencijalno će biti brži te će ekran imati veći frame rate. U funkciji lv_disp_draw_buf_init() gdje je stavljeno NULL, može se staviti drugi buffer (disp_draw_buf2), ali to nije napravljeno jer ESP 32 nema dovoljno RAM-a za to, ali dodavanje drugog buffer-a bi značajno ubrzalo rad LVGL-a i povećalo frame rate.

7.1. LVGL handler

Kako bi LVGL mogao raditi potrebno je pozivati lv_timer_handler() funkciju svakih pedesetak milisekundi, što je prilično nezgodno jer za slušanje BLE notifikacija potrebno je konstantno slušati za notifikacije. Srećom ESP32 ima dvije jezgre pa je zato odlučeno da će se jedna jezgra baviti LVGL-ovim updejtanjem, a druga s BLE potrebama.

U setupu je napravljen task za core0:

```
// TASK HANDLES //
TaskHandle_t LVGL_handler_task; // goes on core0

void setup(){
    // Create a task to handle the LVGL updates
    xTaskCreatePinnedToCore(
        LVGL_handler_function, /* Task function. */
        "LVGL_handler_task",   /* name of task. */
```

```
        100000,                 /* Stack size of task */
        NULL,                   /* parameter of the task */
        1,                      /* priority of the task */
        &LVGL_handler_task,     /* Task handle to keep track of created task */
        0);                     /* pin task to core 0 */
    // Setup functions //
    init_LVGL();
    init_tabs();
    init_BLE();
    //
}
```

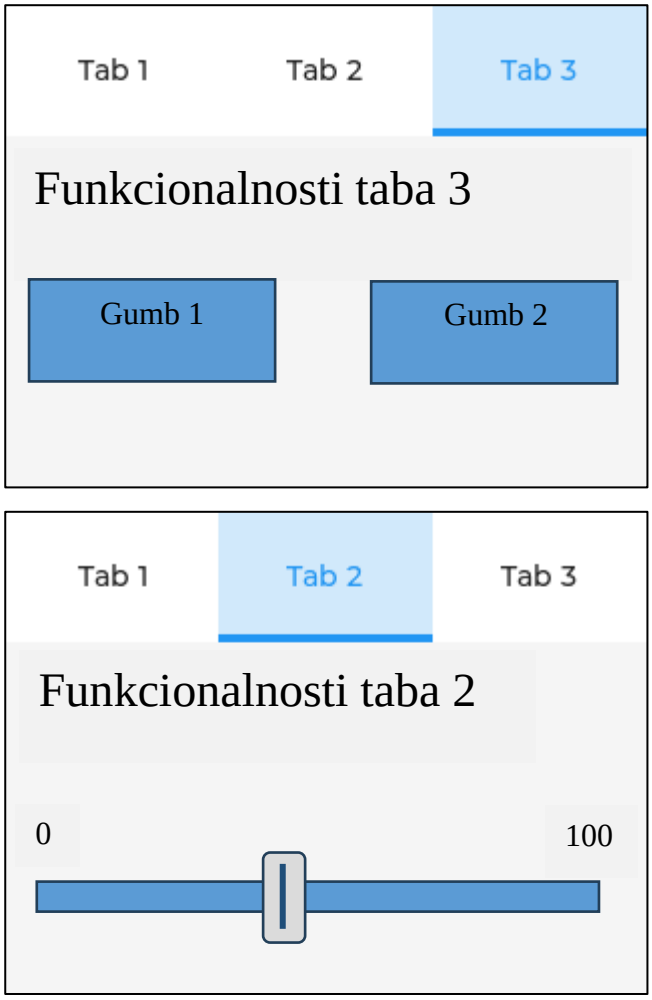
Task na jezgri 0 koji se bavi updejtanjem LVGL-a:

```
void LVGL_handler_function(void *pvParameters){
    /*
     *   This task handles the LVGL timer as to
     *   free up the main loop
     */
    uint32_t timer_handler_time;
    while (true){
        // Timer handler //
        timer_handler_time = lv_timer_handler();    // don't touch this
        vTaskDelay(timer_handler_time + 30);        //
        //
    }
}
```

8. GUI (Graphical User Interface)

8.1. Općeniti dizajn sučelja

Odlučeno je da će ovaj GUI imati strukturu tabova kao browser. Svaki tab u ovom GUI-u će biti zadužen za jedan skup sličnih funkcija (npr. tab postavke, tab komunikacija, tab komande, itd.). Ova struktura GUI-a omogućava laganu nadogradnju novih funkcionalnosti nadodavanjem novih tabova. Ovakva struktura GUI-a omogućava nadogradnju koja ne utječe na prijašnje tabove, tako da nadogradnjom novih funkcija se ne treba mijenjati kod postojećih funkcija. U sklopu ovog rada napravit će se samo jedan tab (connection_tab), ali on će demonstrirati kako implementirati nove tabove i nove funkcionalnosti u strukturu ovog GUI-a.



Slika 9. Primjer tab strukture GUI-a

Ova struktura se postiže stvaranjem tabview widgeta, na kojemu će biti svi tabovi:

```
main_tabview = lv_tabview_create(default_screen, LV_DIR_TOP, MAIN_TABVIEW_HEIGHT);
```

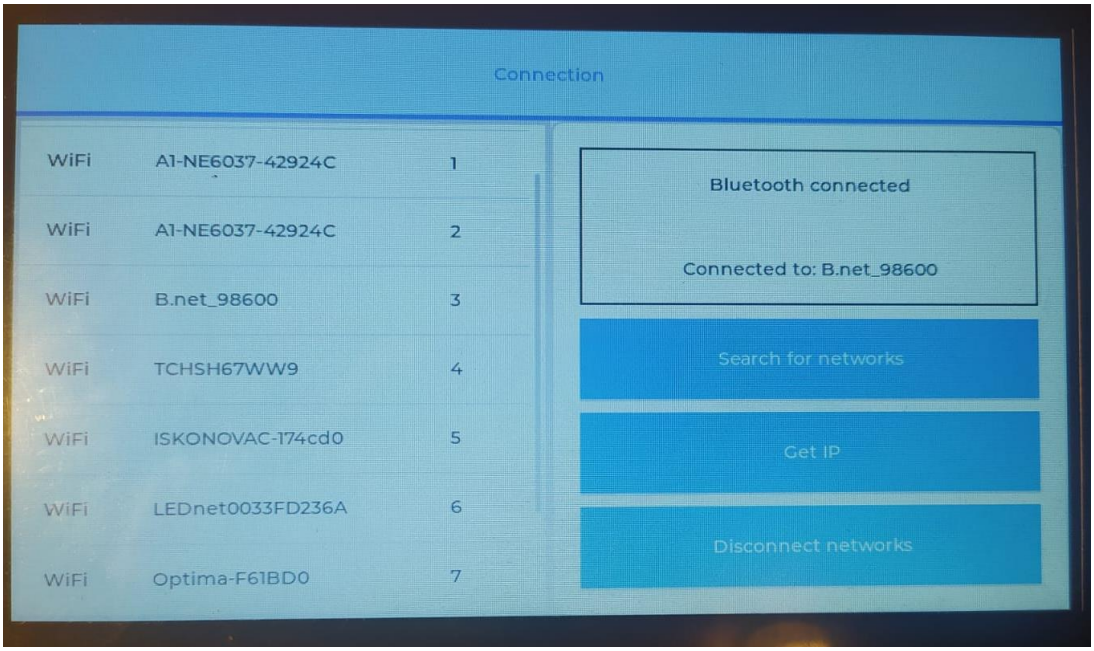
8.2. Connection_tab

Funkcionalnost koja će se u ovom radu dodati GUI-u je connection_tab, ovo će biti tab zadužen za kontroliranje mrežne spojenosti robota kojega se kontrolira. Funkcionalnosti koji će ovaj tab pružiti su:

- Spajanje robota na izabranu WiFi ili Ethernet mrežu
- Odspajanje robota od WiFi ili Ethernet mreža
- Prikazivanje IP informacija spojene mreže robota
- Prikazivanje statusa spojenosti mreže te na koju mrežu je spojen robot
- Prikaz statusa spojenosti robota s HMI-om (je li spojeno preko BLE)

8.3. Dizajn taba

Connection_tab će biti podijeljen na dvije polovice, s lijeve strane će biti tablica sa svim mrežama na koje se korisnik može spojiti, s desne strane će biti label(widget texta) za status spojenosti s bluetoothom te label za status spojenosti robota s mrežom, isto tako će biti i tri gumba: „Search networks“, „Get IP“ i „Disconnect networks“. Spajanje na željene mreže će se izvršavati preko lijeve tablice, tamo će se pritisnuti na željenu mrežu, nakon čega će iskočiti popup koji pita za lozinku te prostor za upisivanje lozinke koji nakon što je kliknut, stvara tipkovnicu na ekranu za upisivanje lozinke. Zadnje, nakon klika na „Get IP“ gumb, treba iskočiti popup s IP informacijama.



Slika 10. Željeni izgled connection taba

8.4. Connection_tab dio programa

Stvaranje widgeta za „connection_tab“ raditi će funkcija „init_connection_tab()“. Prvo je potrebno napraviti tab, te postaviti FlexFlow kako bi se distributirali elementi.

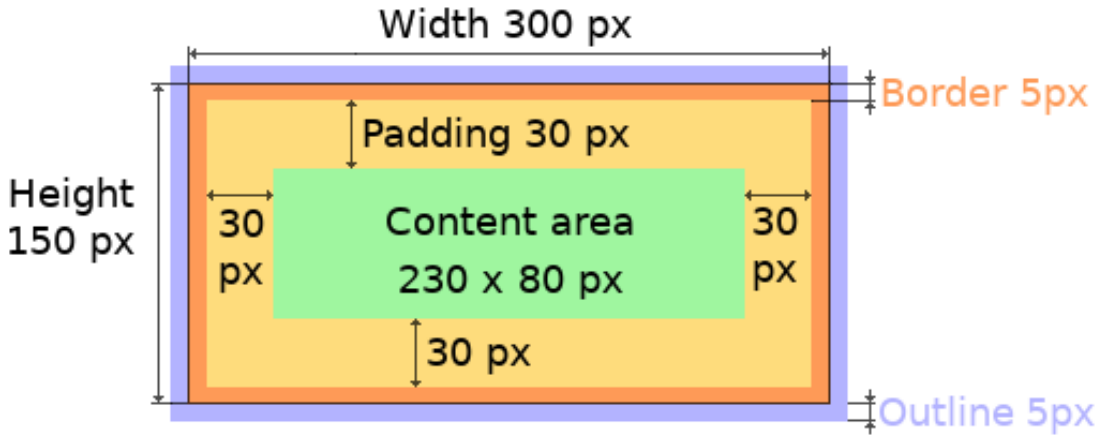
```
void init_connection_tab(){
    /*
     *   This tab deals with connecting wifi/ethernet to PLEA
     */
    connection_tab = lv_tabview_add_tab(main_tabview, "Connection"); // create
connection_tab
    lv_obj_set_flex_flow(connection_tab, LV_FLEX_FLOW_ROW);
```

Za lakši rad, odlučeno je stvoriti dva objekta za svaku polovicu taba(to će zapravo biti samo veliki pravokutnici), connection_buttons_backdrop i connection_table_backdrop te na njima postaviti flex flow za lakše raspoređivanje widgeta.

```
    connection_table_backdrop = lv_obj_create(connection_tab);
    lv_obj_t *connection_buttons_backdrop = lv_obj_create(connection_tab);
    lv_obj_set_height(connection_table_backdrop, CONNECTION_TABLE_BACKDROP_HEIGHT);
    lv_obj_set_height(connection_buttons_backdrop,
CONNECTION_BUTTONS_BACKDROP_HEIGHT);
    lv_obj_set_flex_flow(connection_table_backdrop, LV_FLEX_FLOW_COLUMN);
    lv_obj_set_flex_flow(connection_buttons_backdrop, LV_FLEX_FLOW_COLUMN);

    lv_obj_set_flex_grow(connection_table_backdrop, 1);   // set the ratio of table
    lv_obj_set_flex_grow(connection_buttons_backdrop, 1); // to buttons
```

Pošto defaultni objekti dolaze s padding gap-om(prazna granica oko objekta [6]) i radijusima, korišten je stil da makne ta svojstva.

```
    // Reduce padding and remove radius
    static lv_style_t no_padding_style;
    lv_style_init(&no_padding_style);
    lv_style_set_radius(&no_padding_style, 0);
    lv_style_set_pad_all(&no_padding_style, 0);
    lv_style_set_pad_gap(&no_padding_style, 0);

    lv_obj_add_style(connection_tab, &no_padding_style, 0);
    lv_obj_add_style(connection_table_backdrop, &no_padding_style, 0);
```



Slika 11. Primjer razmaka kod defaultnog objekta [6]

```
    // Connection table
    no_connections_available(connection_table_backdrop);
```

Pošto na početku rada HMI-a program nema pristup nikakvim mrežama, umjesto tablice mreža na lijevu stranu ekrana stavit će se samo tekst „networks unavailable“, to će se raditi pomoću funkcije no_connections_available() koja će pozivati ili kada program tek krene ili kada HMI izgubi konekciju s robotom:

```
void no_connections_available(lv_obj_t* backdrop){
    lv_obj_clean(backdrop);
    lv_obj_t* placeholder_label = lv_label_create(backdrop);
    lv_label_set_text(placeholder_label, "Networks unavailable.");
    lv_obj_center(placeholder_label);
    //lv_obj_set_align(placeholder_label, LV_ALIGN_CENTER);
}
```

Nastavljajući dalje sa init_connection_tab() funkcijom, potrebno je postaviti flex flow na desnom objektu te naporaviti stil za uređivanje gumbova (korištenje flex flow-a detaljno je objašnjeno u dokumentaciji[7]).

```
    lv_obj_set_flex_flow(connection_buttons_backdrop, LV_FLEX_FLOW_COLUMN);
    lv_obj_set_flex_align(connection_buttons_backdrop, LV_FLEX_ALIGN_START,
LV_FLEX_ALIGN_START, LV_FLEX_ALIGN_START); // how to align backdrop

    // Create a style for buttons
    static lv_style_t connect_btn_style;
    lv_style_init(&connect_btn_style);
    lv_style_set_width(&connect_btn_style, LV_PCT(100));
    lv_style_set_flex_grow(&connect_btn_style, 1);
```

```
    lv_style_set_radius(&connect_btn_style, 0);
    lv_style_set_border_color(&connect_btn_style, lv_color_hex(0x2f3436));
    lv_style_set_text_align(&connect_btn_style, LV_TEXT_ALIGN_CENTER);
```

Za potrebe indikacije konekcije sa robotom i robotove konekcije sa mrežom potrebno je napraviti objekt koji će držati dva labela koji će biti indikatori tih konekcija.

```
    // Add connection indicator object
    lv_obj_t *con_indicator_backdrop = lv_obj_create(connection_buttons_backdrop);
    lv_obj_add_style(con_indicator_backdrop, &connect_btn_style, LV_PART_MAIN);
    lv_obj_set_flex_grow(con_indicator_backdrop, 2);

    BLE_connection_status_label = lv_label_create(con_indicator_backdrop);
    lv_label_set_text(BLE_connection_status_label, "Bluetooth disconnected");
    lv_obj_set_align(BLE_connection_status_label, LV_ALIGN_TOP_MID);

    NET_connection_status_label = lv_label_create(con_indicator_backdrop);
    lv_label_set_text(NET_connection_status_label, "Network state unknown");
    lv_obj_set_align(NET_connection_status_label, LV_ALIGN_BOTTOM_MID);
```

Na kraju potrebno je napraviti gumbe za prijašnje navedene funkcije, ovdje će biti navedeno kako je izveden „srch_networks_btn“, ali postupak je isti i za ostala dva gumba:

- Stvaranje gumba i stavljanje teksta na njemu

```
    // Add search connections button
    lv_obj_t *srch_networks_btn = lv_btn_create(connection_buttons_backdrop);
    lv_obj_t *srch_networks_btn_label = lv_label_create(srch_networks_btn);
    lv_label_set_text(srch_networks_btn_label, "Search for networks");
```

- Dodavanje stila na gumb i label

```
    lv_obj_add_style(srch_networks_btn, &connect_btn_style, 0);
    lv_obj_add_style(srch_networks_btn_label, &connect_btn_style, 0);
```

- Dodavanje callback funkcije i centriranje labela

```
    lv_obj_add_event_cb(srch_networks_btn, send_simple_command_cb,
LV_EVENT_CLICKED, (void *)&PLEA_commands[0]); // Send 's'
    lv_obj_center(srch_networks_btn_label);
```

Callback funkcija koja se poziva za svaki gumb je send_simple_command_cb(), u ovom poglavlju se neće objasniti što ta funkcija radi jer se bavi BLE-om što će se objasniti u kasnijem poglavlju. Jedino što je za sad važno je to da svaki od tri gumba šalje callback funkciji jedan char-ova iz PLEA_commands polja.

```
const char PLEA_commands[] = {
    /*
```

```
     *   This are simple commands that will
     *   be sent to PLEA via buttons.
     */
    SEARCH_NETWORKS_COMMAND,    // [0] -serch for networks
    REQUEST_IP_COMMAND,         // [1] -send IP
    DISCONNECT_NETWORK_COMMAND  // [2] -disconnect network
};
```

8.5. Mogućnosti nadogradnje

Za dodavanje novih tabova potrebno je napraviti dodatnu inicijalizacijsku funkciju poput init_connection_tab() te u njoj napraviti tab kojemu je roditelj main_tabview widget, jer se na njemu stavljaju svi tabovi. Tu inicijalizacijsku funkciju treba dodati u init_tabs() funkciju koja se poziva iz setup() funkcije i namijenjena je za inicijalizaciju svih tabova. Svi drugi widgeti koji su potrebni za taj tab trebaju biti dijete ili potomak tog novog taba.

Primjer dodavanja novih tabova sa mogućim funkcionalnostima:

```
void init_tabs(){
    main_tabview = lv_tabview_create(default_screen, LV_DIR_TOP,
MAIN_TABVIEW_HEIGHT); // create main tabview
    init_connection_tab();
    //init_macros_tab();
    //init_PLEA_settings_tab();
    //init_settings_tab();
}
```

9. Dizajn BLE servisa

Za komunikaciju između HMI-a i robota izabran je BLE protokol jer je dizajniran za brze prijenose kratkih paketa podataka preko kratkih udaljenosti (do 10 metara) sa malom potrošnjom energije te je zbog ovih karakteristika vrlo optimalan za HMI potrebe. U ovom poglavlju će biti pokazano kako je programski dizajniran BLE API sa strane HMI-a za connection_tab. Connection_tab ima potrebu za četiri jednim BLE servisom i četiri karakteristike:

- BLE_network_names_ch - prima imena mreža te vrstu mreža koje joj robot prosljeđuje
- BLE_network_connect_ch - šalje robotu željene mreže za spajanje te lozinke
- BLE_network_command_ch - šalje robotu komande
- BLE_network_message_ch – prima poruke od robota

9.1. BLE inicijalizacija

Postoje osnovni koraci kroz koje je potrebno proći kako bi HMI mogao postati BLE server uređaj. Za inicijaliziranje BLE servera potrebno je postaviti HMI kao server uređaj, dodati mu callback za spajanje i odspajanje, inicijalizirat sve servise i njihove karakteristike te na kraju započeti advertising(oglašavanje).

