Univerza v Ljubljani

Fakulteta za elektrotehniko

Žiga Bobnar

Wi-Fi radio

Projektna naloga

Modul B - Vgrajeni sistemi

Univerzitetni študijski program prve stopnje Elektrotehnika

Ljubljana, Junij 2021

Vsebina

1 Predstavitev projekta

V tej projektni nalogi bom predstavil izdelavo predvajalnika glasbe z uporabo mikrokrmilnika. Pri tem je bil cilj kot izziv uporabiti brezžično komunikacijo preko Wi-Fi za prenos zvočnih podatkov. Hkrati pa sem si zastavil nekakšno omejitev, da za celoten projekt uporabim samo material, ki ga imam že na voljo doma, to pa pomeni brez uporabe kakšnih zelo namenskih čipov (na primer MP3 dekodirnik), z izjemo čipov, ki jih je možno enostavno "reciklirati"iz izrabljenih oziroma za današnje standarde zastarelih naprav (na primer star avtoradio).

1.1 Zahteve projekta

Končano prvo fazo projekta (domena te projektne naloge) tako interpretiram kot nekakšno osnovno implementacijo ideje (ang. proof of concept), ki lahko brezžično predvaja glasbo iz strežnika, na zahtevo ustavi predvajanje oziroma ga nadaljuje, zvok pa naj bo zadosti kvaliteten recimo za uporabo v kakšni delavnici (torej primerljiv z FM sprejemnikom, ne neka avdiofilska kvaliteta).

Končan izdelek naj bo tudi prenosljiv, torej napajanje iz baterije, za enostavnost pa predpostavljam da je polnjenje Li-Ion baterij izvedeno zunaj izdelka, saj bi bilo sicer potrebno poskrbeti za ustrezno zaščito baterij in elektronike.

1.2 Osnovna struktura projekta

Celoten projekt sem tekom razvoja razdelil na tri ločene segmente. Prvi del je razvojna ploščica Arduino Due (v nadaljevanju Due), na kateri je mikrokrmilnik Atmel SAM3X8E. To je 32-bitni ARM mikrokrmilniški sistem, ki je jedro projekta in opravlja večino nalog, na primer krmiljenje LCD zaslona, komunikacija z modulom za brezžično povezavo ter predvajanje zvoka preko vgrajenega DA (digitalno-analognega) pretvornika. Drugi del predstavlja modul za brezžično komunikacijo, in sicer sem uporabil modul ESP-01, na katerem je mikrokrmilnik Espressif ESP8266EX (v nadaljevanju samo ESP8266). Ploščico Due sem z modulom ESP8266 povezal preko UART vmesnika. Tretji in zadnji del projekta pa je enostaven računalniški program, ki deluje kot strežnik za pretakanje glasbe (z ustreznim kodiranjem in vzorčno frekvenco), obenem pa omogoča tudi na primer prenos podatkov o trenutnem času, s čimer se izognem potrebi po ročnem nastavljanju ure.

1.3 Seznam uporabljenih elementov:

- Razvojna ploščica Arduino DUE (s krmilnikom Atmel SAM3X8E)
- Modul ESP-01 (s krmilnikom Espressif ESP8266EX)
- Dupont žičke za hitro povezovanje prototip (ang. Dupont jumper wire)
- Linearni regulator Motorola 7805CT (za zagotavljanje ustrezne 5 V napetosti za Arduino)
- Integrirani audio ojačevalnik TDA2030 (izvor neznan, verjetno iz nekega avtoradia)
- LCD zaslon, kompatibilen z vmesnikom Hitachi HD44780 (vzeto iz starega DVD predvajalnika)
- 6 mm pritisni gumb (taktilna tipka, ang. tactile button, reciklirano iz starega avtoradia)

- Li-Ion 3.7 V baterije tipa (dimenzije) 18650 z nominalno kapaciteto okrog 2400 mAh (baterije so pogosto uporabljene kot celice v starejših akumulatorjih prenosnih računalnikov, kjer lahko ena slaba baterija povzroči, da postane na prvi pogled celoten akumulator neuporaben, v resnici pa so nekatere posamezne celice lahko še povsem dobre, sploh za uporabo v nezahtevnih vezjih)

- Zvočnik (v mojem primeru bo to manjši 4 ohmski zvočnik)

- Ohišje (odločil sem se za 3D tiskanje ohišja, možna pa bi bila tudi uporaba lesa, plastičnih plošč, pločevine...)

1.4 Projekt due-radio (Arduino Due)

Za projekt sem uporabil znanje, enako programsko okolje (program Code::Blocks) ter kodo, ki smo jo pisali tekom laboratorijskih vaj pri predmetu Programiranje vgrajenih sistemov (Modul B na univerzitetnem programu elektrotehnike). Koda je napisana v jeziku C.

Projekt sem tako izdelal po predlogi, ki vsebuje nastavitve prevajalnika in razhroščevanja za krmilnik SAM3X9E. Preko JTAG priključkov priključen vmesnik Olimex ARM-USB-OCD-H, ki deluje kot programator in razhroščevalnik in omogoča enostavnejše programiranje.

V tem projektu sem uporabljal določene knjižnice ASF (ang. Advanced software framework, prej tudi Atmel Software Framework), kar omogoča hitrejši razvoj funkcionalnosti z uporabo že pripravljenih metod, namesto direktne interakcije z registri (v nekaterih primerih pa je le-ta hitrejša). Te knjižnice so na primer *clock*, *delay*, *ioport*, *serial*, *dacc*, *pio*, *tc*, *uart* ter *usart*.

Poleg teh knjižnic sem napisal oziroma dopolnjeval nekaj lastnih, ki smo jih razvijali na laboratorijskih vajah, primer tega so na primer gonilnik za LCD, FIFO vrsta (ang. first in, first out), gonilnik za DAC (bazira na obstoječem *dacc* gonilniku iz ASF) ter serijski vmesniki za uporabo UART in USART komunikacije na višjih nivojih.

1.4.1 Logična zgradba projekta

Ker je s časom postalo dodajanje daljših kosov kode v datoteke, ki sicer nimajo veze z dejansko prvotno mišljeno funkcijo te datoteke (na primer beleženje napak v dnevnik dogodkov znotraj gonilnika za komunikacijo z Wi-Fi, za kar hitro postane precej nepraktično vsakič podvajati celotno kodo za pisanje neposredno v medpomnilnik dnevnika), sem se odločil za lažji razvoj postaviti nekakšno logično delitev in abstrakcijo kode, s tem pa možnost za enostavno prilagajanje obstoječe kode na novo strojno opremo (recimo povsem drugačen LCD, zunanji DAC).

Tako je možno sedaj razdeliti projekt na naslednje module (vse, kar je potrebno za uporabo kode enega modula v drugem modulu naj bi bila vključitev datoteke *.h* modula in klic ustrezne funkcije):

- *esp_module.h*: Wi-Fi komunikacija, že glede na naslov projektne naloge verjetno najbolj ključen del celotnega projekta. Ta omogoča komunikacijo z modulom ESP-01, z uporabo strukturiranih ukazov in odzivov, preko serijskega vmesnika. Glavni problem te komunikacije je bila zagotovo visoka odzivnost, saj je za občutek realnočasnosti predvajanja posnetka potrebno zvočne vzorce v ustreznih intervalih nastavljati na izhod.

