Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Dežman

Nova funkcionalnost razvojnega okolja Arduino IDE

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Nikolaj Zimic

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Odprtokodno razvojno okolje Arduino IDE je enostavno razvojno okolje za enostavne rešitve, ki temeljijo na odprtih strojnih platformah. Pri platformah, ki omogočajo priključitev v omrežje lahko uporabnik nalaga programsko kodo preko omrežja. Razvojno orodje omogoča uporabo konzole preko serijske linije, oziroma USB priključka, ne omogoča pa uporabo le-te preko omrežja.

V diplomski nalogi nadgradite razvojno okolje tako, da bo uporabniku omogočalo uporabo konzole preko omrežja. Pri tem bodite pozorni predvsem na to, da bo uporaba za uporabnike čim bolj enostavna. Za testiranje uporabite modul WeMos D1 mini z mikrokontrolerjem ESP8266.

Zahvaljujem se mentorju prof. dr. Nikolaju Zimicu za pomoč in vodenje pri pisanju diplomske naloge, moji družini, ki mi je v času študija in še posebno v obdobju pisanja diplomske naloge stala ob strani in me spodbujala, in lektorici, ki je zagotovila slovnično pravilnost moje diplomske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
IDE	integrated development environment	integrirano razvojno okolje
MDNS	multicast domain name system	sistem domenskih imen preko večvrstnega oddajanja
OTA	over-the-air programming	programiranje preko zraka
SSH	secure shell	varna lupina
USB	universal serial bus	univerzalno serijsko vodilo

Povzetek

Naslov: Nova funkcionalnost razvojnega okolja Arduino IDE

Avtor: Anže Dežman

Odprtokodno razvojno okolje Arduino IDE nam poleg prenosa programov na mikrokontroler preko žične serijske povezave nudi tudi način prenosa programov preko omrežja z uporabo protokola OTA (angl. Over The Air). Ta protokol pa ne podpira pošiljanja testnih izpisov iz mikrokontrolerja v serijsko konzolo, saj je ta funkcionalnost vezana na serijsko povezavo, ki se z uporabo OTA izgubi. Cilj diplomske naloge je bil preusmeriti testne izpise preko serijske povezave v testne izpise preko omrežja. Najprej smo se s pregledom izvorne kode seznanili s privzetim postopkom pošiljanja testnih izpisov preko serijske povezave ter z delovanjem serijske konzole okolja Arduino IDE. Nato smo razvili Arduino knjižnico za preusmeritev testnih izpisov in nadgradili razvojno okolje Arduino IDE tako, da te testne izpise preko omrežja sprejme in prikaže na spremenjeni serijski konzoli. Diplomska naloga vsebuje opis izvorne kode in delovanja naše knjižnice, opis naših sprememb okolja Arduino IDE ter preprost primer uporabe naše knjižnice. Za testiranje Arduino knjižnice smo uporabili razvojno ploščico WEMOS D1 mini, ki temelji na mikrokontrolerju ESP8266. Pri razvoju smo uporabili programska jezika C++ in Java.

Ključne besede: mikrokontroler, ESP8266, Arduino, OTA.

Abstract

Title: New feature for Arduino Integrated Developement Environment

Author: Anže Dežman

The open-source Arduino Integrated Development Environment allows the programmer to not only upload programs to a micro controller with a wired serial connection, but also over the network with the use of Over The Air programming, shortened to OTA. This protocol, however, does not support the sending of debug output from the micro controller to the serial console, because this functionality depends on a serial connection that is lost when using OTA. The goal of this diploma paper was reroute debug output over serial connection to debug output over network. We began by reviewing the source code and familiarized ourselves with the default process of sending the debug output over serial connection and the workings of Arduino's serial console. Afterwards we developed an Arduino library which reroutes debug output over the network and then we modified the Arduino IDE to allow said debug output over network to be shown in its modified serial console. This diploma paper includes the description of our library's source code, its functionality, the description of our modifications to the Arduino Integrated Development Environment as well as a simple use case of our library. Our library was tested on WEMOS D1 mini development board, which is based around the ESP8266 micro controller. In the course of development C++ and Java programming languages were used.

Keywords: microcontroller, ESP8266, Arduino, OTA.