Potrebne varijable:

```
BLEServer *BLE_HMI_server = NULL;
BLEAdvertising *BLE_advertising = NULL;
```

U init_BLE() funkciji inicijalizira se BLE općenito za sve servise:

```
void init_BLE(){

```

Prvo je potrebno inicijalizirati HMI kao BLEDevice te mu zadati ime po kojemu će ga klijent uređaji moć prepoznati:

```
    // Create the BLE Device
    BLEDevice::init("PLEA HMI");
```

Zatim je potrebno napraviti HMI u server uređaj te mu pridati klasu koja će se baviti događanjima koje se žele napraviti kada se neki novi klijent spoji ili odspoji od HMI-a:

```
    // Create the BLE Server
    BLE_HMI_server = BLEDevice::createServer();
```

```
    BLE_HMI_server->setCallbacks(new ServerCallbacks()); // Set callback on connect
and disconnect
```

Klasa koja se poziva pri spajanju i odspajanju novog uređaja:

```
class ServerCallbacks : public BLEServerCallbacks
{
    void onConnect(BLEServer *BLE_HMI_server){
        // what to do on connection
        BLEdeviceConnected = true;
        lv_label_set_text(BLE_connection_status_label, "Bluetooth connected");
        lv_label_set_text(NET_connection_status_label, "Network state unknown");
        //Serial.println("Device connected");
    };

    void onDisconnect(BLEServer *BLE_HMI_server){
        // what to do on disconnection
        BLEdeviceConnected = false;
        lv_label_set_text(BLE_connection_status_label, "Bluetooth disconnected");
        lv_label_set_text(NET_connection_status_label, "Network state unknown");
        //Serial.println("Device disconnected");
        no_connections_available(connection_table_backdrop);
        BLEDevice::startAdvertising(); // wait for another connection
    }
};
```

Ova klasa je zadužena za mijenjanje stanja indikatora, koji se koristi u glavnoj petlji, koji govori je li robot spojen. Uz to mijenja indikator BLE konekcije koji korisnik vidi na ekranu kako bi mogao znati je li spojen sa robotom. Na kraju ako se robot odspoji poziva advertising(oglašavanje), jer da bi robot mogao naći HMI on mora prvo oglašavati svoje postojanje i dostupnost drugim uređajima.

Vraćanjem na init_BLE() funkciju, ostavljen je prostor za ubacivanje inicijalizacijskih funkcija za sve servise, ali pošto je za sada jedini servis na HMI-u BLE_network_service to je i jedini servis koji je potrebno inicijalizirati. Dodavanjem novih servisa za druge tabove predviđeno je da se ovdje stavlja:

```
    // Initialize services
    init_BLE_network_service();
```

Na kraju potrebno je isključiti „scan response“ jer se ovdje neće slati dodatne scan informacije i potrebno je staviti minimalno preferirano odgovorno vrijeme na „0x0“ kako bi HMI bio fleksibilan za sva odgovorna vremena. Kada je sve postavljeno, može se krenuti oglašavati uređaj, nakon toga klijenti se mogu spojiti na HMI.

```
    // Start advertising
    BLE_advertising->setScanResponse(false);
    BLE_advertising->setMinPreferred(0x0);
    BLEDevice::startAdvertising();
}
```

9.2. BLE servisna inicijalizacija

Kako bi se karakteristike mogle napraviti, mora postojati servis koji će ih sadržavati. Ideja je da svaki tab ima svoji servis koji će sadržavati sve njegove karakteristike. Tako će connection_tab sadržavati svoji servis koji će se zvati BLE_network_service. Postupak za inicijalizaciju servisa i njegovih karakteristika biti će prikazan ovdje pomoću init_BLE_network_service() funkcije, ali za stvaranje inicijalizacijskih funkcija za nove servise može se koristiti isti postupak.

Potrebne varijable:

```
BLECharacteristic *BLE_network_names_ch = NULL;
BLECharacteristic *BLE_network_connect_ch = NULL;
BLECharacteristic *BLE_message_ch = NULL;
BLECharacteristic *BLE_network_commands_ch = NULL;
BLEAdvertising *BLE_advertising = NULL;
```

Prvo je potrebno stvoriti servis na server uređaju, sve karakteristike i servisi moraju imati svoji UUID (Universally Unique Identifier). UUID je poseban string slova i brojeva koji se koriste za prepoznavanje servisa ili karakteristike (primjer izgleda UUID-a: 9fd1e9cf-97f7-4b0b-9c90-caac19dba4f8). Najlakši način za dobiti validan UUID je korištenjem online generatora.

```
void init_BLE_network_service(){
    /*
     *   This function creates the BLE_network_service
     *   and it starts it.
     */
    BLE_network_service = BLE_HMI_server->createService(NETWORK_SERVICE_UUID);
```

Nakon što je servis kreiran, u njega se mogu stavljati karakteristike. Prva će biti karakteristika za primanje imena mreža.

```
    // Create BLE_network_names_ch characteristic
    // It is intended for receiveing network names from Raspberry PI
    BLE_network_names_ch = BLE_network_service->createCharacteristic(
        NETWORK_NAMES_CH_UUID,
        BLECharacteristic::PROPERTY_WRITE);
    BLE_network_names_ch->setCallbacks(new recieveNetworkNamesCallback()); // Add
callback on recieve from client
```

Stvaranje nove karakteristike radi se pomoću „createCharacteristic()“ funkcije, za stvaranje nove karakteristike potrebno je ovoj funkciji dati UUID nove karakteristike, te svojstva koja će karakteristika imati (PROPERTY_WRITE, PROPERTY_READ, itd.). BLE_network_names_ch treba imati PROPERTY_WRITE svojstvo jer će klijent u tu karakteristiku upisivati. Ako će se karakteristici slati podatci od strane klijenta, potrebno joj je dati callback klasu, kod BLE_network_names_ch to je recieveNetworkNamesCallback(), ali ovdje se neće objasniti callback klase, već u idućem podpoglavlju.

Princip je isti i za karakteristiku koja prima poruke od klijenta, samo što će ona imati drugu callback klasu jer treba drugačije stvari raditi, ovdje ta klasa neće biti objašnjena već nakon idućeg podpoglavlja.

```
    // Create BLE_message_ch characteristic
    // It is intended for receiveing IPs from Raspberry PI
    BLE_message_ch = BLE_network_service->createCharacteristic(
        NETWORK_MESSAGE_CH_UUID,
        BLECharacteristic::PROPERTY_WRITE);
    BLE_message_ch->setCallbacks(new recieveMessageCallback()); // Add callback on
recieve from client
```

Za karakteristiku koja šalje podatke za spajanje mreže i karakteristiku koja šalje naredbe, potrebno je staviti svojstva PROPERTY_READ i PROPERTY_NOTIFY jer će one zapravo slati notifikacije robotu koje će robot čitati. Da bi notifikacije mogle dobro raditi potrebno je još na karakteristiku staviti BLE2902 deskriptor, on uključuje mogućnost notifikacija na klijentu za ovu karakteristiku.

```
    // Create BLE_network_connect_ch characteristic
    // It sends network type, name and password of the network we want to connect
to
    BLE_network_connect_ch = BLE_network_service->createCharacteristic(
        NETWORK_CONNECT_CH_UUID,
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLE_network_connect_ch->addDescriptor(new BLE2902()); // Add a BLE descriptor
for notifications

    // Create BLE_network_commands_ch characteristic
    // It sends simple commands via buttons
    BLE_network_commands_ch = BLE_network_service->createCharacteristic(
        NETWORK_COMMANDS_CH_UUID,
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLE_network_commands_ch->addDescriptor(new BLE2902()); // Add a BLE descriptor
for notifications
```

Na kraju je potrebno predati cijeli servis advertising-u kako bi se servis mogao oglašavati klijentima (ali tek kada oglašavanje započne) i započeti servis:

```
    BLE_advertising->addServiceUUID(NETWORK_SERVICE_UUID); // Add
BLE_network_service to advertising
    BLE_network_service->start(); // Start the service
}
```

9.2.1. Callback klasa za BLE_network_names_ch

Callback ima samo jednu funkciju koja prosljeđuje primljeni string (od maksimalno 20 char-ova) funkciji BLE_string_from_chunks(). Pošto će string koji sadrži imena svih mreža biti puno duži od 20 char-ova, karakteristika dobiva samo komade punog stringa, zato je BLE_string_from_chunks() funkcija zadužena da pretvori te komade (tj. „chunkove“) u potpuni string te da javi nazad da je string završen preko network_string_completed flag-a . Nakon što se dobije puni string imena mreža, networks_string_received se stavlja u true da signalizira glavnoj petlji ovaj događaj.

```
// class that deals with receiving string chunks from client
class recieveNetworkNamesCallback : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *receiver_characteristic)
    {
        std::string BLE_received_string = receiver_characteristic->getValue();
        BLE_string_from_chunks(BLE_received_string, &network_names_string,
&network_string_completed);
        if (network_string_completed)
        {
            networks_string_received = true;
            network_string_completed = false;
        }
    }
};
```

BLE_string_from_chunks() nadodaje na adresu stringa u svakom ciklusu novi „chunk“ od 20 char-ova koji su došli preko karakteristike. Kada funkcija pročita da je zadnji char u primljenom chunku '#' onda zna de je ovo kraj stringa te signalizira nazad preko flag-a.

```
void BLE_string_from_chunks(std::string chunk, std::string *storage_string, bool
*completed_message_indicator){
    /*
     *   Takes the incoming chunks of the string coming in
     *   and appends them to a string. Detects the end of a
     *   message with the '#' and changes an indicator booll
     *   to true.
     */
    *storage_string += chunk; // Append the chunk that was sent over BLE to
```

```
                                // a string that will containt the whole message
    if (chunk.back() == '#' || chunk == "")
    { // If the chunk ends with '#' or it is empty
        *completed_message_indicator = true;
    }
}
```

Ovaj sistem komunikacije radi zato što je odlučeno da će slanje dugačkih stringova biti izvedeno tako da se šalju komadi originalnog stringa veličina 20 char-ova, a da će se kraj stringa prepoznati tako što će se na kraj stringa stavljati '#'. Primjer općenitog izgleda stringa za ovu karakteristiku je „W:<<Mreža1>>E:<<Mreža2>>…W:<<MrežaN>>#“. Imena svih mreža se stavljaju u duple strelice (<<ime_mreže>>), a ispred imena ide ili „W:“ ili „E:“ ovisno o tome je li mreža WiFi ili Ethernet, na kraju za svaku mrežu se onda dobije izgled: „tip_mreže:<<ime_mreže>>“. Kada se tako nanizaju sve mreže na kraju se stavlja '#' za označavanje kraja stringa. Druge karakteristike koje primaju podatke od robota imaju malo drugačiji standard stringa koji primaju iz svrhe sigurnosti da se ne bi slučajno mogao poslati string na krivu karakteristiku i raditi.

9.2.2. Callback klasa za BLE_network_names_ch

Ovaj callback radi identično kao i callback iz prošlog podpoglavlja, ali koristi druge flag-ove za indikaciju završenosti stringa i signalizacije glavnoj petlji.

```
// class that deals with receiving string chunks from client
class recieveMessageCallback : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *receiver_characteristic)
    {
        std::string BLE_received_string = receiver_characteristic->getValue();
        BLE_string_from_chunks(BLE_received_string, &net_message_string,
&net_message_string_completed);
        if (net_message_string_completed)
        {
            net_message_string_received = true;
            net_message_string_completed = false;
        }
    }
};
```

Ono što je drugačije je standard stringa koji se očekuje primiti. String se sastoji od pojedinih informacija ekapsuliranim u strelicama (<<informacija>>) te kraj stringa je označen hashtagom '#'. Primjer izgleda stringa: <<informacija1>><<informacija2>><<informacija3>>#

10. Logika primanja podataka

Kada su servisi i karakteristike napravljeni tok informacija je spreman, ali za neko smisleno korištenje informacija koje dolaze potrebno je imati neku logiku koja će kontrolirati ponašanje HMI-a i proslijeđene naredbe za robota po nekom željenom postupku. Inače kod komunikacija je problem što protokoli poput BLE-a moraju konstantno slušati za to hoće li primiti neku notifikaciju. Zato je u ovom programu deligirana jezgra 0 za LVGL updejtanje, a jezgra 1 za BLE-ovo slušanje notifikacija i procesiranje logike. Struktura logike na jezgri 1 će biti objašnjena u idućim podpoglavljima. (Za određivanje BLE mogućnosti ESP-a kako bi se mogao napraviti optimalan protokol između klijenta i servera, potrebno je pogledati BLE poglavlje u dokumentaciji [8].)

10.1. Glavna petlja

Glavna petlja koja se cijelo vrijeme vrti samo cijelo vrijeme provjerava je li robot spojen na HMI preko BLE-a i ako je robot spojen, onda petlja provjerava dva flag-a (networks_string_received i net_message_string_received). Ti flag-ovi javljaju da je primljen string imena mreža ili je primljena poruka od robota.

```
void loop(){
    if (BLEdeviceConnected == true){
        if (networks_string_received == true){
            networks.clear();
            networks.shrink_to_fit();
            BLE_network_names_from_string(network_names_string, networks);
            put_network_names_in_table(networks);
            network_names_string = "";
            network_string_completed = false;
            networks_string_received = false;
        }
        if (net_message_string_received == true){
            BLE_array_from_string(&net_message_string, net_message_strings_array,
&net_message_num_strings);
            handle_net_message(net_message_strings_array,
&net_message_num_strings);
            net_message_string = "";
            net_message_string_completed = false;
            net_message_string_received = false;
        }
    }
}
```

10.1.1. Logika za primljeni string imena mreža

Ako se primi string imena mreža, networks_string_received flag će postati true. Odlučeno je da će se informacije o mrežama spremati u strukturu „Network“ koja će imati varijable: name, type, password i table_entry number. Svaki „Network“ će biti spremljen u vektor „networks“.

```
// Struct that holds network information
struct Network
{
    std::string name;     //
    std::string type;     // "WiFi" or "Ethernet"
    std::string password; //
    byte table_entry_nmb; //
};
std::vector<Network> networks;  // vector that holds network names
```

Ciklus logike kada dođe novi string imena mreža je:

1) Brisanje prošlih mreža iz vektora i smanjivanje vektora:

```
networks.clear();
networks.shrink_to_fit();
```

2) Procesiranje stringa te stavljanje informacija u strukturu (objašnjeno u idućem poglavlju):

```
BLE_network_names_from_string(network_names_string, networks);
```

3) Kreiranje tablice i stavljanje informacija u tablicu (objašnjeno u idućem poglavlju):

```
put_network_names_in_table(networks);
```

4) Resetiranje stringa i flag-ova:

```
network_names_string = "";
network_string_completed = false;
networks_string_received = false;
```

10.1.2. Logika za primljeni string poruke

Ako se primi string poruke od robota, message_string_received flag će postati true. Ciklus logike kada dođe novi string poruke je:

1) Rastavljanje stringa u zasebne informacije te stavljanje informacija u polje(objašnjeno u idućem poglavlju)::

```
BLE_array_from_string(&net_message_string, net_message_strings_array,
&net_message_num_strings);
```

2) Rad s primljenim informacijama(objašnjeno u idućem poglavlju)::

```
handle_net_message(net_message_strings_array, &net_message_num_strings);
```

3) Resetiranje stringa i flag-ova:

```
net_message_string = "";
net_message_string_completed = false;
net_message_string_received = false;
```

11. Prikaz mrežnih podataka i odabir mreže

11.1. Procesiranje stringa mreža

Ono što se prima od robota je samo string koji je potrebno pretvoriti u zasebne podatke koje program može koristiti. To radi funkcija BLE_network_names_from_string(). Funkcija uzima string informacija te vektor(networks_string) u koji se informacije upisuju(„networks“).

```
void BLE_network_names_from_string(std::string networks_string,
std::vector<Network> &networks){
    /*
     *   Chops the networks_string into individual networks.
     *
     *   Function searches for string chunks that start with W:
     *   or E: and the name of the network that is between << and >>
     *   '#' marks the end of the string.
     *
     *   Legend:
     *   W:  WiFi network
     *   E:  Ethernet network
     *   <<  Start of a network's name
     *   >>  End of a network's name
     *
     *   Example (one WiFi network with a name net123):
     *   W:<<net123>>#
     */
```

Funkcija na početku čisti cijeli vektor:

```
    networks.clear(); // Empty the vector before writing new networks
```

Zatim definira „regex“ izraz kao „re“. Regex stoji za „regular expression“, to je klasa iz standardne C++ biblioteke. Regex izrazi su izrazi koji služe za definiranje kakav izraz se traži u stringu.

```
    std::regex re(R"((W:|E:)<<([^>]+)>>)");
```

Sintaksa regex izraza ovdje govori da se traži izraz koji započinje s „W:“ ili „E:“ (W:|E:) te drugi izraz koji se nalazi između dvostrukih strelica (<<([^>]+)>>). Zatim je potrebno stvoriti varijablu koja će držati nađene izraze.

```
    std::smatch match;
```

Potrebno je još napravi iterator koji počinje na početku stringa „networks_string“.

```
    std::string::const_iterator searchStart(networks_string.cbegin());
```

Sada se mogu izvući informacije iz stringa

```
    // Extract network names and their types
    byte i = 0;
    while (std::regex_search(searchStart, networks_string.cend(), match, re))
    {
        std::string type = match[1] == "W:" ? "WiFi" : "Eth";
```

```
        std::string name = match[2];
        networks.push_back({name, type, "", i}); // Initially, the password is
empty
        searchStart = match.suffix().first;
        i++;
    }
}
```

Ovaj dio koda prolazi kroz cijeli string, svaki put kad nađe točan izraz spremi ga u match, Ako je prvi dio match-a „W:“ mreža je WiFi, ako ne onda je „Eth“. Ime je drugi dio match-a. Svaki put kad se nađe match prođe se u petlju te brojač se poveća i zabilježi koji broj mreže je to. Sve ove informacije se stavljaju u vektor pod jednu mrežu.

11.2. Tablica mreža

Kada su sve mreže postavljene u vektor, potrebno je napraviti tablicu sa svim mrežama gdje će korisnik imati mogućnost izabrati mrežu na koju se želi spojiti. Taj zadatak rješava funkcija put_network_names_in_table().