- *audio_player.h*: Predvajalnik glasbe, skrbi za prenos naslednjega segmenta posnetka, posredovanje trenutne vrednosti signala gonilniku digitalno-analognega pretvornika ter hranjenje informacij o predvajanem posnetku.

- *lcd.h*: LCD gonilnik za 16x2 zaslone, kompatibilne z HD44780 protokolom. Ta vsebuje funkcije za inicializacijo zaslona, čiščenje, nastavljanje besedila, poleg tega pa funkcije, ki so po funkcionalnosti zelo podobne funkciji *printf* in omogoča zelo enostavno formatiranje besedila (vstavljanje vrednosti dinamičnih argumentov v predpripravljeni obliki).

- *dac.h*: DAC gonilnik, omogoča inicializacijo ter nastavljanje izhodnih vrednosti na izbrani kanal digitalno-analognega pretvornika.

- *fifo.h*: FIFO vrsta, omogoča ciklično branje do zadnjega zapisanega elementa ter pisanje do zadnjega prebranega elementa. Poleg tega pa omogoča

tudi "kukanje"(ang. peek), kar poenostavi preverjanje naslednjega podatka brez odstranjevanja.

- *console.h in console_definitions.h*: Serijska konzola. En od problemov programiranja tega projekta (več o tem v nadaljevanju), je omejeno število znakov, ki se lahko na zaslonu prikažejo. Za namene enostavnejšega pregleda nad dejanskim dogajanjem sem zato napisal vmesnik za komunikacijo z računalnikom preko serijske konzole. Ta vmesnik omogoča poleg pisanja besedila formatiranega podobno kot pri LCD gonilniku s stilom *printf* tudi uporabo ukazov VT100, ki v večini modernih programov omogočajo spreminjanje barve besedila, torej je možno izdelati manj monoton pregled nad trenutnim stanjem.

- *timeguard.h*: Za potrebe omejevanja dovoljenega časa izvajanja nekaterih ukazov (na primer povezovanje v omrežje Wi-Fi, ki lahko traja), sem uporabil časovne števce, vgrajene v mikrokrmilnik. Z uporabo funkcij pa je možno pridobiti vrednost ustrezno skalirano v milisekunde oziroma sekunde. To omogoča, da lahko program, ki se izvaja v neskončni zanki prekine izvajanje, če je le-to predolgo.

- *buttons.h*: Gonilnik za odzivanje na pritiske gumbov. Uporabljen za človeško interakcijo s programom.

1.5 Projekt modem-esp8266 (ESP8266)

Drugi del projekta kot že prej omenjeno sestavlja modul ESP-01. To je modul, ki se je na tržišču pojavil okrog leta 2014, vgrajen mikrokrmilnik pa med drugim omogoča povezavo in komunikacijo z Wi-Fi napravami, ki delujejo pri frekvencah 2.4 GHz (to pomeni, da mora biti ta način delovanja vključen na sodobnejših dostopnih točkah, ki omogočajo tudi delovanje pri frekvencah 5 GHz). Teoretično bi lahko uporabil sam mikrokrmilnik na ploščici (ESP8266) za brezžično predvajanje glasbe, ampak modul nima zadosti priključkov. Na različici 01 je samo 8 priključkov, od tega pa so samo 4 priključki na voljo za programirljive izhode, kar pomeni da bi bilo precej oteženo priključiti še kakšne tipke in zaslon.

Zaradi teh omejitev bo modul opravljal samo nekakšno vlogo modema, ki prek serijske povezave sprejema ukaze od Due in jih ustrezno interpretira ter pošlje po brezžičnem omrežju in vrne ustrezno oblikovan rezultat.

Projekt je izdelan v okolju PlatformIO, ki je popularna razširitev urejevalnika kode Visual Studio Code in omogoča programiranje velikega nabora vgrajenih sistemov. Za enostavnost (in ker sem želel čim večji fokus projekta posvetiti projektu na Due) sem kot osnovo uporabil knjižnice Arduino okolja, ki omogočajo enostavno sestavljanje besedilnih spremenljivk (ang. string). Poleg tega sem uporabil knjižnici *ESP8266WiFi.h* in *ESP8266HTTPClient.h*, napisani s strani proizvajalca čipa, ki omogočata enostavno povezovanje na dostopne točke in povezavno na HTTP strežnik.

Vsa koda projekta se nahaja v datoteki *main.cpp* in vsebuje vse od inicializacije do procesiranja in izvajanja ukazov na enem mestu. Za ta pristop sem se odločil, ker se mi projekt zdi dovolj enostaven, da ločitev na več datotek še ni potrebna.

1.6 Projekt wav-server (HTTP strežnik)

Za delovanje prenosa glasbe, sem na začetku imel v mislih direktno povezavo na že obstoječe internetne radio postaje. Te pogosto prenašajo zvok v neki različici MP3 kodiranja. Problem pa je ne samo v omejeni procesorski zmogljivosti krmilnika, ampak tudi v mojem nepoznavanju celotnega kompleksnega procesa obdelave MP3 signala v surov avdio signal. Možna bi bila uporaba oziroma adaptacija kakšne že obstoječe kode, ki počne to pretvorbo, a sem se odločil za drugačen pristop.

Pri razvoju projekta je bilo potrebno vse skupaj kar velikokrat testirati. To pomeni, da je vsaka dodatna povezava na zunanji strežnik nova spremenljivka, na katero imamo malo vpliva in lahko prinaša frustracije, kadar ne deluje pravilno. Temu sem se sprva poskušal izogniti z lokalno namestitvijo programa, ki bi deloval na enak način kot tisti komercialni programi, ki jih uporabljajo radijske postaje. Problem se pojavi ker imajo vsi ti programi na zastonjski različici zelo omejene funkcionalnosti in je njihova konfiguracija veliko bolj zapletena, kot bi morala biti

za enostavno pretakanje glasbe.

Zato sem v projekt dodal še tretji del, ki pravzaprav na zunaj izgleda kot čisto običajen HTTP strežnik, torej deluje na enakem protokolu kot običajne spletne strani. Izkoristil pa sem možnost programskega vračanja odziva tako, da lahko krmilnik zahteva s strežnika določen krajši segment izbranega posnetka (ang. chunk), vrne pa se kar neposredno zaporedje podatkov, kot bi bilo to kodirano v WAV formatu. WAV format omogoča precej bolj enostavno uporabo za nastavljanje izhoda, saj dejansko predstavlja zaporedne vrednosti v enakomernih intervalih s frekvenco vzorčenja (ang. sampling frequency). To pomneni, da potrebuje krmilnik samo pridobiti vrednosti posnetka in jih zaporedno v ustreznem časovnem intervalu nastavljati na izhod digitalno-analognega pretvornika.