Poglavje 1

Uvod

Na trg prihaja vedno več majhnih, zmogljivih, in kar je najpomembnejše, poceni računalniških modulov oziroma razvojnih ploščic (*angl. development board*). Eden od razlogov za to je pojav poceni mikrokontrolerjev, kot je mikrokontroler *ESP8266* kitajskega podjetja *Espressif Systems*. Omenjeni mikrokontroler že tovarniško vsebuje celostno rešitev povezljivosti z brezžičnim internetom oziroma *WiFi*, vključno z anteno [42]. To programerjem omogoča takojšnje upravljanje z *WiFi* nameščanja dodatnih delov ali programskih knjižnic, ki bi to omogočale.

Od tu naprej pride na vrsto razvojna ploščica *WEMOS D1 mini*. Majhna ploščica, zgrajena okoli mikrokontrolerja ESP8266, ki ponuja 11 nožic, tipko za ponovni zagon oz. *RESET* in *MicroUSB* vmesnik za serijsko povezavo in napajanje [44]. Je poceni razvojna ploščica, kar jo naredi mamljivo za začetnike programiranja mikrokontrolerjev, saj v primeru okvare ploščice menjava ni predraga [48].

Mikrokotnrolerje *ESP8266* je možno programirati neposredno iz računalnika z uporabo razvojnih orodij podjetja *Espressif Systems* ali pa z drugimi razvojnimi orodji, kot je naprimer odprtokodni *Arduino IDE*. Kot medij za nalaganje programov se privzeto uporabi serijska povezava, pri *WEMOS D1 mini*je to serijska povezava preko *USB* kabla. Preko iste povezave poteka tudi komunikacija med ploščico in razvojnim računalnikom oziroma med progra-

mom, ki teče na njej, in programerjem.

Serijska povezava s kablom pa ima nekaj slabosti:

1. počasno nalaganje programov,
2. potreba po fizični prisotnosti razvojnega računalnika v dosegu *USB* kabla,
3. možnost nenadne prekinitve nalaganja programa zaradi nestabilnosti gonilnika, slabe povezave med razvojno ploščico in razvojnim računalnikom [3].

Slabost številka 2 nas ovira glede mobilnosti oziroma dostopnosti ploščice, ki si jo lahko privoščimo, slabosti 1 in 3 pa nas ovirata pri izkoristku časa.

Ker ima ploščica *WEMOS D1 mini*vgrajeno *WiFi* anteno, lahko koristimo nalaganje programov preko zraka (*ang. Over The Air — OTA*) [40]. V razvojnem okolju *Arduino IDE* je implementacija že prisotna [40]. *OTA* omogoča hitrejše nalaganje programov in izniči potrebo po fizični bližini razvojnega računalnika. Nalaganje poteka preko interneta, zato potrebujemo samo eno brezžično dostopno točko, na katero se ploščica lahko poveže.

Glavna slabost *OTA* pa je ta, da se brez žične serijske povezave izgubi možnost komunikacije med programom na ploščici in programerjem, prav tako *Arduino IDE* ne sprejema komunikacije s ploščic, ki so povezane preko interneta. To komunikacijo mora programer ali realizirati sam ali pridobiti knjižnico od drugih, kar pa pomeni, da se za začetnike uporaba *OTA* ne zdi primerna.

Želimo si, da bi *OTA* in *Arduino IDE* dopolnili tako, da omogočata komunikacijo med programom na ploščici in programerjem tudi preko interneta, z uporabo WiFi. Tako bi omogočili programerjem, še najbolj pa začetnikom, lažje brezžično programiranje *WEMOS D1 mini*, drugih razvojnih ploščic z mikrokontrolerjem ESP8266 in potencialno še drugih mikrokontrolerjev.

To besedilo opisuje, kako smo v razvojnem okolju *Arduino IDE* realizirali *C++* knjižnico, ki simulira serijsko povezavo preko brezžične povezave in

kako smo s pomočjo razvojnega okolja *Eclipse IDE* spremenili razvojno okolje *Arduino IDE* tako, da to komunikacijo programerju omogoča. V poglavju 2 bomo podrobneje opisali razvojno ploščico *WEMOS D1 mini* ter razvojni okolji *Arduino IDE* in *Eclipse IDE*, v poglavju 3 bomo opisali, kako smo realizirali knjižnico za simulacijo serijske povezave preko brezžičnega interneta, v poglavju 4 bomo opisali, kako smo spremenili razvojno okolje *Arduino IDE*, da prej omenjeno komunikacijo podpira, v poglavju 5 bomo opisali potrebne nastavitve požarnega zidu, v poglavju 6 pa bomo opisali primer uporabe izdelka diplomske naloge. Na koncu pa bomo v sklepu povzeli celoten proces, opozorili na težave, na katere smo naleteli, in podali možne izboljšave rešitve.