Funkcija uzima vektor te nalazi broj mreža u vektoru:

```
void put_network_names_in_table(const std::vector<Network> &networks){
    byte number_of_networks = networks.size();
```

Čisti objekt koji će sadržavati tablicu te stvara novu tablicu. (Potrebno je očistiti prijašnji objekt jer on može sadržavati „networks unavailable“ tekst ili prošlu tablicu koja može imati druge mreže).

```
    lv_obj_clean(connection_table_backdrop);
    connection_table = lv_table_create(connection_table_backdrop);
```

Zatim je potrebno maknuti prošli event callback s tablice, jer ako se to ne napravi, svaki put kada dođe do nove tablice aktivirat će se isti event više puta. Kada je maknut prošli event callback treba se staviti novi event callback. Callback se poziva na LV_EVENT_VALUE_CHANGED jer ako se na callback ulazi pomoću LV_EVENT_CLICKED ili LV_EVENT_PRESSED, neće se moći skrolati po tablici bez da se uđe u callback (ako broj mreža prelazi veličinu tablice, tablica postaje scrollable).

```
    lv_obj_remove_event_cb(connection_table, open_password_popup);  // If we don't
remove previous callback, the callback will happen twice
    lv_obj_add_event_cb(connection_table, open_password_popup,
LV_EVENT_VALUE_CHANGED, NULL);
```

Potrebno je odrediti veličinu tablice te postaviti da će imati 3 stupca i njihove veličine. Veličina tablice je određena relativno na roditeljski objekt (stavljeno je na 100% širine i visine), dok je veličina stupaca određena apsolutno brojem piksela.

```
    lv_obj_set_size(connection_table, LV_PCT(100), LV_PCT(100));    // Make it
takeup the whole backdrop
```

```
    lv_table_set_col_cnt(connection_table, 3);                    //
    lv_table_set_col_width(connection_table, 0, 80);              //
    lv_table_set_col_width(connection_table, 2, 80);              //
    lv_table_set_col_width(connection_table, 1, 220);             //
```

Na kraju se pomoću for petlje stavljaju sve mreže u tablicu.

```
    for (byte i = 0; i < number_of_networks; i++)
    {
        lv_table_set_cell_value(connection_table, i, 0, networks[i].type.c_str());
        lv_table_set_cell_value(connection_table, i, 1, networks[i].name.c_str());
        lv_table_set_cell_value(connection_table, i, 2,
std::to_string(networks[i].table_entry_nmb).c_str());
        /*
        Serial.print(networks[i].name.c_str());         //
        Serial.print(" entry: ");                       // Troubleshooting block
        Serial.println(networks[i].table_entry_nmb);    //
        */
    }
}
```

WiFi	Ime_Mreže_1	1.
WiFi	Ime_Mreže_2	2.
WiFi	Ime_Mreže_3	3.
WiFi	Ime_Mreže_4	4.
WiFi	Ime_Mreže_5	5.
Eth	Ime_Mreže_6	6.
Eth	Ime_Mreže_7	7.

Slika 12. Dijagram približnog izgleda tablice

11.3. Spajanje na mrežu

Za pokretanje procesa spajanja na mrežu potrebno je kliknuti na bilo koje polje u redu mreže na koju se korisnik želi spojiti. Pošto sve fizičke interakcije s widgetom stvaraju event na tome widgetu, a u na tablicu je stavljen event callback za LV_EVENT_VALUE_CHANGED, to će pokrenuti funkciju open_password_popup(). Ova funkcija stvara popup prozor koji ima na sebi gumbe „connect“ i „cancel“, isto tako ako je odabrana WiFi mreža imat će „textarea“ widget. To je widget namijenjen za upisivanje teksta, klikom na njega iskočit će na ekranu tipkovnica. Ako mreža nije tipa WiFi nego

ethernet, onda popup neće imati textarea jer za spajanje na ethernet nije potrebna lozinka. Proces izrade ove funkcionalnosti objašnjen je ispod.

Potrebno je dobiti roditeljski objekt za saznavanje koje polje je pritisnuto.

```
void open_password_popup(lv_event_t *e){
    // Get the clicked network
    lv_obj_t *table = lv_event_get_target(e);
```

Dobivanje informacija o željenoj mreži za spajanje se radi tako da se nađe polje koje je bilo pritisnuto u tablici te se saznaje u kojem je redu. Pošto je poznat standard za spremanje mrežnih informacija u tablicu (prvi stupac – tip mreže, drugi stupac – ime mreže, treći stupac – redni broj mreže), poznato je koji stupci u tom redu se trebaju dohvatiti za tražene informacije.

```
    uint16_t row, col;
    lv_table_get_selected_cell(table, &row, &col);
    const char* network_name = lv_table_get_cell_value(table, row, 1);
    const char* network_type = lv_table_get_cell_value(table, row, 0);
    const char* msg_prt1 = "Connect to: ";
    char pupup_message[50];
    snprintf(pupup_message, sizeof(pupup_message), "%s%s", msg_prt1, network_name);
```

Zatim se stvara poruka koja će se prikazivati u popup prozoru, a ona treba sadržati ime željene mreže.

```
    snprintf(pupup_message, sizeof(pupup_message), "%s%s", msg_prt1, network_name);
```

Spremaju se informacije o odabranoj mreži u globalnu strukturu.

```
    selected_network.name = network_name;
    selected_network.type = network_type;
```

Kada su svi potrebni podatci spremni, izrađuje se popup widget sa željenim porukama na njemu. (Isto stvara se stil za popup koji će se kasnije koristiti za pomicanje popup-a.)

```
    // Create password popup
    lv_obj_t *password_popup = lv_msgbox_create(NULL, "Enter password",
pupup_message, NULL, false);
    lv_obj_set_size(password_popup, 300, 200);
    lv_obj_center(password_popup);
    // Create and apply password popup style
    lv_style_init(&password_popup_style);
    lv_obj_add_style(password_popup, &password_popup_style, LV_PART_MAIN);
```

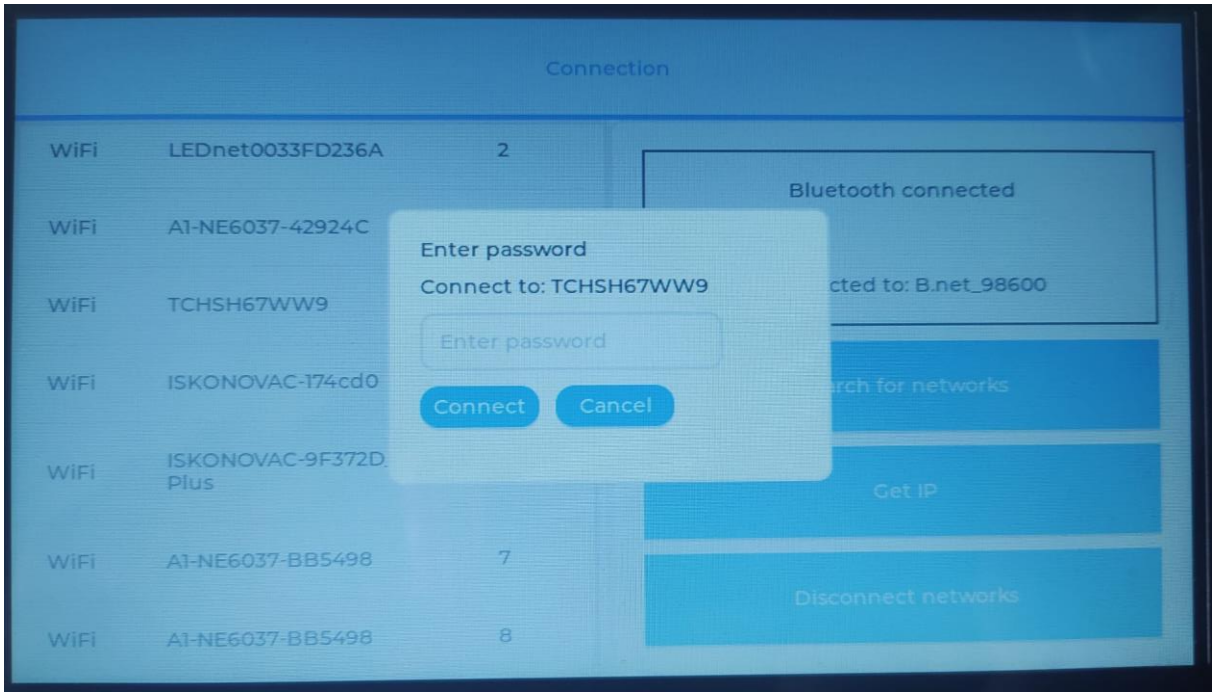
Ako je izabrana mreža WiFi, potrebno je dodati textarea na popup te dodati callback tom textarea koji će se baviti kontrolom tipkovnice.

```
    if(strcmp(network_type, "WiFi") == 0){
        // Create a text area for password input for WiFi
        lv_obj_t *password_textarea = lv_textarea_create(password_popup);
        lv_textarea_set_password_mode(password_textarea, true);
        lv_textarea_set_one_line(password_textarea, true);
        lv_textarea_set_placeholder_text(password_textarea, "Enter password");
        lv_obj_set_width(password_textarea, LV_PCT(80));
        lv_obj_center(password_textarea);
```

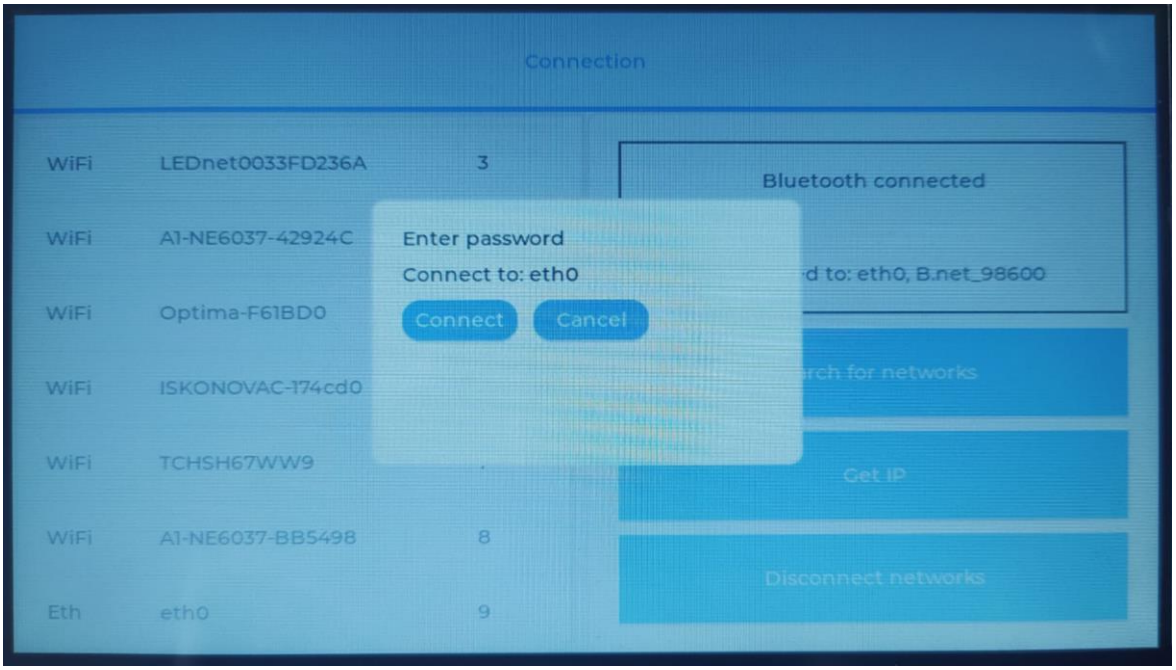
```
        lv_obj_add_event_cb(password_textarea, password_textarea_cb,
LV_EVENT_FOCUSED, password_popup);
    }
```

Isto tako potrebno je dodati gumbe za spajanje i odustajanje sa prikladnim callbak-ovima.

```
    // Create connect button
    lv_obj_t *connect_btn = lv_btn_create(password_popup);
    lv_obj_t *connect_btn_label = lv_label_create(connect_btn);
    lv_obj_set_size(connect_btn, 80, 30);
    lv_obj_align(connect_btn, LV_ALIGN_BOTTOM_LEFT, -30, -10);
    lv_label_set_text(connect_btn_label, "Connect");
    lv_obj_add_event_cb(connect_btn, password_popup_connect_btn_cb,
LV_EVENT_CLICKED, &selected_network);
    lv_obj_center(connect_btn_label);

    // Create cancel button
    lv_obj_t *cancel_btn = lv_btn_create(password_popup);
    lv_obj_t *cancel_btn_label = lv_label_create(cancel_btn);
    lv_obj_set_size(cancel_btn, 80, 30);
    lv_obj_align(cancel_btn, LV_ALIGN_BOTTOM_RIGHT, 30, -10);
    lv_label_set_text(cancel_btn_label, "Cancel");
    lv_obj_add_event_cb(cancel_btn, close_password_popup, LV_EVENT_CLICKED,
password_popup);
    lv_obj_center(cancel_btn_label);
}
```



Slika 13. Željeni izgled popup-a za WiFi



Slika 14. Željeni izgled popup-a za WiFi

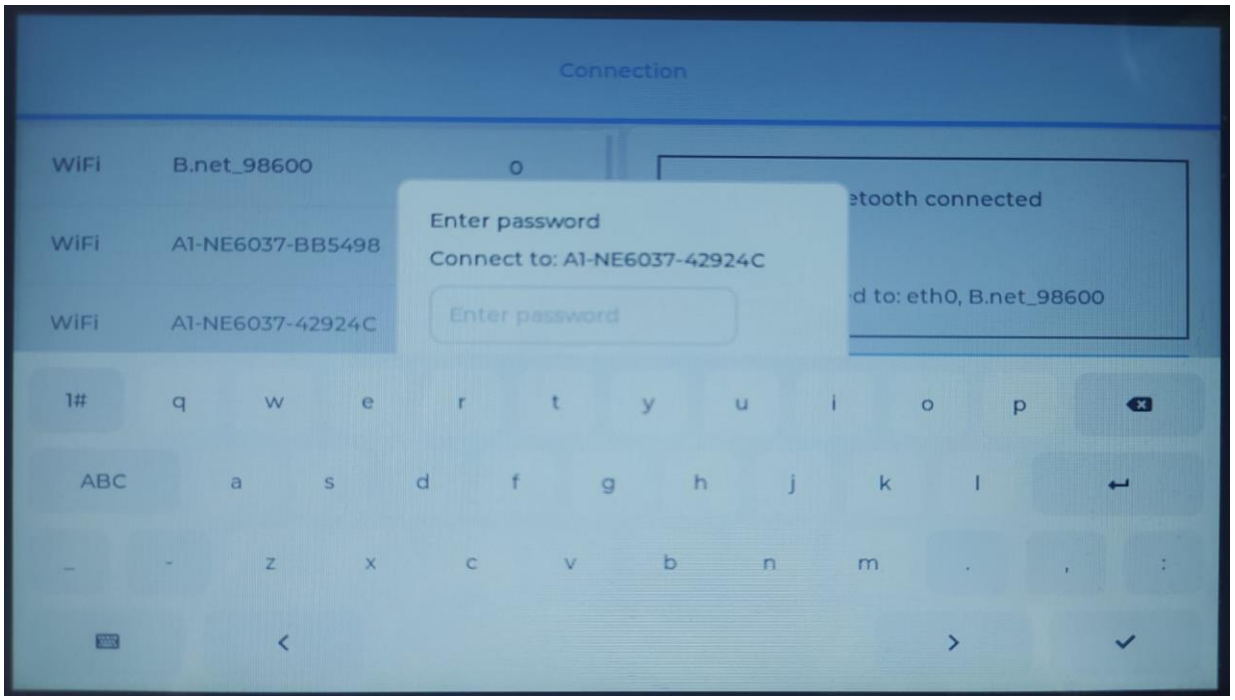
11.3.1. Upisivanje lozinke

Pritiskom na textarea na ekranu se stvara tipkovnica i popup se malo podiže gore pomoću funkcije za translaciju. Zatim se stvara touchscreen tipkovnica te joj se dodaje callback koji se poziva na LV_EVENT_READY (on se poziva kada se na tipkovnici stisne gumb s kvačicom).

```
static void password_textarea_cb(lv_event_t *e){
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* password_textarea = lv_event_get_target(e);
    lv_obj_t* password_popup = (lv_obj_t*)lv_event_get_user_data(e);  // Get the
parent popup from event data

    if (code == LV_EVENT_FOCUSED)
    {
        lv_style_set_translate_y(&password_popup_style, -25);
        lv_obj_refresh_style(password_popup, LV_PART_MAIN, LV_STYLE_PROP_ANY);

        // Create the keyboard dynamically
        lv_obj_t *keyboard = lv_keyboard_create(lv_layer_top());
        lv_keyboard_set_textarea(keyboard, password_textarea);
        lv_obj_add_event_cb(keyboard, keyboard_delete_cb, LV_EVENT_READY,
password_textarea);
    }
}
```

Kada se na tipkovnici klikne gumb s kvačicom, potrebno je izbrisati tipkovnicu te zapamtiti upisanu lozinku u strukturi izabrane mreže.

```
static void keyboard_delete_cb(lv_event_t *e){
    lv_obj_t *keyboard = lv_event_get_target(e);
    lv_obj_t *text_area = (lv_obj_t *)lv_event_get_user_data(e);

    // Clear focus and reset style translation before deleting the keyboard
    lv_obj_clear_state(text_area, LV_STATE_FOCUSED);
    selected_network.password = lv_textarea_get_text(text_area);

    // Reset the popup style before deleting the keyboard
    lv_style_set_translate_y(&password_popup_style, 0);
    lv_obj_refresh_style(lv_obj_get_parent(text_area), LV_PART_MAIN,
LV_STYLE_PROP_ANY);

    lv_obj_del(keyboard);  // Delete the keyboard
}
```



Slika 15. Tipkovnica za popup

11.3.2. „Connect“ i „Cancel“ gumbovi

Pritiskom na „Close“ gumb samo se briše sve iz memorije strukture za odabranu mrežu.

```
void close_password_popup(lv_event_t *e){
    lv_obj_t *cancel_btn = lv_event_get_target(e);
    lv_msgbox_close(lv_obj_get_parent(cancel_btn));
    //
```

```
    selected_network.name = "";     //
    selected_network.password = ""; // Clear the selected network
    selected_network.type = "";     //
}
```

Dok pritiskom na „Connect“ gumb poziva se funkcija za spajanje na mrežu te se čisti struktura za izabranu mrežu i zatvara se popup prozor.

```
static void password_popup_connect_btn_cb(lv_event_t *e){
    connect_to_network((Network*) lv_event_get_user_data(e));
    selected_network.name = "";     //
    selected_network.password = ""; // Clear the selected network
    selected_network.type = "";     //
    lv_obj_t *connect_btn = lv_event_get_target(e);
    lv_msgbox_close(lv_obj_get_parent(connect_btn));
}
```

11.3.3. Funkcija za spajanje na mrežu

Ova funkcija prima strukturu „Network“ te ju koristi za slanje stringa iz koje klijent može izvući informacije potrebne za spajanje na mrežu. „connect_info_strring“ je string u kojemu se konstruira poruka za slanje.

```
void connect_to_network(Network* network){
    // Construct string that will be sent to Raspberry PI
    std::string connect_info_string = "";
```

U string se prvo dodaje informacija o tome je li WiFi ili ethernet te ime mreže. Ovisno o tome je li mreža WiFi ili ethernet, dodaje se lozinka te na kraju se dodaje hashtag kao oznaka kraja stringa.

```
    if(network->type == "WiFi"){
        connect_info_string = "<<W:>>";
        connect_info_string = connect_info_string + "<<" + network->name + ">>";
        connect_info_string = connect_info_string + "<<" + network->password + ">>"
+ '#';
    }else if(network->type == "Eth"){   // If it's an ethernet network we don't
need a password
        connect_info_string = "<<E:>>";
        connect_info_string = connect_info_string + "<<" + network->name + ">>" +
'#';
    }
```