Ker se ta program izvaja na računalniku, ki je dovolj zmogljiv za višjenivojske jezike, sem se odločil za izdelavo projekta na osnovi okolja Node.js in sicer v skriptnem jeziku TypeScript. To je nadnabor (ang. superset) v spletnem svetu znanega jezika JavaScript. Za razliko od slednjega ima TypeScript med drugim sposobnost preverjanja veljavnosti tipa spremenljivk in sintakse, uporabo razredov (ang. class) in vmesnikov (ang. interface). Vse to omogoča precej enostavnejše odkrivanje napak v primerjavi z JavaScript kodo, hkrati pa je še vedno na precej visokem nivoju programiranja, tako da je programiranje relativno hitro.

Za moje potrebe najpomembnejša lastnost pa je to, da ima že ogromno izdelanih knjižnic s kodo. Z njihovo uporabo lahko zelo hitro izdelamo spletni strežnik, ki vrača sprogramirane dinamične odzive. Poleg tega pa obstajajo knjižnice za branje in operacije z WAV posnetki.

2 Izdelava projekta

V tem poglavju bom predstavil sam proces izdelave projekta. Skozi izdelavo ne bom šel v kronološkem zaporedju dejanskega razvoja, saj se je marsikatera funkcionalnost prepletala z drugimi. Poskusil pa bom opisati proces tako, da mu je možno slediti in ob tem v vsakem naslednjem delu dograjevati izdelek. Torej v vsakem poglavju bom poskusil izdelati nekakšen zaključen funkcionalni del izdelka.

2.1 Krmiljenje LCD zaslona

Prva težava, ki se pojavi je opazovanje, kaj se z izdelkom dogaja. Za povratno informacijo je sicer možno z uporabo razhroščevalnika ustaviti program in spremljati dogajanje. Vendar pa to predstavlja zahtevo po računalniku. Precej bolj enostavno s stališča končnega uporabnika, je imeti že v sam izdelek vgrajen zaslon, ki lahko prikazuje trenutno stanje oziroma morebitne napake.

2.1.1 LCD HD44780

Izbrani LCD uporablja krmilnik kompatibilen s precej pogosto uporabljenim protokolom Hitachi HD44780. Ta krmilnik omogoča, da z ukazi lahko v zaslon zapišemo ustrezne vrednosti za prikaz besedila, po tem pa ne potrebujemo več skrbeti za periodično osveževanje samega zaslona, saj to dela vgrajen krmilnik. Poleg tega omogoča dva načina prenosa podatkov, 8-bitni ali dvakrat po 4-bitni. Zasloni tega tipa se lahko razlikujejo med seboj na različne načine, na primer število vrstic, število stolpcev, število pikslov na znak, barva ozadja, barva

ospredja, možnost osvetlitve in barva osvetlitve. Najpogostejši so zasloni z znaki v 16 stolpcih in 2 vrsticah.

Za samo analizo protokola je najboljše uporabiti kar dokumentacijo proizvajalca (oziroma kar izvornega HD44780), ki jo je možno dobiti na internetu. Za pošiljanje ukazov v sam zaslon so pomembni priključki E (Enable), RS (Register select), R/W (Read/Write) in podatkovne linije D7-D0. Kadar ima RS logično vrednost 0 pomeni uporabo ukaznega registra, pri vrednosti 1 pa je aktiven podatkovni register (za znake). Kadar je R/W enak 0, je zaslon konfiguriran za branje v smeri iz zaslona, za vrednost 1 pomeni pisanje v zaslon (naj bo to ukaz ali pa vrednost znaka). Za prenos vsake vrednosti v zaslon moramo linijo E za kratek čas postaviti na vrednost 1 in nato nazaj na 0.

Seznam ukazov in določenih argumentov je prikazan v spodnji tabeli (povzeto po dokumentaciji):

Tabela 2.1: Ukazi za HD44780 (ang. instruction set)

Instruction	RS	R/W	D7-D0
Clear display	0	0	0000 0001
Return home	0	0	0000 001-
Entry mode set	0	0	0000 01IS
I = I/D==1 - increment			
I = I/D==0 - decrement			
S==1 - Accompanies display shift			
Display on/off control	0	0	0000 1DCB
D==1 - sets display on			
D==0 - sets display off			
C==1 - sets cursor on			
C==0 - sets cursor off			
B==1 - enables cursor blinking			
B==0 - disables cursor blinking			
Cursor or display shift	0	0	0001 SR–
S = S/C==1 - display shift			
S = S/C==0 - cursor move			
R = R/L==1 - shift to the right			
R = R/L==0 - shift to the left			
Function set	0	0	001L NF–
L = DL==1 - 8-bit data length mode			

L = DL==0 - 4-bit data length mode			
N==1 - 2 lines			
N==0 - 1 line			
F==1 - 5x10 dots character font			
F==0 - 5x8 dots character font			
Set CGRAM address	0	0	01AA AAAA
A = ACG - CGRAM address (character gen RAM)			
Set DDRAM address	0	0	1AAA AAAA
A = ADD - DDRAM address (display data RAM)			
Read busy flag address	0	1	BAAA AAAA
B = BF==1 - internally operating			
B = BF==0 - instructions acceptable			
A = AC - Address counter			
Write data to CG or DDRAM	1	0	[Write data value]
Read data from CG or DDRAM	1	1	[Read data value]

Priključitev priključkov zaslona je možna na veliko različnih načinov. V mojem primeru sem se odločil uporabiti isto priključitev kot smo jo uporabljali na laboratorijskih vajah. Zaporedje priključkov na LCD je odvisno od samega tipa priključka (lahko 16 priključkov en za drugim, lahko pa v konektorju 8x2). Za natančno informacijo o zaporedju je pametno preveriti v dokumentaciji dejanskega zaslona. Običajno pa si vsaj glede oznak sledijo v zaporedju prikazanem v spodnji tabeli:

Tabela 2.2: Priključitev LCD na Adruino Due

PIN	Oznaka	Arduino Due PIN	Opis priključka
1	GND	GND	Masa zaslona
2	VCC	5V	Napajanje zaslona
3	Vo	GND	Nastavitev kontrasta
4	RS	D51 (*C.12*)	Register select
5	R/W	D49 (*C.14*)	Read/Write
6	E	D47 (*C.16*)	Enable
7	D0	-	
8	D1	-	
9	D2	-	

10	D3	-	
11	D4	D50 (*C.13*)	
12	D5	D48 (*C.15*)	
13	D6	D46 (*C.17*)	
14	D7	D44 (*C.19*)	
15	A (+)	-	Anoda osvetlitve - pozitivni priključek
16	K (-)	-	Katoda osvetlitve - negativni priključek