Konačni izgled stringa na kraju je: „<<tip>><<ime_mreže>><<lozinka_ako_je_mreža_wifi>>#“. Kada je string mrežnih informacija gotov, potrebno je poslati string klijentu, ali pošto postoji vrlo velika mogućnost da će string biti veći od 20 char-ova, prvo je potrebno string razdvojiti na više manjih dijelova što radi funkcija „BLE_chunks_array_from_string()“.

```
    int chunk_num;
    std::string string_chunks_array[20];
    delay(20);
    chunk_num = BLE_chunks_array_from_string(string_chunks_array,
connect_info_string, BLE_CHUNK_SIZE);
    BLE_send_connect_network_info(string_chunks_array, chunk_num);
}
```

Funkcija BLE_chunks_array_from_string() uzima polje za spremanje dijelova stringa(chunk-ova), string koji se treba nasjeckati i željenu veličinu chunk-ova te na kraju daje promijenjeno polje i broj chunk-ova. Kako bi se uopće odredilo koliko punih chunk-ova treba biti podijelit će se duljina stringa s veličinom chunk-a, a ako postoji ostatak potrebno ga je spremiti.

```
int BLE_chunks_array_from_string(std::string *chunks_array, std::string&
whole_string, uint16_t chunk_size){
    uint16_t string_length = whole_string.length();             //
    uint16_t num_chunks = string_length / chunk_size;           // Calculate number
of chunks
    uint16_t num_incomplete_chunk = string_length % chunk_size; // If there is a
remainder
```

Zatim je potrebno proći kroz for petlju koja daje substringove i broji chunk-ove. Ako je ostao ostatak, potrebno je i njega spremiti u polje chunk-ova.

```
    uint16_t array_entry = 0;
    for (uint16_t i = 0; i < string_length; i += chunk_size) {
        chunks_array[array_entry] = whole_string.substr(i, chunk_size);
        array_entry++;
    }

    if(num_incomplete_chunk != 0){
        chunks_array[array_entry] = whole_string.substr(string_length -
num_incomplete_chunk, num_incomplete_chunk);
    }else{
        array_entry--;
    }
```

Na kraju je potrebno vratiti broj chunk-ova u polju.

```
    return array_entry;
}
```

Kada je dobiveno polje stringova za slanje poziva se funkcija „BLE_send_connect_network_info()“. Ona uzima polje i broj stringova u polju te s pomoću for petlje periodički prolazi kroz polje i šalje stringove na karakteristiku za spajanje na mrežu (BLE_network_connect_ch). Ovaj string klijent treba interpretirati te se s pomoću informacija iz stringa pravilno spojiti. Zato klijent treba znati standard poslane poruke.

```
void BLE_send_connect_network_info(std::string* send_chunks_array, int& chunk_num){
    Serial.println("Sent to PLEA: ");
    for(int i=0; i<chunk_num; i++){
        BLE_network_connect_ch->setValue(send_chunks_array[i].c_str());
        BLE_network_connect_ch->notify();
        delay(300);
    }
}
```

12. Naredbe za klijenta

Za slanje jednostavnih naredbi odlučeno je da će se klijentu slati samo jedan bajt (formata char) te da će klijent ovisno o bajtu koji je primio, znati koja naredba je njemu poslana. Za slanje naredba koje sa sobom nose dodatne informacije potrebne za izvršenje naredbe, potrebno je implementirati sustav poput onog objašnjenog za BLE_network_connect_ch karakteristiku. Jednostavne naredbe koje se trebaju slati klijentu su „search for networks“, „get IP“ i „disconnect networks“. Slanje naredbi se izvodi tako da svaka od ovih naredbi ima gumb za naredbu(objašnjeno u prijašnjim poglavljima) te svaki gumb ima callback na istu funkciju (send_simple_command_cb), ali joj šalju drugačije bajtove ovisno o tome koji gumb je pritisnut. Bajtovi koji se šalju (to jest char-ovi) su definirani u strukturi za naredbe. Ovaj sistem omogućava laganu nadogradnju dodatnih naredbi.

Struktura za naredbe:

```
const char PLEA_commands[] = {
    /*
     *   This are simple commands that will
     *   be sent to client via buttons.
     */
    SEARCH_NETWORKS_COMMAND,    // [0] -serch for networks   ('s')
    REQUEST_IP_COMMAND,         // [1] -send IP              ('p')
    DISCONNECT_NETWORK_COMMAND  // [2] -disconnect network   ('d')
};
```

Primjer stavljanja callback-a za „search for networks“ gumba:

```
lv_obj_add_event_cb(srch_networks_btn, send_simple_command_cb, LV_EVENT_CLICKED,
(void *)&PLEA_commands[0]); // Send 's'
```

Funkcija koja šalje jednostavne naredbe putem „BLE_network_commands_ch“ karakteristike:

```
void send_simple_command_cb(lv_event_t *e){
    char command = *(char *)lv_event_get_user_data(e); // Get the command from
user_data
    std::string command_str(1, command);               //
    BLE_network_commands_ch->setValue(command_str);    //
    BLE_network_commands_ch->notify();                 //
}
```

13. Poruke od klijenta

Klijent šalje poruke HMI-u za potrebe indikacije. Već je objašnjeno kako se šalju informacije o mrežama, to je poruka za koju postoji zasebna karakteristika jer koristi drugačiji standard poruke od ostalih. Druge poruke se šalju na karakteristiku „BLE_network_message_ch“ te koriste standard izgleda poslanog stringa: „<<info1>><<info2>>…<<info_zadnji>>#“. Broj informacija koji se šalje može biti bilo koji, ali prva informacija se inače koristi za indikaciju kakav tip poruke je poslan (jesu li to informacije o npr. IP, status spojenosti mreže, itd.).

13.1. Proces primanja poruke

Kada klijent pošalje string poruku na „BLE_network_message_ch“ karakteristiku, poziva se callback „recieveMessageCallback“. Callback radi isto kao i „recieveNetworkNamesCallback“ objašnjen prije, ali koristi druge flag-ove za indikaciju završenosti primljenog stringa i za indikaciju u glavnoj petlji.

```
class recieveMessageCallback : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *receiver_characteristic)
    {
        std::string BLE_received_string = receiver_characteristic->getValue();
        BLE_string_from_chunks(BLE_received_string, &net_message_string,
&net_message_string_completed);
        if (net_message_string_completed)
        {
            net_message_string_received = true;
            net_message_string_completed = false;
        }
    }
};
```

Nakon što je cijeli string primljen, u glavnoj petlji ulazi se u ovaj dio petlje, zatim se poziva funkcija koja rastavlja string u polje koje sadrži zasebne informacije te se to polje dalje šalje u funkciju koja procesira poruku. Nakon toga se resetira stanje flag-ova i briše se string poruke iz memorije.

```
        if (net_message_string_received == true){
            BLE_array_from_string(&net_message_string, net_message_strings_array,
&net_message_num_strings);
            handle_net_message(net_message_strings_array,
&net_message_num_strings);
            net_message_string = "";
            net_message_string_completed = false;
            net_message_string_received = false;
        }
```

Za izdvajanje pojedinih informacija iz stringa, funkcija „BLE_array_from_string()“ koristi regex funkcionalnosti kako bi našla sve podatke koje se nalaze između duplih strelica (<<podatak>>) te ih stavlja u polje po redu po kojemu u nađene. (Primjer korištenja regex funkcionalnosti je detaljno objašnjeno u prijašnjim poglavljima.)

```
void BLE_array_from_string(std::string* whole_string, std::string* strings_array,
uint16_t* number_of_strings){
    /*
     *   Chops a string into individual strings and puts
     *   them in an array.
     *
     *   Function searches for string chunks that start with << and
     *   end with >>. '#' marks the end of the whole string. Puts
     *   all the chunks into a string array.
     */

    *number_of_strings = 0;

    std::regex re(R"(<<([^>]+)>>)"); // Regular expression to extract IPs
    std::smatch match;
    std::string::const_iterator searchStart((*whole_string).cbegin());

    // Extract data
    uint16_t i = 0;
    while (std::regex_search(searchStart, (*whole_string).cend(), match, re))
    {
        strings_array[i] = match[1].str();
        searchStart = match.suffix().first;
        i++;
    }
    *number_of_strings = i;
}
```

13.2. Procesiranje poruke

Procesiranje poruke radi „handle_net_message()“ funkcija. Prvo provjerava je li prvi string iz polja samo jedan char, ako je onda je to naredba za HMI.

```
void handle_net_message(std::string* strings_array, uint16_t* number_of_strings) {
    /*
    *   Function takes in an array of strings:
    *       if (the first string in the array is a single char) -> it is a command
    *       elese -> puts the elements of the array into a popup
    */
    if (strings_array[0].length() == 1) {   // If the first string from the array
is a single char then it is a command
        char command = strings_array[0][0]; // Gives the char in the first string
from the array
```

Naredbe su posebne poruke koje daju jednu od pred definiranih indikacija HMI-u. Trenutno su definirane naredbe za informacije o spojenosti mreže, odspojenosti mreža i indikaciju errora. Moguće je jednostavno dodati još naredbi dodavanjem novih case-eva u switch-u.

```
        switch (command) {
            case NETWORK_CON_MESSAGE:
                Serial.println("NET CONNECTED");
                display_network_con_status(strings_array);
                break;
            case NETWORK_DISCON_MESSAGE:
                Serial.println("NET DISCONNECTED");
                lv_label_set_text(NET_connection_status_label, "Networks
disconnected");
                connected_to_network = false;
                break;
            case NETWORK_ERROR_MESSAGE:
                Serial.println("NET ERROR");
                break;
            default:
                Serial.println("UNKNOWN MESSAGE");
                break;
        }
```

Ako prvi string iz polja nije samo jedan char, potrebno je poruku prikazati u popup prozoru. Prvi string iz polja onda postaje naziv prozora, a ostali podatci idu jedan ispod drugog na popup-u.

```
    } else {    // Create message popup
        Serial.println("MESSAGE POPUP");
        std::string message = "";
        std::string title = strings_array[0];

        for(uint16_t i=1; i<(*number_of_strings); i++){
            message += strings_array[i] + '\n';
        }

        // popup create
        // array[0] = title
        // array[1...n] = messages (each message goes in a seperate row)
        lv_obj_t* popup = lv_msgbox_create(NULL, title.c_str(), message.c_str(),
NULL, true);
        lv_obj_set_size(popup, 300, 200);
        lv_obj_center(popup);
    }
    // Empty the array
    for (uint16_t i = 0; i < *number_of_strings; i++) {
        strings_array[i].clear();
    }
    *number_of_strings = 0;
}
```

13.2.1. Prikaz statusa mrežne

Poruka za prikazivanje statusa mrežne spojenosti sadrži kao prvu informaciju char 's' te ostale informaciju u poruci govore koje su mreže spojene (Primjer izgleda naredbe: <<C>><<E: prva ethernet mreža na koju je robot spojen, W: druga WiFi mreža na koju je robot spojen>>#). Za prikazivanje ovih informacija poziva se funkcija „display_network_con_status()“, ona samo uzima drugu informaciju iz polja te ju stavlja kao tekst od widgeta „NET_connection_status_label“.

```
void display_network_con_status(std::string* status_array){
    /*
    *   Status array standard:
    *   [0] - NETWORK_CON_MESSAGE
    *   [1] - Network status message
    */
    // Example to send to NETWORK_MESSAGE_CH_UUID:
    // <<C>><<E: Network1, W: Network2>>#

    std::string con_message = status_array[1];
    lv_label_set_text(NET_connection_status_label, con_message.c_str());
    connected_to_network = true;
}
```

Ako robot nije spojen ni na jednu mrežu, onda šalje naredbu da nije spojen ni na jednu mrežu (<<D>>#), ta naredba postavlja tekst widgeta „NET_connection_status_label“ u "Networks disconnected".

```
            case NETWORK_DISCON_MESSAGE:
                Serial.println("NET DISCONNECTED");
                lv_label_set_text(NET_connection_status_label, "Networks
disconnected");
                connected_to_network = false;
                break;
```

13.2.2. Prikaz IP-eva i drugih poruka

Kada se klikne na gumb „Get IP“ robotu se preko „BLE_network_commands_ch“ karakteristike šalje char 'p'. Kada robot zaprimi tu naredbu, šalje nazad string s nazivom koji ide u zaglavlje popup prozora i IP informacije na „BLE_network_message_ch“ (Primjer izgleda stringa IP informacija: <<IP info>><<lo: IP>><<wlan0: IP>><<eth0: IP>>#). IP poruka se procesira kao i svaka druga poruka koja nije naredba, prva informacija ide u zaglavlje, ostale idu u popup jedna ispod druge. Slanje svih drugih poruka HMI-u koristi isti princip. (Za dizajn izgleda poup-a savjetuje se pregled literature o dizajnu. [9])

14. Klijent

Klijent ovog HMI sučelja je PLEA afektivni robot. Robot koristi Raspberry PI 4 s Linux operativnim sistemom (Raspberry Pi OS). Kako bi HMI mogao kontrolirati stvari na robotu, potrebno je implementirati način na koji će robot komunicirati s HMI-om te API koji će HMI koristiti za kontrolu robota. Za te potrebe odlučeno je da će robot cijelo vrijeme run-ati python skriptu koja će se baviti ovime. Za potrebe BLE komunikacije koristi se „bluepy“ biblioteka, a za kontroliranje mrežne spojenosti koristi se standardni command line tool za ovaj Linux distro, „nmcli“.

14.1. Spajanje klijenta na server

U ovom sistemu zadaća klijenta je da odgovara na serverove zahtjeve i da održava konekciju sa serverom. Zato na početku glavne petlje, potrebno je definirati sve varijable potrebne za stvaranje BLE konekcije i za praćenje mrežne spojenosti. Pošto za BLE komunikaciju je potrebno konstantno provjeravati ima li novih poruka, ovaj program će imati više „thread-ova“. Jedan thread će se baviti glavnim tokom programa, dok će se drugi baviti slušanjem poruka od servera i slanjem statusa mrežne spojenosti.

```
if __name__ == "__main__":
    BLE_main()

def BLE_main():
    # Variables #
    peripheral = None
    previous_connected_networks = ""

    # Declare wanted characteristics
    global network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate
    notification_thread = None  # Thread for handling incoming notifications
    global stop_event
    global currently_connected_networks
    ###
```

Varijabla „stop_event“ će se koristiti kao indikator spojenosti BLE konekcije, ako konekcija pukne ta varijabla će biti „set-ana“. Petlja za slušanje poruka koristi ovu varijablu za indikaciju glavnoj petlji o diskonekciji.

```
    stop_event = threading.Event()
```

Nakon deklaracije svih potrebnih varijabli pokreće se beskonačna petlja u kojoj se procesira logika spojenosti te komunikacija sa serverom. Na početku svakog kruga prvo se gleda je li varijabla „peripheral“ postavljena u nešto. Varijabla „peripheral“ je objekt koji drži ime uređaja na koji je

klijent spojen (u ovom slučaju to je HMI). Ako klijent nije spojen na HMI poziva se funkcija koja spaja klijenta na HMI te ako se uspije spojiti, vraća ime servera na koji se spojila kao varijablu peripheral.

```
    while True:
        if peripheral is None:  # We are not connected via BLE
            peripheral = BLE_connect_to_device()  # Try to connect
```

Funkcija koja spaja klijenta na server:

```
# BLE connect functions
def BLE_connect_to_device():
    try:
        print("Attempting to connect to ESP32...")
        peripheral = btle.Peripheral(DEVICE_MAC_ADDRESS)
        print("Connected to ESP32")
        return peripheral
    except Exception as e:
        print(f"Failed to connect: {e}")
        return None
```

Nakon pokušaja spajanja uređaja potrebno je provjeriti je li se uspio spojiti. Ako se uspio spojiti, potrebno je spojiti se na željene karakteristike u servisu („network_service“).

```
            if peripheral is not None:  # If we succeeded
                # Connect to wanted characteristics #
                network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate = network_service(peripheral)
```

Funkcija koja obavlja spajanje na karakteristike servisa za mrežnu spojenost („network_service“):

```
def network_service(peripheral):
    # Connect to the service
    network_service = peripheral.getServiceByUUID(UUID(NETWORK_SERVICE_UUID))

    # Send characteristics #
    network_names_ch = network_service.getCharacteristics(NETWORK_NAMES_CH_UUID)[0]
    network_message_ch =
network_service.getCharacteristics(NETWORK_MESSAGE_CH_UUID)[0]
    ###

    # Receive characteristics #
    network_connect_ch =
network_service.getCharacteristics(NETWORK_CONNECT_CH_UUID)[0]
    network_commands_ch =
network_service.getCharacteristics(NETWORK_COMMANDS_CH_UUID)[0]
```

```
    delegate = NetworkNotificationDelegate(network_connect_ch, network_commands_ch)
    peripheral.setDelegate(delegate)

    network_connect_ch_handle = network_connect_ch.getHandle() + 1
    peripheral.writeCharacteristic(network_connect_ch_handle, b'\x01\x00',
withResponse=True)  # Enable notifications

    network_commands_ch_handle = network_commands_ch.getHandle() + 1
    peripheral.writeCharacteristic(network_commands_ch_handle, b'\x01\x00',
withResponse=True)  # Enable notifications

    return network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate
```

Na karakteristike koje će slati podatke HMI-u potrebno je dodati bitove koji omogućavaju notifikacije na strani servera, dok na karakteristike koje će primati podatke od HMI-a potrebno je dodati klasu koja će se baviti primanjem notifikacija, „NetworkNotificationDelegate“(biti će objašnjeno u kasnijem podpoglavlju).

Nakon što se povežu karakteristike, potrebno je osigurati da „stop_event“ varijabla nije set-ana. Poslije toga stvara se novi thread za BLE notifikacije, zvani „notification_loop“. Ovaj thread će koristiti varijablu „peripheral“ za korištenje bluepy funkcija i varijablu „stop_event“ za signalizaciju spojenosti BLE-a glavnoj petlji.

```
                stop_event.clear()
                notification_thread = threading.Thread(target=notification_loop,
args=(peripheral, stop_event))
```

Na kraju je potrebno notifikacijski thread napraviti da je „daemon“ (što znači da će se taj thread terminirati po zatvaranju glavnog thread-a) i započeti rad thread-a.

```
                notification_thread.daemon = True
                notification_thread.start()
```

U suprotnom ako se klijent nije uspio spojiti na server, pokušat će opet za pet sekundi i to će raditi sve dok se ne spoji.

```
            else:  # If we didn't succeed, retry
                print("Retrying in 5 seconds...")
                time.sleep(5)
```

14.2. Procedura nakon spajanja

Nakon što je uređaj spojen može se ući u dio glavne petlje koji obavlja zadatke kada je uređaj spojen. Na svakom početku novog kruga potrebno je provjeriti je li „stop_event“ bio set-an, ako je onda je

potrebno terminirati notification_thread pomoću .join() funkcije te nakon toga probat odspojiti klijent od servera pomoću .disconnect() funkcije.

```
        else:  # We are connected via BLE
            if stop_event.is_set():  # If the notification thread isn't running
                print("Connection lost, attempting to reconnect...")
                if notification_thread:
                    notification_thread.join()
                try:
                    peripheral.disconnect()  # Ensure disconnection
                except Exception as e:
                    print(f"Error during disconnection: {e}")
                peripheral = None
                time.sleep(5)
```

Ako „stop_event“ nije set-an, u svakom krugu glavne petlje provjerava se je li dobiven flag „connect_network_string_finished“, on signalizira da je došla naredba za spajanje na mrežu. Kada flag postane „True“ odrađuje se funkcija koja spaja robota na mrežu uz pomoć informacija dobivenih preko stringa iz „network_connect_ch“ karakteristike. Ako se tijekom toga dogodi diskonekcija BLE-a, „stop_event“ varijabla postaje set-ana te cijeli proces spajanja klijenta i servera iz prijašnjeg podpoglavlja se ponavlja.

```
            else:
                try:  # Main loop handling
                    if delegate.connect_network_string_finished:    # Check if we
need to connect to a network
                        connect_to_network(delegate.connect_network_string)

                        # After connecting reset flag and string
                        delegate.connect_network_string = ""
                        delegate.connect_network_string_finished = False

                except btle.BTLEDisconnectError:
                    print("Detected disconnection during main loop tasks.")
                    stop_event.set()
```

14.3. Spajanje na mrežu

Spajanje na mrežu se događa kada se primi string s informacijama za spajanje od karakteristike „network_connect_ch“, kada se to dogodi flag „connect_network_string_finished“ postane „True“ što u „BLE_main()“ thread-u poziva „connect_to_network()“ funkciju. (Proces je detaljnije objašnjen u točci 15.5.) Prvi korak funkcije za spajanje na mrežu je micanje duplih strelica koje odvajaju pojedine informacije, nakon toga te pojedine informacije se stavljaju u polje „network_info_list“.

```
def connect_to_network(network_info):
    # Incoming string looks like this:
```

```
    # If it's WiFi: <<W:>>><<NetworkName>>><<Password>>
    # If it's Eth:  <<E:>>><<NetworkName>>
    #
    # Function takes in a string, chops it into type,
    # name and password and connects to it
    #
    network_info = network_info[:-3]  # Removes >>#
    network_info = network_info[2:]  # Removes <<
    network_info_list = network_info.split('>>><<')  # Makes it a list
```

Zatim je potrebno validirati da su primljeni podatci dobro strukturirani. Te izdvojiti tip mreže i ime mreže u zasebne varijable.

```
    if len(network_info_list) < 2:  # Validate the list length
        print("Error: network_info string is not in the expected format.")
        return
    network_type = network_info_list[0]
    network_name = network_info_list[1]
```

Ako je mreža WiFi, potrebna je i lozinka za spajanje. Ovi podatci se koriste za spajanje na mrežu koristeći „nmcli“ subproces. Nakon toga potrebno je javiti je li spajanje bilo uspješno ili neuspješno.

```
    if network_type == "W:":
        network_password = network_info_list[2]
        # Connect to WiFi
        result = subprocess.run(['nmcli', 'device', 'wifi', 'connect',
network_name, 'password', network_password], capture_output=True, text=True)
        print(result)
        if result.returncode == 0:
            print(f"Successfully connected to Wi-Fi network {network_name}")
        else:
            print(f"Failed to connect to Wi-Fi network {network_name}")
            print(result.stderr)
```

Ako je mreža ethernet, nije potrebna lozinka za spajanje pa se ona ni ne mora slati. Ovi podatci se koriste za spajanje na mrežu koristeći „nmcli“ subproces. Nakon toga potrebno je javiti da li je spajanje bilo uspješno ili neuspješno.

```
    elif network_type == "E:":
        # Connect to Ethernet
        result = subprocess.run(['nmcli', 'device', 'connect', network_name],
capture_output=True, text=True)
        if result.returncode == 0:
            print(f"Successfully connected to Ethernet network {network_name}")
        else:
            print(f"Failed to connect to Ethernet network {network_name}")
            print(result.stderr)
    else:
        print("Error: Unknown network type.")
```

14.4. Thread za slušanje notifikacija i slanje mrežnog statusa

Kako se poruke koje HMI šalje ne bi slučajno propustile potrebno je konstantno slušati za nove notifikacije. Uz to potrebno je stalno provjeravati status mrežne spojenosti robota jer mu je moguće promijeniti mrežnu spojenost bez da se to napravi preko HMI-a. Pošto se te dvije provjere trebaju stalno raditi odlučeno ih je obje smjestiti u jedan thread. U tu svrhu postoji thread „notification_loop“, on se izvršava cijelo vrijeme dok je robot spojen na server. Na početku je potrebno deklarirati varijablu „previous_connected_networks“ u svrhu praćenja promjena u mrežnoj spojenosti.

```
def notification_loop(peripheral, stop_event):
    previous_connected_networks = ""
```

Ako „stop_event“ nije set-an čeka se za nove notifikacije jednu sekundu. Ovo je dio koda koji će pozvati „handleNotification()“ funkciju iz „NetworkNotificationDelegate()“ klase (objašnjeno u idućem podpoglavlju), ako prihvati notifikaciju. Kada prođe sekunda nastavlja se u drugi dio koda.

```
    while not stop_event.is_set():
        try:
            if peripheral.waitForNotifications(1.0):  # Constantly listen for
notifications
                continue
```

Drugi dio ove petlje gleda promjene u mrežnoj spojenosti. Pomoću „subprocess“ biblioteke moguće je iz python skripte pozivati druge skripte i programe. Naredba „subprocess.run()“ uzima za prvi parametar polje koje sadrži pojedine informacije koje trebaju biti stavljene u terminal. Ovdje je toj funkciji dano polje ['nmcli', '-t', '-f', 'NAME', 'connection', 'show', '--active'], te će se u terminal staviti string izgleda: „nmcli -t -f' NAME connection show --active“. Ova naredba na kraju vraća string sa svim spojenim mrežama. Ako se dogodila promjena u mrežnoj spojenosti od zadnjeg prolaska kroz petlju, poziva se funkcija koja šalje to HMI-u("BLE_send_networks_connection_status_message()“). Ako se slučajno dogodi diskonekcija BLE komunikacije za vrijeme prolaska kroz ovaj proces, događa se ekscepcija i „stop_event“ varijabla postaje set-ana, što signalizira drugi thread i blokira petlju ovog thread-a.

```
            # Constantly check if any new networks have connected or disconnected
            currently_connected_networks = subprocess.run(['nmcli', '-t', '-f',
'NAME', 'connection', 'show', '--active'], capture_output=True, text=True).stdout
            if previous_connected_networks != currently_connected_networks:     #
Check if any network connection has changed
                BLE_send_networks_connection_status_message(currently_connected_ne
tworks)
                previous_connected_networks = currently_connected_networks
        except btle.BTLEDisconnectError:
            print("Notification loop detected disconnection.")
            stop_event.set()  # Signal the main thread that disconnection has
occurred
```

14.4.1. Slanje statusa mrežne spojenosti HMI-u

Standard poruke za status mrežne spojenosti je prihvaćen među serverom i klijentom. Da bi HMI mogao dobro koristiti informacije koje mu robot pošalje dogovoren je izgled poruke kao „<<C>><<poruka o statusu spojenosti>>#“ za poruku o spojenim mrežama te kao <<D>># ako nije nijedna mreža spojena. Funkcija pri pozivanju dobiva string koji sadrži sve trenutno spojene mreže, preko varijable „connected_networks“, ali taj string je potrebno procesirati jer dolazi sa dodatnim informacijama koje su nepotrebne i još je potrebno razdvojiti zasebne mreže u polje radi lakšeg konstruiranja poruke.

Primjer izgleda ulaznog stringa:

„lo\nIME_MREŽE1\nIME_MREŽE2\nIME_MREŽE3\n“

Primjer željenog polja iz gornjeg stringa:

Polje = ["IME_MREŽE1", "IME_MREŽE2","IME_MREŽE3"]

String je potrebno očistiti od neželjenih dijelova te razdvojiti pojedinačna imena mreža iz stinga u polje, to se radi sa „.split()“ funkcijom te se koristi „\n“ za razdvajanje pojedinih imena.

```
def BLE_send_networks_connection_status_message(connected_networks):
    if connected_networks[-3:] == "lo\n":   # Removes loopback network if it's in
the string
    connected_networks = connected_networks[:-3]
    connected_networks = connected_networks.rstrip("\n")
    connected_networks = connected_networks.split("\n")     # Gets a list of
connected networks
```

Varijabla „connected_networks“ koja je prije bila string sada postaje polje stringova. Potrebno je provjeriti, ako nije bila nađena niti jedna spojena mreža u polju nula neće biti ničega. Tako se zna da nije ni jedna mreža spojena pa je krajnja poruka: „<<D>>#“.

```
    if connected_networks[0] == "":     # If we aren't connected to any networks
        #print("Networks disconnected")
        BLE_connected_networks_message  = "<<D>>#"      # Return networks
disconnected message
```

Ako postoje mreže na koje je robot spojen konstruirati će se drugačiji string. String koji će biti konstruiran ima izgled: „<<C>><<Connected to: ime_mreže1, ime_mreže2, …,ime_mrežeN>>#“.

```
    else:
        BLE_connected_networks_message = "<<C>><<Connected to: "
```

```
        BLE_connected_networks_message += ', '.join(connected_networks)
        BLE_connected_networks_message += ">>#"
```

Kada je string poruke konstruiran potrebno ga je poslati s pomoću „BLE_chop_string_to_chunks()“ funkcije i poslati ga na „network_message_ch“ karakteristike s pomoću „BLE_send_array()“ funkcije.

```
    BLE_connected_networks_message_array =
BLE_chop_string_to_chunks(BLE_connected_networks_message, 20)
    #print(BLE_connected_networks_message_array)
    BLE_send_array(BLE_connected_networks_message_array, network_message_ch)    #
Send the connection message
```

„BLE_chop_string_to_chunks()“ radi isto kao i „BLE_chunks_array_from_string()“ funkcija na HMI-u, samo što je prevedena da radi u python-u. Funkcija uzima string te u for petlji siječe string na manje substringove veličine „chunk_size“ (20 char-ova po standardu) te ih stavlja u polje jednu za drugom. Ako postoji ostatak manji od veličine „chunk_size“, on se stavlja na kraj polja.

```
# BLE #
def BLE_chop_string_to_chunks(string, chunk_size):
    #
    # Takes in a string and chops it into an arraz
    # of chunks, size of chunk_size.
    #
    chunks_array = []
    string_length = len(string)

    # Size of chunks we can send over BLE is 20 bytes
    for i in range(string_length // chunk_size):
        chunks_array.append(string[i * chunk_size : (i + 1) * chunk_size])

    if string_length % chunk_size != 0: # If there is a leftover smaller than 20
chars
        chunks_array.append(string[-(string_length % chunk_size):])
    return chunks_array
```

„BLE_send_array()“ funkcija je vrlo jednostavna. Funkcija uzima polje koje se želi poslati i karakteristika na koju se želi poslati te periodički šalje svaki string iz polja s vremenom između slanja koje je 0.003 sekunde. (Vremenski razmak postoji kako se ne bi prepunio buffer karakteristike.)

```
def BLE_send_array(array, characteristic):
    for entry in array: # Send the chunks over BLE
        characteristic.write(bytes(str(entry), 'utf-8'))
        time.sleep(0.003)
```

14.5. Rad sa notifikacijama

Kako bi primanje notifikacija moglo raditi potrebno je stvoriti klasu koja se bavi delegacijom notifikacija. Klasi je potrebno dati delegaciju BLE notifikacija s pomoću „btle.DefaultDelegate“. U klasi postoje dvije definicije, definicija inicijalizacije (gdje se definiraju karakteristike koje koriste klasu, a to su karakteristike koje primaju poruke od HMI-a i varijable koje su potrebne klasi) te definicija funkcije za procesiranje dobivenih poruka.

```
class NetworkNotificationDelegate(btle.DefaultDelegate):
    def __init__(self, network_connect_ch, network_commands_ch):
        super().__init__()
        self.network_connect_ch = network_connect_ch
        self.network_commands_ch = network_commands_ch
        self.connect_network_string = ""
        self.connect_network_string_finished = False
    def handleNotification(self, cHandle, data):
```

Ako notifikacijski thread primijeti notifikaciju, poziv se funkcija „handleNotification()“ iz ove klase. Ova funkcija dobiva „handle“ karakteristike od koje je karakteristike dobivena notifikacija te podatke koji su dobiveni. Ako su podatci došli od „network_connect_ch“ karakteristike, to će biti dugački string te je potrebno spremati dolazeće podatke u string tako da se nadodaju u svakom novom ciklusu. Ako je na kraju dobivenog stringa '#' to znači da je to kraj poruke i onda se to signalizira petlji „BLE_main()“ koja će dalje procesirati taj string.

```
        if cHandle == self.network_connect_ch.getHandle():
            self.connect_network_string += data.decode('utf-8')
            if self.connect_network_string[-1] == '#':
                self.connect_network_string_finished = True
```

Ako je karakteristika s koje je došla notifikacija „network_commands_ch“ onda se zna da će „data“ biti veličine samo jednog char-a pa se odmah može procesirati ta poruka. To se radi pomoću funkcije „handle_network_commands()“.

```
        elif cHandle == self.network_commands_ch.getHandle():
            handle_network_commands(data.decode('utf-8'))
```

Tok ove funkcije je vrlo jednostavan, korištenjem python-ove „match“ funkcije, koja se može koristiti slično kao i „switch“ funkciji iz C jezika, odrađuju se drugačije stvari ovisno o primljenom slovu s karakteristike. Funkcija je namjerno strukturirana za lagano dodavanje novih naredbi.

```
def handle_network_commands(network_command):
    match network_command:
        case 's':  # Search for networks
```

```
            print("Search for networks")
            networks_string = get_networks_string()
            networks_array = BLE_chop_string_to_chunks(networks_string, 20)
            BLE_send_array(networks_array, network_names_ch)
            return
        case 'p':  # Get IP
            print("Get IP")
            IPv4_string = get_ipv4_addresses()
            IPv4_array = BLE_chop_string_to_chunks(IPv4_string, 20)
            BLE_send_array(IPv4_array, network_message_ch)
            return
        case 'd':  # Disconnect from networks
            print("Disconnect from all networks")
            disconnect_all_networks()
            return
        case _:
            print("Unknown command")
            return
###
```

14.5.1. Naredba „Search for networks“

Ova naredba se poziva slanjem slova 's'. Funkcija prvo dobiva string mrežnih podataka pomoću „get_networks_string()“ funkcije te je nasjecka u polje i šalje na karakteristiku „network_names_ch“ (isto kao i u 15.3.1. podpoglavlju).

Funkcija „get_networks_string()“ koristi „nmcli“ kako bi dobila string imena svih WiFi mreža koje robot može naći. Potrebno je razdvojiti sva imena te ih staviti u polje „wifi_networks“.

```
def get_networks_string():
    networks = ""
    # List available Wi-Fi networks
    wifi_result = subprocess.run(['nmcli', '-t', '-f', 'SSID', 'device', 'wifi',
'list'], capture_output=True, text=True)
    wifi_networks = wifi_result.stdout.split('\n')
```

Kada se to napravi kreće se konstruiranje stringa koji će funkcija vraćati. Za svaku WiFi mrežu u string se dodaje: „W:<<ime_mreže>>“.

```
    for ssid in wifi_networks:
        if ssid.strip():  # Skip empty SSIDs
            networks += f"W:<<{ssid.strip()}>>"
```

Zatim se radi ista stvar za ethernet mreže te se na kraju dodaje '#' i vraća se napravljeni string.

```
    # Check if eth0 is physically connected
```

```
    result = subprocess.run(['sudo', 'ethtool', 'eth0'], capture_output=True,
text=True)
    for line in result.stdout.split('\n'):
        if 'Link detected:' in line:
            # Check if the link is detected as yes
            if 'yes' in line:
                networks += f"E:<<eth0>>"
    networks += '#'  # Tells the receiver the string is done
    return networks
```

Kada se dobije string nađenih mreža, on se siječe na chunk-ove i šalje HMI-u.

14.5.2. Naredba „Get IP“

Ova naredba se poziva slanjem slova 'p'. Funkcija prvo dobiva string IP podataka s pomoću „get_ipv4_adresses()“ funkcije te je nasjecka u polje i šalje na karakteristiku „network_message_ch“. Korištenjem subprocesa koji u terminal poziva „ip addr“, dobiva se string informacija o IP adresama.

```
def get_ipv4_addresses():
    try:
        # Run the ip command to get IP address information
        result = subprocess.run(['ip', 'addr'], capture_output=True, text=True)
```

String je potrebno parsirati tako da iz stringa izvučemo polje koje sadrži ime vrste interfejsa (lo – loopback, wlan0 – WiFi, eth0 – ethernet) i odgovarajuće IP adrese koje su na njima.

```
        interfaces = {}
        interface_name = None

        # Parse the output to extract IPv4 addresses
        for line in result.stdout.split('\n'):
            if line.startswith(' '):
                # This line contains IP address information
                if 'inet ' in line:  # Only consider 'inet' for IPv4
                    ip_address = line.strip().split()[1]
                    if interface_name:
                        interfaces[interface_name].append(ip_address)
            else:
                # This line contains the interface name
                if line:
                    parts = line.split(': ')
                    if len(parts) > 1:
                        interface_name = parts[1].split('@')[0]
                        if interface_name not in interfaces:
                            interfaces[interface_name] = []
```

Kada se dobije polje sa svim informacijama konstruira se string koji će se slati karakteristici, a ima izgled: „<<IP info>><<lo: loopback IP>><<wlan0: WiFi IP>><<eth0: ethernet IP>>#“.

```
        IPs_string = "<<IP info>>"
```

```
        for interface, addresses in interfaces.items():
            IPs_string += f"<<{interface}: {', '.join(addresses)}>>"
        IPs_string += '#'
        return IPs_string
    except Exception as e:
        print(f"Error retrieving IP addresses: {e}")
        return ""
```

Kada se dobije string IP informacija, on se siječe na chunk-ove i šalje HMI-u.

14.5.3. Naredba „Disconnect from networks“

Ova naredba se poziva slanjem slova 'd'. Ovo je jednostavna naredba koja samo poziva funkciju „disconnect_all_networks()“. Ova funkcija nalazi sve trenutno spojene veze na robotu s pomoću funkcije „get_active_networks()“ te se odspaja od svih mreža, osim loopback mreže.

```
def disconnect_all_networks():
    active_connections = get_active_networks()
    for uuid, device in active_connections:
        if device != "lo":
           result = subprocess.run(['nmcli', 'device', 'down', device],
capture_output=True, text=True)
           if result.returncode == 0:
              print(f"Disconnected {device} with UUID {uuid}")
           else:
              print(f"Failed to disconnect {device} with UUID {uuid}:
{result.stderr}")
```

Funkcija „get_active_networks()“ koristi „nmcli” kako bi dobila string svih trenutno spojenih mreža te parsira taj string i vraća nazad polje koje sadrži sve trenutno spojene mreže.

```
def get_active_networks():
    result = subprocess.run(['nmcli', '-t', '-f', 'UUID,DEVICE', 'connection',
'show', '--active'],
                            capture_output=True, text=True)
    if result.returncode != 0:
        print(f"Error getting active connections: {result.stderr}")
        return []

    active_connections = []
    for line in result.stdout.strip().split('\n'):
        if line:
            uuid, device = line.split(':')
            active_connections.append((uuid, device))

    return active_connections
```

15. Zaključak

Zaključak ovog rada sumira ključne aspekte dizajna i implementacije HMI sustava za interakciju s afektivnim robotom PLEA. Provedena analiza i razvoj obuhvatili su izbor hardvera, dizajn grafičkog sučelja, uspostavljanje komunikacijskih protokola te implementaciju softverskog rješenja korištenjem LVGL i LovyanGFX biblioteka. Poseban naglasak stavljen je na integraciju BLE komunikacije, koja omogućuje učinkovitu i niskoenergetsku razmjenu podataka između HMI-a i robota. Demonstracija dodavanja nove funkcionalnosti, kao što je kontrola mrežnih postavki, pokazala je kako ovaj sustav može biti lako nadograđen s dodatnim značajkama, čime se osigurava njegova fleksibilnost i prilagodljivost budućim potrebama. Time se ovaj rad ne završava samo kao dokumentacija trenutnih rješenja, već služi kao temelj za daljnji razvoj i optimizaciju HMI sustava u različitim primjenama. Među budućim smjerovima razvoja koji su planirani za ovaj HMI su kontrola postavki PLEA robota, puna integracija linux terminala preko HMI-a, prikaz kamere s PLEA robota i druge nove funkcionalnosti.