Ker pa sem želel ohraniti nekaj fleksibilnosti glede priključitve LCD zaslona, sem ustvaril strukturo za nastavitev priključkov, poleg priključkov pa vsebuje še pomnilnik za izpis znakov ter kazalce na začetek prve ter začetek druge vrstice (to pride velikokrat prav, kadar je potrebno prvo vrstico pustiti pri miru). Enostavna koda te strukture izgleda tako:

```
1  struct _lcd {
2      uint32_t rs;
3      uint32_t rw;
4      uint32_t enable;
5
6      uint32_t d4;
7      uint32_t d5;
8      uint32_t d6;
9      uint32_t d7;
10
11     char __lcd_buffer[33]; // Pomnilnik za znake na zaslonu
12     char* _lcd_string;     // Kazalec na zacetek pomnilnika
13     char* lcd_upper;       // Kazalec na zacetek prve vrstice
14     char* lcd_lower;       // Kazalec na zacetek druge vrstice
15 };
16 typedef struct _lcd lcd_t;
```

V nizkonivojske funkcije se ne bom preveč poglabljal, saj je za običajno uporabo povsem dovolj uporaba "visokonivojskih"abstrakcij, kot je na primer inicializacija z `void lcd_init(lcd_t* lcd)`, izpis medpomnilnika na zaslon `void lcd_write_string(lcd_t* lcd)`, izpis znakovnega niza na zaslon `void lcd_write_string(lcd_t* lcd, uint8_t* value)` ter funkcije za formatirano izpisovanje v zaslon `void lcd_write_formatted(lcd_t* lcd, const char* format, ...)`

, `void lcd_write_upper_formatted(lcd_t* lcd, const char* format, ...)` in `void lcd_write_lower_formatted(lcd_t* lcd, const char* format, ...)`. Poleg tega pa sta še funkciji za čiščenje vrstic `void lcd_clear_upper(lcd_t* lcd)` in `void lcd_clear_lower(lcd_t* lcd)`.

Omembe vredna je mogoče le implementacija pisanja v 4-bitnem načinu, kjer se najprej v zaslon pošljejo podatki D7-D4, nato pa takoj še D3-D0:

```
1  void lcd_driver_raw_send(lcd_t* lcd, uint8_t value, bool
       is_command) {
2      ioport_set_pin_level(lcd->rs, !is_command);
3      ioport_set_pin_level(lcd->rw, 0);
4
5      // Poslje bite D7-D4
6      lcd_driver_raw_data_pins_set(lcd, value >> 4);
7
8      //   Ekvivalent za 8-bitni nacin:
9      // ioport_set_pin_level(lcd->d7, value & (1 << 7));
10     // ioport_set_pin_level(lcd->d6, value & (1 << 6));
11     // ioport_set_pin_level(lcd->d5, value & (1 << 5));
12     // ioport_set_pin_level(lcd->d4, value & (1 << 4));
13
14     lcd_driver_pulse_enable_pin(lcd);
15
16
17     // Poslje bite D3-D0
18     lcd_driver_raw_data_pins_set(lcd, value);
19
20     //   Ekvivalent za 8-bitni nacin (d3 do d0 ne obstajajo v
           strukturi!):
21     // ioport_set_pin_level(lcd->d3, value & (1 << 3));
22     // ioport_set_pin_level(lcd->d2, value & (1 << 2));
23     // ioport_set_pin_level(lcd->d1, value & (1 << 1));
24     // ioport_set_pin_level(lcd->d0, value & (1 << 0));
25
26     lcd_driver_pulse_enable_pin(lcd);
27 }
```

2.1.2 *lcd.h* referenca

Visokonivojske funkcije:

`void lcd_init(lcd_t* lcd)`
Inicializira LCD gonilnik z uporabo podane strukture.

`void lcd_write_lcd_string(lcd_t* lcd)`
Pošlje vrednosti iz `lcd->__lcd_buffer` v LCD.

`void lcd_write_string(lcd_t* lcd, uint8_t* value)`
Pošlje vrednosti iz `value` v LCD.

`void lcd_write_formatted(lcd_t* lcd, const char* format, ...)`
Izpiše formatiran tekst na začetek zaslona in z možnostjo preliva v naslednjo vrstico.

`void lcd_write_upper_formatted(lcd_t* lcd, const char* format, ...)`
Izpiše formatiran tekst samo v zgornjo vrstico.

`void lcd_write_lower_formatted(lcd_t* lcd, const char* format, ...)`
Izpiše formatiran tekst samo v spodnjo vrstico.

`void lcd_write_string_at_cursor(lcd_t* lcd, uint8_t* value, uint8_t length)`
Izpiše tekst od trenutne pozicije kurzorja naprej.

`void lcd_wait_busy_status(lcd_t* lcd)`
Počaka, da LCD signalizira konec operacij.

`void lcd_clear_upper(lcd_t* lcd)`
Izprazni vsebino zgornje vrstice v `lcd->__lcd_buffer`.

`void lcd_clear_lower(lcd_t* lcd)`
Izprazni vsebino spodnje vrstice v `lcd->__lcd_buffer`.

2.2 Implementacija DAC (digitalno-analognega pretvornika)

DAC predstavlja povezavo med digitalnimi vrednostmi, ki jih lahko zapišemo z dvojiškimi vrednostmi (biti), z analognim svetom, kjer so signali zvezne veličine. Omogoča pretvorbo večbitnih vrednosti v diskretne analogne nivoje, katerih razlika je vedno manjša, čim višjo resolucijo ima pretvornik (število bitov pretvornika).

2.2.1 Arduino Due DACC

Arduino Due (oziroma SAM3X8E) ima že vgrajen in z zunanje strani dostopna dva kanala DAC. Ta dva kanala se nahajata na pinih DAC0 in DAC1. Krmiljenje pa poteka preko periferne enote dacc (ang. digital-to-analog converter controller) vgrajene v sam mikrokrmilnik. Krmilnik ima na voljo 12 bitov resolucije, dejansko pa bom zaradi enostavnosti uporabljal 8-bitne vrednosti, saj je le-te najbolj enostavno prenašati preko serijskih vmesnikov.

Podobno kot že za LCD gonilnik, sem za lažje konfiguriranje pripravil strukturo, ki vsebuje indeks kanala, določene nastavitve za območje vrednosti ter frekvenco vzorčenja za prožilnike (trenutno ni v implementirano):

```
1 struct _dac {
2     uint32_t channel;            // Kanal DAC kontrolerja
3     uint32_t max_value;          // Najvisja vrednost izhoda
4     uint32_t min_value;          // Najnizja vrednost izhoda
5     uint32_t sampling_frequency; // Frekvenca vzorcenja
6 };
7 typedef struct _dac dac_t;
```

Kar se implementacije tiče je trenutno ta datoteka samo abstrakcija dejanske strojne opreme za enostavno in hitro interakcijo.

2.2.2 *dac.h* referenca

`void dac_init(dac_t* dac)`
Inicializira DACC vmesnik.

`bool dac_tx_ready(dac_t* dac)`
Preveri, če je DACC pripravljen na naslednji podatek.

`void dac_write(dac_t* dac, uint32_t value)`
Nastavi vrednost v DACC medpomnilnik.

2.3 FIFO vrsta (First in, first out)

FIFO vrsta omogoča učinkovit način za hitro shranjevanje podatkov, ki bodo kasneje prebrani, v medpomnilnik. Pri tem ne prihaja do nobenih prilagajanj velikosti (ang. resizing), saj se podatki, ki so bili že prebrani lahko prepišejo. Osnovna struktura je torej sestavljena iz nekega kosa pomnilnika, namenjenega podatkom, podatka o velikosti tega kosa pomnilnika, ter dveh indeksov. Prvi predstavlja trenutno pozicijo branja, drugi pa pisanja.

Tako izgleda struktura FIFO vrste:

```
1 struct _fifo {
2     uint32_t read_idx;  // Trenutna pozicija za pisanje
3     uint32_t write_idx; // Trenutna pozicija za branje
4     uint32_t size;      // Stevilo elementov, ki jih vrsta hrani
5     uint8_t* buffer;    // Pomnilnik velikosti size
6 };
7 typedef struct _fifo fifo_t;
```

Osnovna implementacija FIFO vrste je pisanje N znakov v medpomnilnik ter branje N znakov iz njega. Pisanje je dokaj trivialno, dokler imamo na voljo podatke, jih zapisujemo v pozicijo trenutnega indeksa pisanja in ga povečujemo, če pridemo indeksa, ki je enak velikosti vrste, pa se ponastavi nazaj na 0 in lahko pišemo naprej. Če je slučajno indeks pisanja en korak pred indeksom branja, pa se ustavimo, saj namreč ne želimo prepisati še neprebranih podatkov. Branje

deluje na podoben način, povečujemo indeks branja, dokler ne pridemo do indeksa pisanja oziroma dokler ne preberemo željenega števila podatkov.

Kar sem opazil tekom razvoja je to, da je v osnovi čista FIFO vrsta lahko kar nekoliko omejujoča. Na primer ne vemo, če je nek podatek na voljo, če ne preberemo vsaj naslednjega podatka, s tem pa ta podatek ni več del FIFO vrste (razen če manipuliramo neposredno FIFO strukturo, kar pa mislim da ni pametno, saj precej hitro ob nepazljivostih lahko pride do kakšne napake in vrsta postane neuporabna). Manjka torej funkcija, ki bi omogočala "kukanje"(ang. peek) naslednjih N vrednosti in funkcija, ki preveri če sta indeksa branja in pisanja različna. Tudi implementacija teh dveh funkcij je dokaj enostavna, dodal pa sem še funkcijo za "ponastavitev"vrste, ki enostavno oba indeksa vrne na 0 in s tem počisti neprebrane podatke (podatki sicer še vedno ostanejo v pomnilniku, le FIFO funkcije ne morejo dostopati do njih!). S tem je implementacija FIFO vrste precej bolj prijazna za programiranje, možne pa so še raznorazne nadaljnje razširitve.

2.3.1 *fifo.h* referenca

`uint32_t fifo_read(fifo_t* fifo, uint8_t* data_buffer, uint32_t n)`
Prebere `n` vrednosti iz FIFO vrste v `data_buffer`.

`uint32_t fifo_peek(fifo_t* fifo, uint8_t* data_buffer, uint32_t n)`
Prebere `n` vrednosti iz FIFO vrste v `data_buffer`, vendar brez premikanja bralnega indeksa naprej.

`bool fifo_has_next_item(fifo_t* fifo)`
Preveri in vrne, če ima FIFO vrsta na voljo vsaj en element.

`uint32_t fifo_write(fifo_t* fifo, uint8_t* data_ptr, uint32_t n)`
Zapiše `n` vrednosti iz `data_ptr` v FIFO vrsto.

`uint32_t fifo_write_single(fifo_t* fifo, uint8_t value);`
Zapiše vrednost `value` v FIFO vrsto.

`void fifo_discard(fifo_t* fifo)`
Postavi indeksa branja in pisanja na 0.

2.4 Konzola (komunikacija z računalnikom)

Precej hitro se je pojavila težava, da je za vsako spremembo bilo potrebno postavljati točke zaustavitve programa (ang. breakpoint), kjer naj se program ustavi in lahko ročno spremljamo njegov potek. To lahko postane precej zamudno, kadar je potrebno spremljati spremenljivke realnega okolja (na primer povezavo na brezžično omrežje), sploh pa kadar je preko serijske povezave priključena še druga naprava, ki pošilja podatke (modem), kjer je ustavljanje programa za pregled podatkov zelo nepraktično. Zaradi tega sem se odločil implementirati komunikacijo z računalnikom, pri čemer lahko vidim vse, kar se dogaja v krmilniku.

Arduino Due ima priročno že povezan serijski vmesnik na USB pretvornik (ta je čip MEGA16u2, priključen pa je na micro USB konektor, ki je najbližje okroglega napajalnega vhoda - programming port). Komunikacija poteka v obliki protokola UART. Za branje podatkov sem priročno izkoristil prej implementirano FIFO vrsto, v katero se začasno shranjujejo podatki takoj, ko so dekodirani.

Poleg samega vmesnika za interakcijo z konzolo, sem dodal tudi definicije pogosto uporabljenih konstant v datoteko *console_definitions.h*. Te konstante so na primer nekatere standardne baud hitrosti prenosa podatkov po UART, posebni ASCII znaki ter VT100 ukazi. Veliko današnjih emulatorjev konzole (programske konzole) upošteva te ukaze, z njimi pa je možno spreminjati položaj kurzorja, brisanje znakov, nastavljanje barve ozadja in barve besedila, spreminjanje velikosti znakov in verjetno še kaj več. Ukazi so včasih imenovani tudi "ANSI escape codes", saj je prvi znak zaporedja vedno ASCII vrednost za "escape".

Zaenkrat je komunikacija s konzolo samo enosmerna, z dovolj truda bi pa verjetno bilo povsem možno izdelati ustrezen vmesnik, ki lahko upošteva tudi ukaze dobljene z računalnika. To pomeni, da bi bilo lahko možno na primer celotno testiranje nekoliko avtomatizirati. Seveda pa to ni cilj te faze projekta.

2.4.1 *console.h* referenca

`void console_init(void)`
Inicializira konzolo z ustrezno baud hitrostjo in omogoči ustrezne izhodne

prikljucke.

`void console_enable(void)`
Omogoči sprejemanje in oddajanje podatkov preko serijskega vmesnika.

`void console_process_input(void)`
NI implementirano! V prihodnje je tu lahko procesiranje vhodnih ukazov, ki pridejo po vmesniku iz računalnika.

`void console_put_char(const uint8_t value)`
Zapiše en znak v konzolo.

`void console_put_raw_string(const char* str)`
Zapiše niz znakov v konzolo.

`void console_put_line(const char* str)`
Zapiše niz znakov v konzolo in doda znak za novo vrstico na konec.

`void console_put(const char* str)`
Zapiše niz znakov v konzolo (ekvivalentno `console_put_raw_string`).

`void console_put_formatted(const char* format, ...)`
Zapiše formatiran niz znakov v konzolo in doda novo vrstico na konec.