LITERATURA

[1] LVGL documentation (https://docs.lvgl.io/8.3/), pristupljeno 16.4.2024.

[2] 5 Tips for Building Embedded UI with LVGL and PlatformIO (https://punchthrough.com/5-tips-for-building-embedded-ui-with-lvgl-and-platformio/), pristupljeno 23.6.2024.

[3] Impressive UIs for ESP Projects with SquareLine Studio (https://www.espressif.com/en/news/ESP_UIs_SquareLine_Studio), pristupljeno 14.5.2024.

[4] STM32-LTDC, LCD-TFT, LVGL(MCU3) (https://fastbitlab.com/lvgl-and-simulator-download/), pristupljeno 23.5.2024.

[5] CrowPanel ESP32 HMI 7.0 Display (https://www.elecrow.com/wiki/esp32-display-702727-intelligent-touch-screen-wi-fi26ble-800480-hmi-display.html), pristupljeno 22.5.2024.

[6] LVGL positions, sizes, and layouts (https://docs.lvgl.io/8.3/overview/coords.html), pristupljeno 26.6.2024.

[7] LVGL Documentation 8.3 (https://docs.lvgl.io/8.3/_downloads/39cea4971f327964c804e4e6bc96bfb4/LVGL.pdf), pristupljeno 10.7.2024.

[8] ESP32 Series Datasheet Version 4.7 (https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf), pristupljeno 1.7.2024.

[9] The Essential Guide to User Interface Designe (https://profagaskar.wordpress.com/wp-content/uploads/2020/03/wiley_the_essential_guide_to_user_interf.pdf), pristupljeno 5.6.2024.