`bool console_char_ready(void)`
Preveri in vrne, če je v konzoli na voljo kakšen znak za sprejem.

`void console_wait_until_char_ready(void)`
Počaka, da je na voljo znak za sprejem.

`uint8_t console_get_char(void)`
Prebere znak iz medpomnilnika za sprejem.

`uint8_t console_peek_char(void)`
Prebere znak iz medpomnilnika za sprejem brez spreminjanja trenutne pozicije branja.

2.5 Enostaven časovnik (timeguard)

Ker se nekatere operacije lahko izvajajo dalj časa in pogosto preverjamo njihovo stanje v neskončni "whileżanki, se mi zdi pametno implementirati nek način, da po določenem času obupamo nad preverjanjem in določimo da je ukaz neuspešen.

Implementacija timeguard za delovanje uporablja interni časovni števec, omogoča pa branje trenutnih vrednosti časa v milisekundah in sekundah ter računanje razlike med trenutnim časom in podanim časom.

2.5.1 *timeguard.h* referenca

`void timeguard_init(void)`
Inicializira ustrezne periferne enote za uporabo časovnega števca.

`int32_t timeguard_get_time(void)`
Prebere vrednost iz časovnega števca. Ta vrednost še ni skalirana.

`int32_t timeguard_get_time_ms(void)`
Prebere vrednost trenutnega časa v milisekundah.

`int32_t timeguard_get_diff_ms(int32_t previous_time_ms)`
Izračuna razliko med trenutnim in podanim časom v milisekundah.

`int32_t timeguard_get_time_s(void)`
Prebere vrednost trenutnega časa v sekundah.

`int32_t timeguard_get_diff_s(int32_t previous_time_s)`
Izračuna razliko med trenutnim in podanim časom v sekundah.

2.6 Gonilnik za tipke

Človeška interakcija z napravami je še najbolj enostavna za uporabo in implementacijo, če uporabimo tipke. Te večinoma omogočajo odlično taktilno povratno informacijo ob pritisku in sprostitvi (občuten klik ter zvok). V primerjavi z na primer zasloni na dotik, kjer je ta povratna informacija običajno vizualna,

je tipke tudi veliko bolj enostavno uporabiti v električnem vezju, saj lahko samo preverjamo, kdaj je njihovo notranje stikalo sklenjeno in kdaj ne.

Naloga gonilnika za tipke je torej, da ustrezno reagira na spremembe stanja tipk. V vgrajenih napravah pri neposredni povezavi tipk na priključke mikrokrmilnika, lahko to delamo na vsaj dva različna načina. Prvi v ustreznih časovnih intervalih prebere stanja tipk, drugi pa lahko izkoristi v mikrokrmilnik vgrajeno možnost uporabe prekinitev (ang. interrupt). Slednji način omogoča, da program ves čas, kadar se s tipkami nič ne dogaja izvaja druge operacije, ko pa zazna spremembo stanja, pa skoči v funkcijo za procesiranje tega dogodka.

V mojem primeru sem uporabil kar prvi način, saj za implementacijo potrebuje samo kodo za branje ter procesiranje stanja, ki jo lahko periodično kličem ob vsaki ponovitvi glavne zanke. S tem sicer lahko zamudimo nekatere zelo kratke pritiske, če bi bilo izvajanje glavne zanke časovno predolgo. Ima pa takšen način to prednost, da je predvidljiv in se ne vsiljuje v preostale procese, kot na primer nastavljanje izhodnih vrednosti na DAC, kar se mora zgoditi v dokaj natančnih časovnih intervalih, sicer bi lahko prišlo do popačenj.

Funkcijo, ki prebira tipke in ugotavlja, katere so bile ravnokar pritisnjene in katere ravnokar spuščene, bi lahko struktuiral tako:

```
1  void buttons_check()
2  {
3      // Staticna spremenljivka hrani prejsnje stanje. Za razliko
          od globalnih spremenljivk, je ta omejena samo znotraj
          funkcije in ne "onesnazuje" globalnega dela programa.
4      static uint32_t old_state = 0;
5
6      // Posamezni biti v spremenljivki predstavljajo tocno
          doloceno tipko.
7      uint32_t state =
8          (!ioport_get_pin_level(PIO_PC26_IDX) << 3) |
9          (!ioport_get_pin_level(PIO_PC25_IDX) << 2) |
10         (!ioport_get_pin_level(PIO_PC24_IDX) << 1) |
11         (!ioport_get_pin_level(PIO_PC23_IDX) << 0);
12
13     // Izracun spremembe stanj
14     uint32_t rising_edge = ~old_state & state;
15     uint32_t falling_edge = old_state & ~state;
```

```
16
17      // Koda, ki ustrezno odreagira na spremembe
18      ...
19      //
20
21      // Za konec moramo pripraviti funkcijo za naslednje izvajanje
            .
22      old_state = state;
23  }
```

2.6.1 *buttons.h* referenca

`void buttons_init(void)`
Inicializira ustrezne priključke za tipke.

`void buttons_process(void)`
Preveri novo stanje tipk in ustrezno reagira na morebitne pritiske.

2.7 Wi-Fi komunikacija

S kodo iz prejšnjih razdelkov, sem prišel do neke podlage platforme, na kateri pa sedaj lahko končno začnem graditi še tisti bistveni del projekta, to je komunikacija z brezžičnim omrežjem. Ker krmilnik SAM3X8E oziroma Arduino Due nima nobenega vgrajenega modula za brezžično komunikacijo. Možna je sicer uporaba t. i. Arduino WiFi 101 Shield, ki ima podporo za 5 V in 3.3 V način delovanja, slednji je obvezen za krmilnik na Due, problem pa je, da izdelek ni več v prodaji in niti ni bil precej poceni rešitev (cena naj bi bila okrog 45 EUR).

Zato sem se odločil, da uporabim precej dostopen modul ESP-01, ki ga je možno kupiti relativno poceni (iz uvoza stanejo pod 2 EUR). Na tem modulu je glavna komponenta mikrokrmilnik Espressif ESP8266EX, poleg tega pa običajno vsebujejo še 512 KiB oziroma 1 MiB pomnilnika flash za programiranje. Modul deluje na napetosti 3.3 V, kar pomeni, da je povsem kompatibilen z Due, vsebuje 8 priključkov:

...TODO: slika

Običajno se na takšen modul naloži vmesnik za ukaze AT (ang. AT command set), ti ukazi so poznani tudi kot Hayes command set. Ti ukazi so bili prvotno zasnovani za komunikacijo s telefonskimi modemi, uporabljajo pa se tudi na primer za komunikacijo z GSM modemi. Predvidevam, da je ta način komunikacije precej dober, sicer se ne bi pogosto uporabljal, vendar je lahko problem nepoznavanje nabora ukazov (kot v mojem primeru). Učenje in pisanje ukazov, ki bi omogočali pošiljanje in sprejemanje paketov po brezžičnem omrežju pa je tukaj samo del problema.