PRILOZI

I. main.cpp (C++ kod HMI-a)
(dodatni potrebni header fajlovi)

I.I. BLE_config.h

I.II. gfx_config.h

I.III. UI_parameters.h

I.IV. hardware.h

II. PLEA_BLE_network.py (python kod za PLEA robota)

I. Python kod HMI-a

```
// repository link: https://github.com/xXDemionXx/PLEA_HMI //

// HEADERS //

// libraries
#include <lvgl.h>
#include <gfx_conf.h>
#include <regex>
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

// custom headers
#include <hardware.h>
#include <UI_parameters.h>
#include <BLE_conf.h>
//#include <BLE2901/BLE2901.h>    // To solve some other time

////////////////////////////////////////////////////////////////////////////////

// TASK HANDLES //
TaskHandle_t LVGL_handler_task; // goes on core0

// VARIABLES //

// Network
std::vector<std::string> networkNames; // names of networks WiFi or Ethernet
std::string network_names_string;
// Network flags
bool networks_string_received = false;
bool network_string_completed = false; // must be 0 at the start
bool connected_to_network = false;

// network message
std::string net_message_string;
std::string net_message_strings_array[20];   // max size - 20 strings
uint16_t net_message_num_strings;
// network message flags
bool net_message_string_received = false;
bool net_message_string_completed = false; // must be 0 at the start

// Struct that holds network information
struct Network
{
    std::string name;     //
```

```
    std::string type;     // "WiFi" or "Ethernet"
    std::string password; //
    byte table_entry_nmb; //
};
std::vector<Network> networks;  // vector that holds network names

Network selected_network;       // holds the network that we have selected in the
table
Network connected_network;      // holds the network that we will be connected to

// LVGL VARIABLES //

static lv_disp_draw_buf_t draw_buf;
static lv_color_t disp_draw_buf1[screenWidth * screenHeight / 8];
// static lv_color_t disp_draw_buf2[screenWidth * screenHeight / 8];     // to much
RAM usage
static lv_disp_drv_t disp_drv;

// screens
lv_obj_t *default_screen; // create root parent screen

// widgets

lv_obj_t *connection_table_backdrop;
lv_obj_t *connection_tab;
lv_obj_t *main_tabview;
lv_obj_t *BLE_connection_status_label;
lv_obj_t *NET_connection_status_label;
lv_obj_t *connection_table;
lv_obj_t *password_popup;

// styles

static lv_style_t password_popup_style;

///////////////////////////////////////////////////////////////////////////////

// BLE //

// variables

BLEServer *BLE_HMI_server = NULL;
BLEService *BLE_network_service = NULL;
BLECharacteristic *BLE_network_names_ch = NULL;
BLECharacteristic *BLE_network_connect_ch = NULL;
BLECharacteristic *BLE_network_message_ch = NULL;
BLECharacteristic *BLE_network_commands_ch = NULL;
BLEAdvertising *BLE_advertising = NULL;
bool BLEdeviceConnected = false;
```

```
bool BLEolddeviceConnected = true;

const char PLEA_commands[] = {
    /*
     *   This are simple commands that will
     *   be sent to PLEA via buttons.
     */
    SEARCH_NETWORKS_COMMAND,    // [0] -serch for networks
    REQUEST_IP_COMMAND,         // [1] -send IP
    DISCONNECT_NETWORK_COMMAND  // [2] -disconnect network
};

////////////////////////////////////////////////////////////////////////////////

// FUNCTION DECLARATIONS //

// gfx functions
void touchpad_read(lv_indev_drv_t *indev_driver, lv_indev_data_t *data);
void disp_flush(lv_disp_drv_t *disp, const lv_area_t *area, lv_color_t *color_p);

// UI init functions
void init_tabs();
void init_connection_tab();
void init_macros_tab();
void init_settings_tab();
void init_PLEA_settings_tab();
// UI functions
void no_connections_available(lv_obj_t* backdrop);

// LVGL task
void LVGL_handler_function(void *pvParameters);

// network name operations
void put_network_names_in_table(const std::vector<Network> &networkss);

// IP string operation
void BLE_array_from_string(std::string* net_message_string, std::string*
net_message_strings_array, uint16_t* number_of_strings);
int BLE_chunks_array_from_string(std::string* chunks_array, std::string&
whole_string, uint16_t chunk_size);    // returns number of entrys in array

// BLE functions
void init_BLE();
void init_BLE_network_service();
void BLE_string_from_chunks(std::string chunk, std::string *storage_string, bool
*completed_message_indicator);
void BLE_network_names_from_string(std::string networks_string,
std::vector<Network> &networks);
void send_simple_command_cb(lv_event_t *e);
```

```
void connect_to_network(Network* network);
//void disconnect_from_network();   //DEPRICATED
void BLE_send_connect_network_info(std::string* send_chunks_array, int& chunk_num);

// callback functions

void open_password_popup(lv_event_t *e);
static void textarea_event_handler(lv_event_t *e);
void close_password_popup(lv_event_t *e);

static void password_popup_connect_btn_cb(lv_event_t *e);
static void keyboard_delete_cb(lv_event_t *e);
static void password_textarea_cb(lv_event_t *e);


// message
void handle_net_message(std::string* strings_array, uint16_t* number_of_strings);
void display_network_con_status(std::string* status_array);
////////////////////////////////////////////////////////////////////////////////

// connect/disconnect callback class
class ServerCallbacks : public BLEServerCallbacks
{
    void onConnect(BLEServer *BLE_HMI_server){
        // what to do on connection
        BLEdeviceConnected = true;
        lv_label_set_text(BLE_connection_status_label, "Bluetooth connected");
        lv_label_set_text(NET_connection_status_label, "Network state unknown");
        //Serial.println("Device connected");
    };

    void onDisconnect(BLEServer *BLE_HMI_server){
        // what to do on disconnection
        BLEdeviceConnected = false;
        lv_label_set_text(BLE_connection_status_label, "Bluetooth disconnected");
        lv_label_set_text(NET_connection_status_label, "Network state unknown");
        //Serial.println("Device disconnected");
        no_connections_available(connection_table_backdrop);
        BLEDevice::startAdvertising(); // wait for another connection
    }
};

// class that deals with receiving string chunks from client
class recieveNetworkNamesCallback : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *receiver_characteristic)
    {
        std::string BLE_received_string = receiver_characteristic->getValue();
        /*
```

```
        if (BLE_received_string.length() > 0) {                    //
            Serial.println("*********");                           //
            Serial.print("Recieved string: ");                     //
        for (int i = 0; i < BLE_received_string.length();
i++)      //  troubleshooting block
            Serial.print(BLE_received_string[i]);                  //
        Serial.println();                                          //
        Serial.println("*********");                              //
        }
        */
        BLE_string_from_chunks(BLE_received_string, &network_names_string,
&network_string_completed);
        if (network_string_completed)
        {
            networks_string_received = true;
            network_string_completed = false;
        }
    }
};

// class that deals with receiving string chunks from client
class recieveMessageCallback : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *receiver_characteristic)
    {
        std::string BLE_received_string = receiver_characteristic->getValue();
        /*
        if (BLE_received_string.length() > 0) {                    //
            Serial.println("*********");                           //
            Serial.print("Recieved string: ");                     //
        for (int i = 0; i < BLE_received_string.length();
i++)      //  troubleshooting block
            Serial.print(BLE_received_string[i]);                  //
        Serial.println();                                          //
        Serial.println("*********");                              //
        }
        */
        BLE_string_from_chunks(BLE_received_string, &net_message_string,
&net_message_string_completed);
        if (net_message_string_completed)
        {
            net_message_string_received = true;
            net_message_string_completed = false;
        }
    }
};

////////////////////////////////////////////////////////////////////////////////
```

```
// LVGL SETUP FUNCTIONS AND DRIVERS //
void init_LVGL()
{
    Serial.begin(115200);

    // gfx setup
    pinMode(BL_PIN, OUTPUT);   // backlight initialization
    digitalWrite(BL_PIN, 1);   //
    gfx.begin();               // start the LovyanGFX
    gfx.fillScreen(TFT_BLACK); //
    //
    lv_init(); // LVGL initialize

    // display setup
    lv_disp_draw_buf_init(&draw_buf, disp_draw_buf1, NULL, screenWidth *
screenHeight / 8);
    lv_disp_drv_init(&disp_drv); // display driver
    disp_drv.hor_res = screenWidth;
    disp_drv.ver_res = screenHeight;
    disp_drv.flush_cb = disp_flush; // flush function for LVGL
    disp_drv.full_refresh = 1;
    disp_drv.draw_buf = &draw_buf;
    lv_disp_drv_register(&disp_drv);

    // touchscreen setup
    static lv_indev_drv_t indev_drv;
    lv_indev_drv_init(&indev_drv);
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    indev_drv.read_cb = touchpad_read; // touchpad reed function
    lv_indev_drv_register(&indev_drv);

    // default screen
    default_screen = lv_scr_act();

    // SANITY TEST //
    /*
    screen = lv_obj_create(NULL);
    lv_obj_set_style_bg_color(screen, lv_color_hex(0x003a57), LV_PART_MAIN);
    lv_obj_t *label = lv_label_create(screen);
    lv_label_set_text(label, "Hello world");
    lv_obj_set_style_text_color(label, lv_color_hex(0xd61a1a), LV_PART_MAIN);
    lv_obj_align(label, LV_ALIGN_CENTER, 0, 0);
    lv_scr_load(screen);
    delay(3000);
    */
    //
}

void touchpad_read(lv_indev_drv_t *indev_driver, lv_indev_data_t *data){
```

```
    // This function gives LVGL the touch cordinates
    uint16_t touchX, touchY;
    bool touched = gfx.getTouch(&touchX, &touchY);
    if (!touched)
    {
        data->state = LV_INDEV_STATE_REL;
    }
    else
    {
        // set the coordinates
        data->state = LV_INDEV_STATE_PR;
        data->point.x = touchX;
        data->point.y = touchY;
    }
}

void disp_flush(lv_disp_drv_t *disp, const lv_area_t *area, lv_color_t *color_p){
    // This function is used by LVGL to push graphics to the screen
    uint32_t w = (area->x2 - area->x1 + 1);
    uint32_t h = (area->y2 - area->y1 + 1);
    gfx.pushImageDMA(area->x1, area->y1, w, h, (lgfx::rgb565_t *)&color_p->full);
    lv_disp_flush_ready(disp);
}

/////////////////////////////////////////////////////////////////////////////////

void setup(){
    // Create a task to handle the LVGL updates
    xTaskCreatePinnedToCore(
        LVGL_handler_function, /* Task function. */
        "LVGL_handler_task",   /* name of task. */
        100000,                /* Stack size of task */
        NULL,                  /* parameter of the task */
        1,                     /* priority of the task */
        &LVGL_handler_task,    /* Task handle to keep track of created task */
        0);                    /* pin task to core 0 */
    // Setup functions //
    init_LVGL();
    init_tabs();
    init_BLE();
    //
}

void loop(){
    if (BLEdeviceConnected == true){
        if (networks_string_received == true){
            networks.clear();
            networks.shrink_to_fit();
            BLE_network_names_from_string(network_names_string, networks);
```

```
            put_network_names_in_table(networks);
            network_names_string = "";
            network_string_completed = false;
            networks_string_received = false;
        }
        if (net_message_string_received == true){
            BLE_array_from_string(&net_message_string, net_message_strings_array,
&net_message_num_strings);
            handle_net_message(net_message_strings_array,
&net_message_num_strings);
            net_message_string = "";
            net_message_string_completed = false;
            net_message_string_received = false;
        }
    }
}

void LVGL_handler_function(void *pvParameters){
    /*
     *   This task handles the LVGL timer as to
     *   free up the main loop
     */
    uint32_t timer_handler_time;
    while (true){
        // Timer handler //
        timer_handler_time = lv_timer_handler();    // don't touch this
        vTaskDelay(timer_handler_time + 30);        //
        //
    }
}

////////////////////////////////////////////////////////////////////////////////

// UI INIT FUNCTIONS //

void init_tabs(){
    main_tabview = lv_tabview_create(default_screen, LV_DIR_TOP,
MAIN_TABVIEW_HEIGHT); // create main tabview
    init_connection_tab();
    //init_macros_tab();
    //init_PLEA_settings_tab();
    //init_settings_tab();
}

void init_connection_tab(){
    /*
     *   This tab deals with connecting wifi/ethernet to PLEA
     */
```

```
    connection_tab = lv_tabview_add_tab(main_tabview, "Connection"); // create
connection_tab
    lv_obj_set_flex_flow(connection_tab, LV_FLEX_FLOW_ROW);

    connection_table_backdrop = lv_obj_create(connection_tab);
    lv_obj_t *connection_buttons_backdrop = lv_obj_create(connection_tab);
    lv_obj_set_height(connection_table_backdrop, CONNECTION_TABLE_BACKDROP_HEIGHT);
    lv_obj_set_height(connection_buttons_backdrop,
CONNECTION_BUTTONS_BACKDROP_HEIGHT);
    lv_obj_set_flex_flow(connection_table_backdrop, LV_FLEX_FLOW_COLUMN);
    lv_obj_set_flex_flow(connection_buttons_backdrop, LV_FLEX_FLOW_COLUMN);

    lv_obj_set_flex_grow(connection_table_backdrop, 1);   // set the ratio of table
    lv_obj_set_flex_grow(connection_buttons_backdrop, 1); // to buttons

    // Reduce padding and remove radius
    static lv_style_t no_padding_style;
    lv_style_init(&no_padding_style);
    lv_style_set_radius(&no_padding_style, 0);
    lv_style_set_pad_all(&no_padding_style, 0);
    lv_style_set_pad_gap(&no_padding_style, 0);

    lv_obj_add_style(connection_tab, &no_padding_style, 0);
    lv_obj_add_style(connection_table_backdrop, &no_padding_style, 0);

    // Connection table
    no_connections_available(connection_table_backdrop);

    lv_obj_set_flex_flow(connection_buttons_backdrop, LV_FLEX_FLOW_COLUMN);
    lv_obj_set_flex_align(connection_buttons_backdrop, LV_FLEX_ALIGN_START,
LV_FLEX_ALIGN_START, LV_FLEX_ALIGN_START); // how to align backdrop

    // Create a style for buttons
    static lv_style_t connect_btn_style;
    lv_style_init(&connect_btn_style);
    lv_style_set_width(&connect_btn_style, LV_PCT(100));
    lv_style_set_flex_grow(&connect_btn_style, 1);
    lv_style_set_radius(&connect_btn_style, 0);
    lv_style_set_border_color(&connect_btn_style, lv_color_hex(0x2f3436));
    lv_style_set_text_align(&connect_btn_style, LV_TEXT_ALIGN_CENTER);

    // Add connection indicator object
    lv_obj_t *con_indicator_backdrop = lv_obj_create(connection_buttons_backdrop);
    lv_obj_add_style(con_indicator_backdrop, &connect_btn_style, LV_PART_MAIN);
    lv_obj_set_flex_grow(con_indicator_backdrop, 2);

    BLE_connection_status_label = lv_label_create(con_indicator_backdrop);
    lv_label_set_text(BLE_connection_status_label, "Bluetooth disconnected");
    lv_obj_set_align(BLE_connection_status_label, LV_ALIGN_TOP_MID);
```

```
    NET_connection_status_label = lv_label_create(con_indicator_backdrop);
    lv_label_set_text(NET_connection_status_label, "Network state unknown");
    lv_obj_set_align(NET_connection_status_label, LV_ALIGN_BOTTOM_MID);

    // Add search connections button
    lv_obj_t *srch_networks_btn = lv_btn_create(connection_buttons_backdrop);
    lv_obj_t *srch_networks_btn_label = lv_label_create(srch_networks_btn);
    lv_label_set_text(srch_networks_btn_label, "Search for networks");
    lv_obj_add_style(srch_networks_btn, &connect_btn_style, 0);
    lv_obj_add_style(srch_networks_btn_label, &connect_btn_style, 0);
    lv_obj_add_event_cb(srch_networks_btn, send_simple_command_cb,
LV_EVENT_CLICKED, (void *)&PLEA_commands[0]); // Send 's'
    lv_obj_center(srch_networks_btn_label);

    // Add IP button
    lv_obj_t *IP_btn = lv_btn_create(connection_buttons_backdrop);
    lv_obj_t *IP_btn_label = lv_label_create(IP_btn);
    lv_label_set_text(IP_btn_label, "Get IP");
    lv_obj_add_style(IP_btn, &connect_btn_style, 0);
    lv_obj_add_style(IP_btn_label, &connect_btn_style, 0);
    lv_obj_add_event_cb(IP_btn, send_simple_command_cb, LV_EVENT_CLICKED, (void
*)&PLEA_commands[1]); // Send 'p'
    lv_obj_center(IP_btn_label);

    // Add disconnect network button
    lv_obj_t *disconnect_network_btn = lv_btn_create(connection_buttons_backdrop);
    lv_obj_t *disconnect_network_btn_label =
lv_label_create(disconnect_network_btn);
    lv_label_set_text(disconnect_network_btn_label, "Disconnect networks");
    lv_obj_add_style(disconnect_network_btn, &connect_btn_style, 0);
    lv_obj_add_style(disconnect_network_btn_label, &connect_btn_style, 0);
    lv_obj_add_event_cb(disconnect_network_btn, send_simple_command_cb,
LV_EVENT_CLICKED, (void *)&PLEA_commands[2]); // Send 'd'
    lv_obj_center(disconnect_network_btn_label);
}

void init_macros_tab()
{
    /*
     *   This tab deals with quick macro buttons
     */
    lv_obj_t *macros_tab = lv_tabview_add_tab(main_tabview, "Macros");
    lv_obj_t *btn = lv_btn_create(macros_tab);
}

void init_settings_tab()
{
    /*
```

```
     *   This tab deals with settings of the HMI
     */
    lv_obj_t *settings_tab = lv_tabview_add_tab(main_tabview, "Settings");
}

void init_PLEA_settings_tab()
{
    /*
     *   This tab deals with PLEA settings
     */
    lv_obj_t *PLEA_settings_tab = lv_tabview_add_tab(main_tabview, "PLEA
settings");
}

// UI FUNCTIONS //

void put_network_names_in_table(const std::vector<Network> &networks){
    byte number_of_networks = networks.size();

    lv_obj_clean(connection_table_backdrop);
    connection_table = lv_table_create(connection_table_backdrop);
    lv_obj_remove_event_cb(connection_table, open_password_popup);  // If we don't
remove previous callback, the callback will happen twice
    lv_obj_add_event_cb(connection_table, open_password_popup,
LV_EVENT_VALUE_CHANGED, NULL);
    lv_obj_set_size(connection_table, LV_PCT(100), LV_PCT(100));    // Make it
takeup the whole backdrop
    lv_table_set_col_cnt(connection_table, 3);                      //
    lv_table_set_col_width(connection_table, 0, 80);                //
    lv_table_set_col_width(connection_table, 2, 80);                //
    lv_table_set_col_width(connection_table, 1, 220);               //


    for (byte i = 0; i < number_of_networks; i++)
    {
        lv_table_set_cell_value(connection_table, i, 0, networks[i].type.c_str());
        lv_table_set_cell_value(connection_table, i, 1, networks[i].name.c_str());
        lv_table_set_cell_value(connection_table, i, 2,
std::to_string(networks[i].table_entry_nmb).c_str());
        /*
        Serial.print(networks[i].name.c_str());         //
        Serial.print(" entry: ");                       // Troubleshooting block
        Serial.println(networks[i].table_entry_nmb);    //
        */
    }
}

void no_connections_available(lv_obj_t* backdrop){
    lv_obj_clean(backdrop);
```

```
    lv_obj_t* placeholder_label = lv_label_create(backdrop);
    lv_label_set_text(placeholder_label, "Networks unavailable.");
    lv_obj_center(placeholder_label);
    //lv_obj_set_align(placeholder_label, LV_ALIGN_CENTER);
}

// NETWORK FUNCTIONS //

void handle_net_message(std::string* strings_array, uint16_t* number_of_strings) {
    /*
    *   Function takes in an array of strings:
    *       if (the first string in the array is a single char) -> it is a special
command
    *       elese -> puts the elements of the array into a popup
    */
    if (strings_array[0].length() == 1) {   // If the first string from the array
is a single char then it is a command
        char command = strings_array[0][0]; // Gives the char in the first string
from the array
        Serial.print("Command: ");
        Serial.println(command);
        switch (command) {
            case NETWORK_CON_MESSAGE:
                Serial.println("NET CONNECTED");
                display_network_con_status(strings_array);
                break;
            case NETWORK_DISCON_MESSAGE:
                Serial.println("NET DISCONNECTED");
                lv_label_set_text(NET_connection_status_label, "Networks
disconnected");
                connected_to_network = false;
                break;
            case NETWORK_ERROR_MESSAGE:
                Serial.println("NET ERROR");
                break;
            default:
                Serial.println("UNKNOWN MESSAGE");
                break;
        }
    } else {    // Create message popup
        Serial.println("MESSAGE POPUP");
        std::string message = "";
        std::string title = strings_array[0];

        for(uint16_t i=1; i<(*number_of_strings); i++){
            message += strings_array[i] + '\n';
        }

        // popup create
```

```
        // array[0] = title
        // array[1...n] = messages (each message goes in a seperate row)
        lv_obj_t* popup = lv_msgbox_create(NULL, title.c_str(), message.c_str(),
NULL, true);
        lv_obj_set_size(popup, 300, 200);
        lv_obj_center(popup);
    }
    // Empty the array
    for (uint16_t i = 0; i < *number_of_strings; i++) {
        strings_array[i].clear();
    }
    *number_of_strings = 0;
}

void display_network_con_status(std::string* status_array){
    /*
    *   Status array standard:
    *   [0] - NETWORK_CON_MESSAGE
    *   [1] - Network status message
    */
    // Example to send to NETWORK_MESSAGE_CH_UUID:
    // <<C>><<E: Network1, W: Network2>>#

    std::string con_message = status_array[1];
    lv_label_set_text(NET_connection_status_label, con_message.c_str());
    connected_to_network = true;
}

// STRING FUNCTIONS //

void BLE_string_from_chunks(std::string chunk, std::string *storage_string, bool
*completed_message_indicator){
    /*
     *   Takes the incoming chunks of the string coming in
     *   and appends them to a string. Detects the end of a
     *   message with the '#' and changes an indicator booll
     *   to true.
     */
    *storage_string += chunk; // Append the chunk that was sent over BLE to
                              // a string that will containt the whole message
    if (chunk.back() == '#' || chunk == "")
    { // If the chunk ends with '#' or it is empty
        *completed_message_indicator = true;
        /*
        Serial.println("*********");                    //
        Serial.println(storage_string->c_str());        // troubleshooting block
        Serial.println();                               //
        Serial.println("Message indicator:");           //
        Serial.println(*completed_message_indicator);   //
```

```
        Serial.println("*********");                    //
        */
    }
}

void BLE_network_names_from_string(std::string networks_string,
std::vector<Network> &networks){
    /*
     *   Chops the networks_string into individual networks.
     *
     *   Function searches for string chunks that start with W:
     *   or E: and the name of the network that is between << and >>
     *   '#' marks the end of the string.
     *
     *   Legend:
     *   W:  WiFi network
     *   E:  Ethernet network
     *   <<  Start of a network's name
     *   >>  End of a network's name
     *
     *   Example (one WiFi network with a name net123):
     *   W:<<net123>>#
     */
    networks.clear(); // Empty the vector before writing new networks

    std::regex re(R"((W:|E:)<<([^>]+)>>)"); // Regular expression to extract
network names and types
    std::smatch match;
    std::string::const_iterator searchStart(networks_string.cbegin());

    // Extract network names and their types
    byte i = 0;
    while (std::regex_search(searchStart, networks_string.cend(), match, re))
    {
        std::string type = match[1] == "W:" ? "WiFi" : "Eth";
        std::string name = match[2];
        networks.push_back({name, type, "", i}); // Initially, the password is
empty
        searchStart = match.suffix().first;
        i++;
    }

    /*
    for (const auto& network : networks) {          //
        Serial.print("Network Name: ");              //
        Serial.print(network.name.c_str());         //
        Serial.print(", Type: ");                   //  Troubleshooting block
        Serial.print(network.type.c_str());         //
        Serial.print(", Password: ");               //
```

```
        Serial.println(network.password.c_str());   //
    }
    */
}

void BLE_array_from_string(std::string* whole_string, std::string* strings_array,
uint16_t* number_of_strings){
    /*
     *   Chops a string into individual strings and puts
     *   them in an array.
     *
     *   Function searches for string chunks that start with << and
     *   end with >>. '#' marks the end of the whole string. Puts
     *   all the chunks into a string array.
     */

    // **BUG** --> If a string inbetween any <<string>> is to long (>50 char) it
crashes

    *number_of_strings = 0;

    std::regex re(R"(<<([^>]+)>>)"); // Regular expression to extract IPs
    std::smatch match;
    std::string::const_iterator searchStart((*whole_string).cbegin());

    // Extract individual data contain in <<data>> instances
    uint16_t i = 0;
    while (std::regex_search(searchStart, (*whole_string).cend(), match, re))
    {
        strings_array[i] = match[1].str();
        searchStart = match.suffix().first;
        i++;
    }
    *number_of_strings = i;

    /*
    Serial.println("Received IPs:");                //
    for(i=0; i<(*number_of_strings); i++){          //
        Serial.println(strings_array[i].c_str());   // Troubleshooting block
    }                                               //
    Serial.println("*********");                   //
    */
}

////////////////////////////////////////////////////////////////////////////////

// BLE FUNCTIONS //

void init_BLE(){
```

```
    // Create the BLE Device
    BLEDevice::init("PLEA HMI");

    // Create the BLE Server
    BLE_HMI_server = BLEDevice::createServer();
    BLE_HMI_server->setCallbacks(new ServerCallbacks()); // Set callback on connect
and disconnect

    BLE_advertising = BLEDevice::getAdvertising(); // Create advertising

    // Initialize services
    init_BLE_network_service();

    // Start advertising
    BLE_advertising->setScanResponse(false);
    BLE_advertising->setMinPreferred(0x0); // Set value to 0x00 to not advertise
this parameter
    BLEDevice::startAdvertising();
    Serial.println("Waiting for a client connection...");
    //
}

void init_BLE_network_service(){
    /*
     *   This function creates the BLE_network_service
     *   and it starts it.
     */
    BLE_network_service = BLE_HMI_server->createService(NETWORK_SERVICE_UUID); //
Create BLE network service

    // Create BLE_network_names_ch characteristic
    // It is intended for receiveing network names from Raspberry PI
    BLE_network_names_ch = BLE_network_service->createCharacteristic(
        NETWORK_NAMES_CH_UUID,
        BLECharacteristic::PROPERTY_WRITE);
    BLE_network_names_ch->setCallbacks(new recieveNetworkNamesCallback()); // Add
callback on recieve from client

    // Create BLE_network_connect_ch characteristic
    // It sends network type, name and password of the network we want to connect
to
    BLE_network_connect_ch = BLE_network_service->createCharacteristic(
        NETWORK_CONNECT_CH_UUID,
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLE_network_connect_ch->addDescriptor(new BLE2902()); // Add a BLE descriptor
for notifications
```

```
    // Create BLE_network_message_ch characteristic
    // It is intended for receiveing IPs from Raspberry PI
    BLE_network_message_ch = BLE_network_service->createCharacteristic(
        NETWORK_MESSAGE_CH_UUID,
        BLECharacteristic::PROPERTY_WRITE);
    BLE_network_message_ch->setCallbacks(new recieveMessageCallback()); // Add
callback on recieve from client


    // Create BLE_network_commands_ch characteristic
    // It sends simple commands via buttons
    BLE_network_commands_ch = BLE_network_service->createCharacteristic(
        NETWORK_COMMANDS_CH_UUID,
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLE_network_commands_ch->addDescriptor(new BLE2902()); // Add a BLE descriptor
for notifications


    BLE_advertising->addServiceUUID(NETWORK_SERVICE_UUID); // Add
BLE_network_service to advertising
    BLE_network_service->start(); // Start the service
}

void connect_to_network(Network* network){
    // Construct string that will be sent to Raspberry PI
    std::string connect_info_string = "";
    if(network->type == "WiFi"){
        connect_info_string = "<<W:>>";
        connect_info_string = connect_info_string + "<<" + network->name + ">>";
        connect_info_string = connect_info_string + "<<" + network->password + ">>"
+ '#';
    }else if(network->type == "Eth"){   // If it's an ethernet network we don't
need a password
        connect_info_string = "<<E:>>";
        connect_info_string = connect_info_string + "<<" + network->name + ">>" +
'#';
    }

    int chunk_num;
    std::string string_chunks_array[20];
    delay(20);
    chunk_num = BLE_chunks_array_from_string(string_chunks_array,
connect_info_string, BLE_CHUNK_SIZE);
    BLE_send_connect_network_info(string_chunks_array, chunk_num);
}

int BLE_chunks_array_from_string(std::string *chunks_array, std::string&
whole_string, uint16_t chunk_size){
```

```
    /*
    *   Function takes in:
    *   chunk_array[]       -where it will write chunks to
    *   whole_string        -string to chop up into chunks
    *   chunk_size          -size of chunks
    *
    *   Returns number of entrys put into array
    */
    uint16_t string_length = whole_string.length();             //
    uint16_t num_chunks = string_length / chunk_size;           // Calculate number
of chunks
    uint16_t num_incomplete_chunk = string_length % chunk_size; // If there is a
remainder

    Serial.println("String array chunks: ");

    uint16_t array_entry = 0;
    for (uint16_t i = 0; i < string_length; i += chunk_size) {
        chunks_array[array_entry] = whole_string.substr(i, chunk_size);
        Serial.println(chunks_array[array_entry].c_str());
        array_entry++;
    }

    if(num_incomplete_chunk != 0){
        chunks_array[array_entry] = whole_string.substr(string_length -
num_incomplete_chunk, num_incomplete_chunk);
        Serial.println(chunks_array[array_entry].c_str());
    }else{
        array_entry--;
    }

    return array_entry;
}

void BLE_send_connect_network_info(std::string* send_chunks_array, int& chunk_num){
    Serial.println("Sent to PLEA: ");
    for(int i=0; i<chunk_num; i++){
        BLE_network_connect_ch->setValue(send_chunks_array[i].c_str());
        BLE_network_connect_ch->notify();
        //Serial.println(send_chunks_array[i].c_str());   // Troubleshooting line
        delay(300);
    }
}

////////////////////////////////////////////////////////////////////////////////

// CALLBACK FUNCTIONS //

void send_simple_command_cb(lv_event_t *e){
```

```
    char command = *(char *)lv_event_get_user_data(e); // Get the command from
user_data
    std::string command_str(1, command);               //
    BLE_network_commands_ch->setValue(command_str);    //
    BLE_network_commands_ch->notify();                 //
}

// EVENT CALLBACKS //

void open_password_popup(lv_event_t *e){
    // Get the clicked network
    lv_obj_t *table = lv_event_get_target(e);
    uint16_t row, col;
    lv_table_get_selected_cell(table, &row, &col);
    const char* network_name = lv_table_get_cell_value(table, row, 1);
    const char* network_type = lv_table_get_cell_value(table, row, 0);
    const char* msg_prt1 = "Connect to: ";
    char pupup_message[50];
    snprintf(pupup_message, sizeof(pupup_message), "%s%s", msg_prt1, network_name);

    selected_network.name = network_name;
    selected_network.type = network_type;

    // Create password popup
    lv_obj_t *password_popup = lv_msgbox_create(NULL, "Enter password",
pupup_message, NULL, false);
    lv_obj_set_size(password_popup, 300, 200);
    lv_obj_center(password_popup);

    // Create and apply password popup style
    lv_style_init(&password_popup_style);
    lv_obj_add_style(password_popup, &password_popup_style, LV_PART_MAIN);

    if(strcmp(network_type, "WiFi") == 0){
        // Create a text area for password input for WiFi
        lv_obj_t *password_textarea = lv_textarea_create(password_popup);
        lv_textarea_set_password_mode(password_textarea, true);
        lv_textarea_set_one_line(password_textarea, true);
        lv_textarea_set_placeholder_text(password_textarea, "Enter password");
        lv_obj_set_width(password_textarea, LV_PCT(80));
        lv_obj_center(password_textarea);
        lv_obj_add_event_cb(password_textarea, password_textarea_cb,
LV_EVENT_FOCUSED, password_popup);
    }

    // Create connect button
    lv_obj_t *connect_btn = lv_btn_create(password_popup);
    lv_obj_t *connect_btn_label = lv_label_create(connect_btn);
    lv_obj_set_size(connect_btn, 80, 30);
```

```
    lv_obj_align(connect_btn, LV_ALIGN_BOTTOM_LEFT, -30, -10);
    lv_label_set_text(connect_btn_label, "Connect");
    lv_obj_add_event_cb(connect_btn, password_popup_connect_btn_cb,
LV_EVENT_CLICKED, &selected_network);
    lv_obj_center(connect_btn_label);

    // Create cancel button
    lv_obj_t *cancel_btn = lv_btn_create(password_popup);
    lv_obj_t *cancel_btn_label = lv_label_create(cancel_btn);
    lv_obj_set_size(cancel_btn, 80, 30);
    lv_obj_align(cancel_btn, LV_ALIGN_BOTTOM_RIGHT, 30, -10);
    lv_label_set_text(cancel_btn_label, "Cancel");
    lv_obj_add_event_cb(cancel_btn, close_password_popup, LV_EVENT_CLICKED,
password_popup);
    lv_obj_center(cancel_btn_label);
}

static void password_textarea_cb(lv_event_t *e){
    lv_event_code_t code = lv_event_get_code(e);
    lv_obj_t* password_textarea = lv_event_get_target(e);
    lv_obj_t* password_popup = (lv_obj_t*)lv_event_get_user_data(e);  // Get the
parent popup from event data

    if (code == LV_EVENT_FOCUSED)
    {
        lv_style_set_translate_y(&password_popup_style, -25);
        lv_obj_refresh_style(password_popup, LV_PART_MAIN, LV_STYLE_PROP_ANY);

        // Create the keyboard dynamically
        lv_obj_t *keyboard = lv_keyboard_create(lv_layer_top());
        lv_keyboard_set_textarea(keyboard, password_textarea);
        lv_obj_add_event_cb(keyboard, keyboard_delete_cb, LV_EVENT_READY,
password_textarea);
    }
}

static void keyboard_delete_cb(lv_event_t *e){
    lv_obj_t *keyboard = lv_event_get_target(e);
    lv_obj_t *text_area = (lv_obj_t *)lv_event_get_user_data(e);

    // Clear focus and reset style translation before deleting the keyboard
    lv_obj_clear_state(text_area, LV_STATE_FOCUSED);
    selected_network.password = lv_textarea_get_text(text_area);

    // Reset the popup style before deleting the keyboard
    lv_style_set_translate_y(&password_popup_style, 0);
    lv_obj_refresh_style(lv_obj_get_parent(text_area), LV_PART_MAIN,
LV_STYLE_PROP_ANY);
```

```
    lv_obj_del(keyboard);  // Delete the keyboard
}

void close_password_popup(lv_event_t *e){
    lv_obj_t *cancel_btn = lv_event_get_target(e);
    lv_msgbox_close(lv_obj_get_parent(cancel_btn));
    //
    selected_network.name = "";     //
    selected_network.password = ""; // Clear the selected network
    selected_network.type = "";     //
}

static void password_popup_connect_btn_cb(lv_event_t *e){
    /*
    Serial.println("CONNECT_BUTTON_PRESSED");                  //
    Serial.println("Connect to network: ");                    //
    Serial.println(network_to_connect_to->name.c_str());       // Troubleshooting
block
    Serial.println(network_to_connect_to->password.c_str());   //
    Serial.println(network_to_connect_to->type.c_str());       //
    */
    connect_to_network((Network*) lv_event_get_user_data(e));

    selected_network.name = "";     //
    selected_network.password = ""; // Clear the selected network
    selected_network.type = "";     //

    lv_obj_t *connect_btn = lv_event_get_target(e);
    lv_msgbox_close(lv_obj_get_parent(connect_btn));
}
```