Drugi del je to, da bi potreboval izdelati še način oziroma program za komunikacijo s HTTP strežnikom (ang. HTTP client). Sam HTTP protokol je sicer še dokaj enostaven in ga je možno študirati iz prosto dostopnih dokumentov, kot je na primer zdaj že nekoliko zastarel dokument RFC2616, ki predstavlja osnutek Hypertext Transfer Protocol - - HTTP/1.1.

Hkrati pa sem ugotovil, da je proizvajalec samega čipa ESP8266 objavil odprtokodne knjižnice za sam čip ter knjižnice za Arduino med temi knjižnicami pa lahko najdemo med drugim tudi zelo enostavne načine za upravljanje z Wi-Fi funkcionalnostjo na čipu ((to je *ESP8266WiFi.h*) ter za komunikacijo z HTTP strežnikom (to je *ESP8266HTTPClient.h*).

Na podlagi teh ugotovitev sem se odločil za lastno implementacijo "modema", torej kot prvo stvar potrebujem neko enostavno strukturo komunikacije ukazov med moduloma, tako kot ima na primer AT točno definirane strukture ukazov in njegove odzive. Za enostavnost in preglednost sem si zamislil enostavno komunikacijo, v kateri igra Due vlogo glavne naprave (ang. master), ESP-01 pa vlogo podrejene naprave (ang. slave). Torej Due pošilja ukaze v ESP-01, slednji pa mora odgovoriti z ustreznim odgovorom glede na ukaz in v ustreznem času. Vsak ukaz in vsak odgovor (razen določenih izjem), se mora zaključiti z znakom za novo vrstico (ta znak se v programiranju zapiše kot '\n'), argumenti ukaza pa so ločeni s presledkom. Iz tega, sem definiral osnovni nabor ukazov:

Tabela 2.3: Nabor ukazov za komunikacijo z ESP-01

Ukaz	Odziv	Opis delovanja

status	*OK*	Modul vrne stanje OK v kolikor je pripravljen na sprejem ukazov.
connect %SSID %PASSWORD	*OK* oz. besedilo napake	Poskusi se povezati na dostopno točko z nazivom *%SSID* in geslom *%PASSWORD*,
Pogoste napake:		
Connection failed. SSID is empty		Manjka ime omrežja
Connection failed. Password is empty		Manjka geslo
Connection failed		Povezava ni uspela
Connection timed out		Povezovanje je trajalo predolgo
play_next	*OK*	Pošlje ukaz za prestavitev na naslednji posnetek v vrsti na strežnik
play_previous	*OK*	Pošlje ukaz za prestavitev na prejšnji posnetek v vrsti na strežnik
get_currently_playing		Pridobi informacije o trenutnem posnetku iz strežnika
	OK PLAYING {0};{1};{2};{3}	Informacije so pridobljene
Argumenti:		
{0}		Dolžina posnetka v ms
{1}		Frekvenca vzorčenja v Hz
{2}		Trenutni odsek (current chunk)
{3}		Število odsekov (total chunks)
	OK STOPPED	Predvajanje je ustavljeno
	FAIL	Napaka pri pridobivanju podatkov
get_track_info %ID		Pridobi informacije o posnetku z identifikatorjem %ID iz strežnika
	OK {0};{1}	Informacije so pridobljene
Argumenta:		
{0}		Frekvenca vzorčenja v Hz
{1}		Število odsekov (total chunks)
	FAIL	Napaka pri pridobivanju podatkov
get_chunk %ID %CHUNK_INDEX		Prenesi zahtevani odsek %CHUNK_INDEX posnetka %ID iz strežnika
	OK {0}, nato podatki	Odsek uspešno prenešen

Argumenti:		
{0}		Dolžina podatkov
podatki		Podatki so celoštevilske vrednosti signala
	FAIL	Napaka pri pridobivanju podatkov
get_current_time		Pridobi trenutni čas iz strežnika
	OK YYYY-MM-DD HH:mm:ss	Čas pridobljen
	FAIL	Napaka pri pridobivanju podatkov

V podprojektu modem-esp8266 sem nato upoštevajoč zgornje definicije razvil program, ki preko serijskega vmesnika prebira podatke in ob vsakem znaku ' n' izvede ustrezen ukaz. Ukazi na strežnik uporabljajo *HTTPClient*, ki omogoča pošiljanje zahtev na strežnik in enostavno branje odziva. Na drugi strani pa je Due s projektom due-radio, v katerega sem dodal gonilnik *esp_module.h* za komnikacijo z modulom. Pri tem sem uporabil precej podoben način komunikacije kot že prej za implementacijo konzole. Podobno kot pri gonilniku za LCD zaslon, kjer sem smiselno ločil visokonivojske operacije od nizkonivojskih. Tu predstavljajo visokonivojske operacije ukazi za inicializacijo ter funkcije, ki poenostavijo pošiljanje ukazov iz definicij ter njihov sprejem. Nizkonivojske operacije pa uporabljajo predponi *esp_module_rx_* in *esp_module_tx_* in so namenjene neposrednemu pošiljanju podatkov v in iz modula.

Pri vračanju podatkov pa sem definiral še določene strukture na primer za informacije o posnetku in času, ki jih visokonivojske funkcije ustrezno uporabijo in mi omogoča hitro uporabo v drugih delih kode:

```
1 // Podatki o dolocenem posnetku
2 typedef struct {
3     //char* track_path;  // Trenutno ni implementirano
4     int32_t track_length_ms;     // Dolzina posnetka v ms
5     int32_t sampling_frequency; // Frekvenca vzorcenja v Hz
6     int32_t total_chunks;        // Skupno stevilo odsekov
7 } track_info;
8
```

```
9  // Podatki o posnetku, ki se trenutno predvaja na strezniku
10 typedef struct {
11     track_info track;       // Podatki o posnetku
12     int32_t current_chunk; // Trenuten odsek
13 } currently_playing_info;
14
15 // Trenuten cas, razdeljen v locene spremenljivke
16 typedef struct {
17     int year;
18     int month;
19     int day;
20     int hour;
21     int minutes;
22     int seconds;
23 } current_time;
```

2.7.1 *esp_module.h* referenca

`void esp_module_hardware_setup(lcd_t* lcd_ptr)`
Inicializira strojno opremo ESP modula, vhodni parameter `lcd_ptr` je privzeti LCD zaslon, na katerega naj se izpišejo morebitne napake.

`bool esp_module_init(void)`
Poskusi inicializirati ESP modul z ukazom *status*.

`bool esp_module_wifi_connect(const char* ssid, const char* password)`
Začne povezavo na brezžično omrežje.

`void esp_module_play_next()`
Pošlje ukaz za naslednji posnetek.

`void esp_module_play_previous()`
Pošlje ukaz za prejšnji posnetek.

`currently_playing_info* esp_module_get_currently_playing()`
Pridobi informacije o posnetku, ki se trenutno predvaja. V kolikor se ne predvaja, oziroma je prišlo do napake, bo vrnjen kazalec enak `NULL`.