I.I. BLE_config.h

```
#define BLE_SERVER_NAME      "PLEA HMI"     // name of server that client will see

#define BLE_CHUNK_SIZE              20      // how many char can be sent over BLE
at once


#define NETWORK_SERVICE_UUID                "9fd1e9cf-97f7-4b0b-9c90-caac19dba4f8"
#define NETWORK_NAMES_CH_UUID               "0fd59f95-2c93-4bf8-b5f0-343e838fa302"
#define NETWORK_CONNECT_CH_UUID             "15eb77c1-6581-4144-b510-37d09f4294ed"
#define NETWORK_MESSAGE_CH_UUID             "773f99ff-4d87-4fe4-81ff-190ee1a6c916"
#define NETWORK_COMMANDS_CH_UUID            "68b82c8a-28ce-43c3-a6d2-509c71569c44"
```

```
#define SEARCH_NETWORKS_COMMAND     's'     // command for searching networks   's'
#define REQUEST_IP_COMMAND          'p'     // command for getting IP           'p'
#define DISCONNECT_NETWORK_COMMAND  'd'     // command for disconnecting        'd'



#define NETWORK_CON_MESSAGE         'C'     // network connected status         'C'
#define NETWORK_DISCON_MESSAGE      'D'     // network disconnected             'D'
#define NETWORK_ERROR_MESSAGE       'E'     // message for errors               'E'
```

I.II. gfx_config.h

```
#define LGFX_USE_V1
#include <LovyanGFX.hpp>
#include <lgfx/v1/platforms/esp32s3/Panel_RGB.hpp>
#include <lgfx/v1/platforms/esp32s3/Bus_RGB.hpp>
#include <driver/i2c.h>

// #define CrowPanel_70    1       // screen model
#define screenWidth   800
#define screenHeight  480

class LGFX : public lgfx::LGFX_Device
{
public:
    lgfx::Bus_RGB _bus_instance;
    lgfx::Panel_RGB _panel_instance;
    lgfx::Light_PWM _light_instance;
    lgfx::Touch_GT911 _touch_instance;
    LGFX(void)
    {
        {
            auto cfg = _panel_instance.config();
            cfg.memory_width = screenWidth;
            cfg.memory_height = screenHeight;
            cfg.panel_width = screenWidth;
            cfg.panel_height = screenHeight;
            cfg.offset_x = 0;
            cfg.offset_y = 0;
            _panel_instance.config(cfg);
        }

        {
            auto cfg = _bus_instance.config();
            cfg.panel = &_panel_instance;

            cfg.pin_d0 = GPIO_NUM_15;  // B0
```

```
        cfg.pin_d1 = GPIO_NUM_7;  // B1
        cfg.pin_d2 = GPIO_NUM_6; // B2
        cfg.pin_d3 = GPIO_NUM_5;  // B3
        cfg.pin_d4 = GPIO_NUM_4;  // B4

        cfg.pin_d5 = GPIO_NUM_9;  // G0
        cfg.pin_d6 = GPIO_NUM_46;  // G1
        cfg.pin_d7 = GPIO_NUM_3;  // G2
        cfg.pin_d8 = GPIO_NUM_8; // G3
        cfg.pin_d9 = GPIO_NUM_16; // G4
        cfg.pin_d10 = GPIO_NUM_1; // G5

        cfg.pin_d11 = GPIO_NUM_14; // R0
        cfg.pin_d12 = GPIO_NUM_21; // R1
        cfg.pin_d13 = GPIO_NUM_47; // R2
        cfg.pin_d14 = GPIO_NUM_48; // R3
        cfg.pin_d15 = GPIO_NUM_45; // R4

        cfg.pin_henable = GPIO_NUM_41;
        cfg.pin_vsync = GPIO_NUM_40;
        cfg.pin_hsync = GPIO_NUM_39;
        cfg.pin_pclk = GPIO_NUM_0;
        cfg.freq_write = 16000000;

        cfg.hsync_polarity    = 0;
        cfg.hsync_front_porch = 40;
        cfg.hsync_pulse_width = 48;
        cfg.hsync_back_porch  = 40;

        cfg.vsync_polarity    = 0;
        cfg.vsync_front_porch = 1;
        cfg.vsync_pulse_width = 31;
        cfg.vsync_back_porch  = 13;

        cfg.pclk_active_neg = 1;
        cfg.de_idle_high = 0;
        cfg.pclk_idle_high = 0;

        _bus_instance.config(cfg);
        _panel_instance.setBus(&_bus_instance);
    }

    {
        auto cfg = _light_instance.config();
        cfg.pin_bl = GPIO_NUM_2;
        _light_instance.config(cfg);
        _panel_instance.light(&_light_instance);
    }
```

```
        {
            auto cfg = _touch_instance.config();
            cfg.x_min      = 0;
            cfg.x_max      = 799;
            cfg.y_min      = 0;
            cfg.y_max      = 479;
            cfg.pin_int    = -1;
            cfg.pin_rst    = -1;
            cfg.bus_shared = true;
            cfg.offset_rotation = 0;
            cfg.i2c_port   = I2C_NUM_1;
            cfg.pin_sda    = GPIO_NUM_19;
            cfg.pin_scl    = GPIO_NUM_20;
            cfg.freq       = 400000;
            cfg.i2c_addr   = 0x14;
            _touch_instance.config(cfg);
            _panel_instance.setTouch(&_touch_instance);
        }
        setPanel(&_panel_instance);
    }
};


LGFX gfx;       // Making an instance of LGFX class that we will use in all src
files
```

I.III. UI_parameters.h

```
/*
This file contains the parameters of the UI, by changing the parameters here,...
*/
#define SCREEN_HEIGHT                       480
#define SCREEN_WIDTH                        800

//======================//
//  MENU BAR            //


// SIZES //
#define MAIN_TABVIEW_HEIGHT                 75
#define MENU_BAR_BTN_WIDTH                  100
#define CONNECTION_TABLE_BACKDROP_HEIGHT    (SCREEN_HEIGHT - MAIN_TABVIEW_HEIGHT)
#define CONNECTION_BUTTONS_BACKDROP_HEIGHT  (SCREEN_HEIGHT - MAIN_TABVIEW_HEIGHT)

// COLORS //
#define MENU_BAR_BG_COLOR               0x2f3436
#define MENU_BAR_BTN_BG_COLOR           0x222626
//======================//
```

I.IV. hardware.h

```
#define CrowPanel_70    // name of the hardware module
#define BL_PIN  2       // backlight pin
```

II. PLEA_BLE_network.py

```
# Repository link: https://github.com/xXDemionXx/PLEA_BLE_network

from bluepy import btle
from bluepy.btle import Peripheral, UUID, DefaultDelegate
import subprocess
import time
import threading

# Define the UUIDs of the characteristics you want to subscribe to
DEVICE_MAC_ADDRESS = "84:FC:E6:6B:D6:C1"
NETWORK_SERVICE_UUID = "9fd1e9cf-97f7-4b0b-9c90-caac19dba4f8"
NETWORK_NAMES_CH_UUID = "0fd59f95-2c93-4bf8-b5f0-343e838fa302"
NETWORK_CONNECT_CH_UUID = "15eb77c1-6581-4144-b510-37d09f4294ed"
NETWORK_MESSAGE_CH_UUID = "773f99ff-4d87-4fe4-81ff-190ee1a6c916"
NETWORK_COMMANDS_CH_UUID = "68b82c8a-28ce-43c3-a6d2-509c71569c44"

# Class that handles network notifications
class NetworkNotificationDelegate(btle.DefaultDelegate):
    def __init__(self, network_connect_ch, network_commands_ch):
        super().__init__()
        self.network_connect_ch = network_connect_ch
        self.network_commands_ch = network_commands_ch
        self.connect_network_string = ""
        self.connect_network_string_finished = False

    def handleNotification(self, cHandle, data):
        if cHandle == self.network_connect_ch.getHandle():
            self.connect_network_string += data.decode('utf-8')
            if self.connect_network_string[-1] == '#':
                self.connect_network_string_finished = True
        elif cHandle == self.network_commands_ch.getHandle():
            handle_network_commands(data.decode('utf-8'))

def notification_loop(peripheral, stop_event):
    previous_connected_networks = ""
    while not stop_event.is_set():
        try:
            if peripheral.waitForNotifications(1.0):  # Constantly listen for
notifications
                continue
            # Constantly check if any new networks have connected or disconnected
            currently_connected_networks = subprocess.run(['nmcli', '-t', '-f',
'NAME', 'connection', 'show', '--active'], capture_output=True, text=True).stdout
            if previous_connected_networks != currently_connected_networks:     #
Check if any network connection has changed
```

```
                BLE_send_networks_connection_status_message(currently_connected_ne
tworks)
                previous_connected_networks = currently_connected_networks
        except btle.BTLEDisconnectError:
            print("Notification loop detected disconnection.")
            stop_event.set()  # Signal the main thread that disconnection has
occurred

# BLE connect functions
def BLE_connect_to_device():
    try:
        print("Attempting to connect to ESP32...")
        peripheral = btle.Peripheral(DEVICE_MAC_ADDRESS)
        print("Connected to ESP32")
        return peripheral
    except Exception as e:
        print(f"Failed to connect: {e}")
        return None

def network_service(peripheral):
    # Connect to the service
    network_service = peripheral.getServiceByUUID(UUID(NETWORK_SERVICE_UUID))

    # Send characteristics #
    network_names_ch = network_service.getCharacteristics(NETWORK_NAMES_CH_UUID)[0]
    network_message_ch =
network_service.getCharacteristics(NETWORK_MESSAGE_CH_UUID)[0]
    ###

    # Receive characteristics #
    network_connect_ch =
network_service.getCharacteristics(NETWORK_CONNECT_CH_UUID)[0]
    network_commands_ch =
network_service.getCharacteristics(NETWORK_COMMANDS_CH_UUID)[0]

    delegate = NetworkNotificationDelegate(network_connect_ch, network_commands_ch)
    peripheral.setDelegate(delegate)

    network_connect_ch_handle = network_connect_ch.getHandle() + 1
    peripheral.writeCharacteristic(network_connect_ch_handle, b'\x01\x00',
withResponse=True)  # Enable notifications

    network_commands_ch_handle = network_commands_ch.getHandle() + 1
    peripheral.writeCharacteristic(network_commands_ch_handle, b'\x01\x00',
withResponse=True)  # Enable notifications

    return network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate
```

```
def BLE_send_networks_string(networks_string):
    string_length = len(networks_string)
    chunks_array = []

    # Size of chunks we can send over BLE is 20 bytes
    for i in range(string_length // 20):
        chunks_array.append(networks_string[i * 20 : (i + 1) * 20])

    if string_length % 20 != 0:  # If there is a leftover smaller than 20 chars
        chunks_array.append(networks_string[-(string_length % 20):])

    for chunk in chunks_array:  # Send the chunks over BLE
        network_names_ch.write(bytes(str(chunk), 'utf-8'))
        time.sleep(0.003)
###

# Network names #
def get_networks_string():
    networks = ""

    # List available Wi-Fi networks
    wifi_result = subprocess.run(['nmcli', '-t', '-f', 'SSID', 'device', 'wifi',
'list'], capture_output=True, text=True)
    wifi_networks = wifi_result.stdout.split('\n')

    for ssid in wifi_networks:
        if ssid.strip():  # Skip empty SSIDs
            networks += f"W:<<{ssid.strip()}>>"

    # Check if eth0 is physically connected
    result = subprocess.run(['sudo', 'ethtool', 'eth0'], capture_output=True,
text=True)
    for line in result.stdout.split('\n'):
        if 'Link detected:' in line:
            # Check if the link is detected as yes
            if 'yes' in line:
                networks += f"E:<<eth0>>"
    networks += '#'  # Tells the receiver the string is done
    return networks
###

# IPv4 #
def get_ipv4_addresses():
    try:
        # Run the ip command to get IP address information
        result = subprocess.run(['ip', 'addr'], capture_output=True, text=True)

        interfaces = {}
        interface_name = None
```

```
        # Parse the output to extract IPv4 addresses
        for line in result.stdout.split('\n'):
            if line.startswith(' '):
                # This line contains IP address information
                if 'inet ' in line:  # Only consider 'inet' for IPv4
                    ip_address = line.strip().split()[1]
                    if interface_name:
                        interfaces[interface_name].append(ip_address)
            else:
                # This line contains the interface name
                if line:
                    parts = line.split(': ')
                    if len(parts) > 1:
                        interface_name = parts[1].split('@')[0]
                        if interface_name not in interfaces:
                            interfaces[interface_name] = []

        IPs_string = "<<IP info>>"
        for interface, addresses in interfaces.items():
            IPs_string += f"<<{interface}: {', '.join(addresses)}>>"
        IPs_string += '#'
        return IPs_string

    except Exception as e:
        print(f"Error retrieving IP addresses: {e}")
        return ""
###

# Handle network commands #
def handle_network_commands(network_command):
    match network_command:
        case 's':  # Search for networks
            print("Search for networks")
            networks_string = get_networks_string()
            networks_array = BLE_chop_string_to_chunks(networks_string, 20)
            BLE_send_array(networks_array, network_names_ch)
            return
        case 'p':  # Get IP
            print("Get IP")
            IPv4_string = get_ipv4_addresses()
            IPv4_array = BLE_chop_string_to_chunks(IPv4_string, 20)
            BLE_send_array(IPv4_array, network_message_ch)
            return
        case 'd':  # Disconnect from networks
            print("Disconnect from all networks")
            disconnect_all_networks()
            return
        case _:
```

```
            print("Unknown command")
            return
###

# Network #
def connect_to_network(network_info):
    # Incoming string looks like this:
    # If it's WiFi: <<W:>>><<NetworkName>>><<Password>>
    # If it's Eth:  <<E:>>><<NetworkName>>
    #
    # Function takes in a string, chops it into type,
    # name and password and connects to it
    #
    network_info = network_info[:-3]  # Removes >>#
    network_info = network_info[2:]  # Removes <<
    network_info_list = network_info.split('>>><')  # Makes it a list

    if len(network_info_list) < 2:  # Validate the list length
        print("Error: network_info string is not in the expected format.")
        return

    network_type = network_info_list[0]
    network_name = network_info_list[1]

    if network_type == "W:":
        network_password = network_info_list[2]
        # Connect to WiFi
        result = subprocess.run(['nmcli', 'device', 'wifi', 'connect',
network_name, 'password', network_password], capture_output=True, text=True)
        print(result)
        if result.returncode == 0:
            print(f"Successfully connected to Wi-Fi network {network_name}")
        else:
            print(f"Failed to connect to Wi-Fi network {network_name}")
            print(result.stderr)
    elif network_type == "E:":
        # Connect to Ethernet
        result = subprocess.run(['nmcli', 'device', 'connect', network_name],
capture_output=True, text=True)
        if result.returncode == 0:
            print(f"Successfully connected to Ethernet network {network_name}")
        else:
            print(f"Failed to connect to Ethernet network {network_name}")
            print(result.stderr)
    else:
        print("Error: Unknown network type.")

def get_active_networks():
```

```
    result = subprocess.run(['nmcli', '-t', '-f', 'UUID,DEVICE', 'connection',
'show', '--active'],
                            capture_output=True, text=True)
    if result.returncode != 0:
        print(f"Error getting active connections: {result.stderr}")
        return []

    active_connections = []
    for line in result.stdout.strip().split('\n'):
        if line:
            uuid, device = line.split(':')
            active_connections.append((uuid, device))

    return active_connections

def disconnect_all_networks():
    active_connections = get_active_networks()
    for uuid, device in active_connections:
        if device != "lo":
           result = subprocess.run(['nmcli', 'device', 'down', device],
capture_output=True, text=True)
           if result.returncode == 0:
              print(f"Disconnected {device} with UUID {uuid}")
           else:
              print(f"Failed to disconnect {device} with UUID {uuid}:
{result.stderr}")

def BLE_send_networks_connection_status_message(connected_networks):
    if connected_networks[-3:] == "lo\n":   # Removes loopback network if it's in
the string
        connected_networks = connected_networks[:-3]
    connected_networks = connected_networks.rstrip("\n")
    connected_networks = connected_networks.split("\n")     # Gets a list of
connected networks
    if connected_networks[0] == "":     # If we aren't connected to any networks
        #print("Networks disconnected")
        BLE_connected_networks_message  = "<<D>>#"      # Return networks
disconnected message
    else:
        BLE_connected_networks_message = "<<C>><<Connected to: "
        BLE_connected_networks_message += ', '.join(connected_networks)
        BLE_connected_networks_message += ">>#"
    BLE_connected_networks_message_array =
BLE_chop_string_to_chunks(BLE_connected_networks_message, 20)
    #print(BLE_connected_networks_message_array)
    BLE_send_array(BLE_connected_networks_message_array, network_message_ch)    #
Send the connection message
###
```

```
# BLE #
def BLE_chop_string_to_chunks(string, chunk_size):
    #
    # Takes in a string and chops it into an arraz
    # of chunks, size of chunk_size.
    #
    chunks_array = []
    string_length = len(string)

    # Size of chunks we can send over BLE is 20 bytes
    for i in range(string_length // chunk_size):
        chunks_array.append(string[i * chunk_size : (i + 1) * chunk_size])

    if string_length % chunk_size != 0: # If there is a leftover smaller than 20
chars
        chunks_array.append(string[-(string_length % chunk_size):])

    return chunks_array

def BLE_send_array(array, characteristic):
    for entry in array: # Send the chunks over BLE
        characteristic.write(bytes(str(entry), 'utf-8'))
        time.sleep(0.003)

def BLE_main():
    # Variables #
    peripheral = None
    previous_connected_networks = ""

    # Declare wanted characteristics
    global network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate
    notification_thread = None  # Thread for handling incoming notifications
    global stop_event
    global currently_connected_networks
    ###

    stop_event = threading.Event()

    while True:
        if peripheral is None:  # We are not connected via BLE
            peripheral = BLE_connect_to_device()  # Try to connect
            if peripheral is not None:  # If we succeeded
                # Connect to wanted characteristics #
                network_names_ch, network_connect_ch, network_message_ch,
network_commands_ch, delegate = network_service(peripheral)
                ###
                stop_event.clear()
```

```
                notification_thread = threading.Thread(target=notification_loop,
args=(peripheral, stop_event))
                notification_thread.daemon = True
                notification_thread.start()
            else:  # If we didn't succeed, retry
                print("Retrying in 5 seconds...")
                time.sleep(5)

        else:  # We are connected via BLE
            if stop_event.is_set():  # If the notification thread isn't running
                print("Connection lost, attempting to reconnect...")
                if notification_thread:
                    notification_thread.join()
                try:
                    peripheral.disconnect()  # Ensure disconnection
                except Exception as e:
                    print(f"Error during disconnection: {e}")
                peripheral = None
                time.sleep(5)
            else:
                try:  # Main loop handling
                    if delegate.connect_network_string_finished:    # Check if we
need to connect to a network
                        connect_to_network(delegate.connect_network_string)

                        # After connecting reset flag and string
                        delegate.connect_network_string = ""
                        delegate.connect_network_string_finished = False

                except btle.BTLEDisconnectError:
                    print("Detected disconnection during main loop tasks.")
                    stop_event.set()
###

if __name__ == "__main__":
    BLE_main()
```