`track_info* esp_module_get_track_info(int track_id)`
Pridobi informacije o izbranem posnetku. V kolikor je prišlo do napake, bo vrnjen kazalec enak `NULL`.

`uint8_t* esp_module_get_chunk(int track_id, int chunk_index)`
Začne prenos segmenta izbranega posnetka. Dolžina segmenta se shrani in podatki se zapisujejo v FIFO vrsto, dokler niso vsi prebrani. Med branjem ni možna prekinitev.

`current_time* esp_module_get_current_time()`
Pridobi informacijo o trenutnem času s strežnika. V kolikor je prišlo do napake, bo vrnjen kazalec enak `NULL`.

2.8 Implementacija predvajalnika glasbe

——————————TODO!!!!

2.9 Strežnik za glasbo (projekt wav-server)

Zadnja problematika kar se programiranja tiče, je izdelava programa, na katerega se bo lahko modul povezal in z njim komuniciral. Uporabil sem platformo Node.js in skriptni jezik TypeScript. Znotraj tega pa sem uporabil knjižnice *express* za HTTP strežnik, *moment* za formatiranje časa, *wavefile* za nalaganje in ustrezno preoblikovanje zvočnega signala.

Opažam, da se programi v skriptnih jezikih precej hitro zakomplicirajo in celotna koda v eni datoteki postane zelo nepregledna. Zato sem ta podprojekt razdelil v več datotek:

- *main.ts* Zagonska skripta programa, naloži konstante za seznam predvajanja, nastavitve strežnika in zažene strežnik.
- *wavserver.ts* HTTP strežnik, sprejema zahteve, jih ustrezno izvede in vrne HTTP odziv.

- *Streamer.ts* Streamer je mišljeno v tem smislu, da skrbi za seznam predvajanja, ustrezno distribuiranje segmentov posnetka večim poslušalcem (v kolikor je to ustrezno implementirano) in ustrezno nadaljevanje na naslednji segment posnetka.

- *Playlist.ts* Seznam predvajanja skrbi za pripravo posnetkov iz seznama, hranjenje podatkov o trenutnem posnetku in premikanju po seznamu (naslednji, prejšnji posnetek). V tej datoteki se nahaja tudi definicija elementa iz seznama predvajanja, ki vsebuje posnetek (Track).

- *Track.ts* Track oziroma posnetek je objekt, ki vsebuje različne podatke o posnetku (pot do datoteke, frekvenca vzorčenja, velikost segmentov, število segmentov), poleg tega pa hrani tudi celoten posnetek v obliki 8-bitnih števil, ki se ga uporablja za rezanje na segmente. Poskrbi pa tudi za nalaganje podatkov iz datoteke in pretvorbo v ustrezno obliko (8-bitna globina z ustrezno frekvenco vzorčenja).

- *config.ts* Datoteka, ki hrani konfiguracijo strežnika. Ta datoteka naj bo kopirana iz datoteke *config.ts.example*, saj ni shranjena v orodju za beleženje zgodovine. Potrebno je dodati tudi zvočne posnetke, najboljše je ustvariti mapo *music* v isti mapi kot config.ts in posnetke dodati v to mapo. V konfiguraciji je potrebno navesti pot do vsakega od posnetekov.

2.10 Sestavljanje *main.c*

Zdaj je večina programskega dela končno napisana, manjka pa samo še dejanski program na mikrokrmilniku, ki bo vse skupaj povezal in začel predvajati glasbo.

To bo naloga datoteke *main.c*, v projektu due-radio. V njej se nahaja funkcija `int main(void)`, ki je vstopna točka programa in se zažene z mikrokrmilnikom. Tu bo torej potrebno inicializirati vso potrebno strojno opremo, se povezati na omrežje in poskrbeti, da predvajanje normalno deluje.

V smislu Arduino programiranja, sem si funkcijo *main* razširil na funkcijo za inicializacijo `bool boot_setup(void)` ter na glavno neskončno zanko `void main_loop (void)`. Znotraj funkcije *main* tako kličem funkcijo *boot_setup* in preverim njen

rezultat. V primeru, da je bil zagon neuspešen, sem se odločil da implementiram enostaven mehanizem, ki krmilnik resetira in ponovno preizkusi zagon po nekem določenem času (na primer po 10 sekundah). Koda v *main* funkciji izgleda približno tako:

```
1  int main(void)
2  {
3      ...
4      // Na zacetku je inicializacija mikrokrmilnika za pravilno
           delovanje
5
6      // Zagon sistema
7      bool setup_successful = boot_setup();
8      if (setup_successful)
9      {
10         // Zagon uspesen.
11
12         while (1)
13         {
14             // Izvajanje programa v glavni zanki.
15             main_loop();
16         }
17     }
18     else
19     {
20         // Zagon neuspesen.
21         delay_ms(3000);
22         lcd_write_upper_formatted(&lcd, "Boot␣setup␣fail");
23         lcd_write_lower_formatted(&lcd, "Cannot␣continue");
24
25         console_put_line(CONSOLE_VT100_COLOR_TEXT_RED
26         "\n\nBoot␣setup␣has␣failed...\nCannot␣continue."
               CONSOLE_VT100_COLOR_TEXT_DEFAULT);
27
28         console_put_line(CONSOLE_VT100_COLOR_TEXT_RED
29         "Resetting␣in␣10␣seconds."
               CONSOLE_VT100_COLOR_TEXT_DEFAULT);
30
31         delay_ms(10000);
32
33         // Koda za ponoven zagon sistema
34         __DSB;
```

```
35          SCB->AIRCR = ((0x5FA << SCB_AIRCR_VECTKEY_Pos) |
                SCB_AIRCR_SYSRESETREQ_Msk);
36          RSTC->RSTC_CR = RSTC_CR_KEY(0xA5) | RSTC_CR_PERRST |
                RSTC_CR_PROCRST;
37          NVIC_SystemReset();
38
39          // Prepreci izvajanje cesarkoli drugega pred
                ponastavitvijo
40          while (1)
41          {
42          }
43      }
44
45      while (1); // Backup
46  }
```

Kar se tiče zagona sistema, si procesi sledijo v naslednjem vrstnem redu:

1. Inicializacija osnovne strojne opreme mikrokrmilnika (delay, ioport, timeguard).
2. Inicializacija gonilnika za tipke.
3. Inicializacija LCD zaslona in prikaz osnovnih informacij.
4. Inicializacija serijske konzole in izpis pozdravnega sporočila.
5. Inicializacija gonilnika za digitalno-analogni pretvornik (DAC).
6. Inicializacija ESP-01 modula.
7. Povezovanje v Wi-Fi omrežje.
8. Sinhronizacija trenutnega časa.

V glavni zanki pa se periodično izvajajo naslednja opravila:

1. Procesiranje pritiskov na tipke.
2. Procesiranje vhodov iz konzole (sistem še ni implementiran, saj ni zanesljiv).

3. TODO: Izvajanje predvajalnika glasbe.

4. TODO: Posodobitev časa / prikaz na zaslonu.