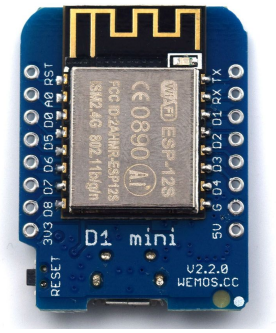
Za izdelavo diplomske naloge je bilo potrebno znanje programskih jezikov *C++* in *Java*, praktični del diplomske naloge pa je bil sprogramiran na Linux distribuciji OpenSUSE.

Poglavje 2

Opis uporabljenih orodij

2.1 Razvojna ploščica WEMOS D1 mini

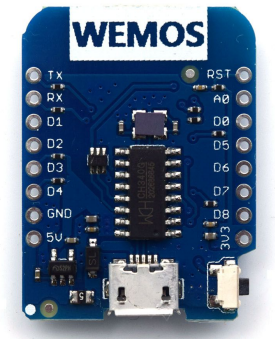
Razvojna ploščica *WEMOS D1 mini* temelji na mikrokontrolerju *ESP8266EX*. Dolga je 34.2 mm, široka 25.6 mm in težka 10 g [44]. Vanj je že tovarniško vgrajen modul *WiFi ESP-12S*, ki je viden na sliki 2.1. Na isti sliki so vidne



Slika 2.1: Razvojna ploščica WEMOS D1 mini. Vir: [44]

.

luknjice za nožice, lučka LED in *RESET* tipka. Na sliki 2.2, ki prikazuje drugo stran ploščice, pa je viden *MicroUSB* vhod, ki služi kot priključek za napajanje in za serijsko povezavo z računalnikom. Ostale tehnične lastnosti, povzete po [44], so:



Slika 2.2: Razvojna ploščica WEMOS D1 mini, druga stran. Vir: [44]

.

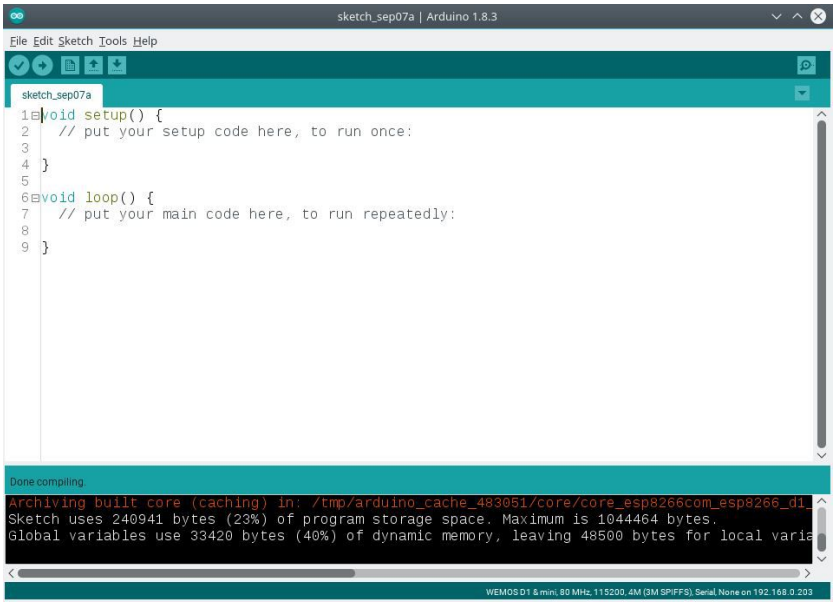
- **Hitrost procesorske ure:** 80 MHz ali 160 MHz
- **Flash pomnilnik:** 4 MB
- **Delovna napetost:** 3.3 V

Za uspešno poezovanje mikrokontrolerja z operacijskima sistemoma Windows in Mac OSX je potrebna namestitev gonilnika *USB-SERIAL CH341*, operacijski sistemi Linux pa tega gonilnika ne potrebujejo [45, 46, 47].

Programe lahko na razvojno ploščico naložimo z orodjema NodeMCU[47] ali Arduino IDE [44, 46]. Pri izdelavi diplomske naloge smo uporabili slednjega.

2.2 Arduino IDE

Arduino IDE (angl. *Integrated Developement Evironment*) je odprtokodno razvojno okolje za pisanje programov, ki se bodo izvajali na raznih mikrokontrolerjih [11, 18].



Slika 2.3: Zaslonska slika okolja Arduino. Za urejanje je odprta prazna *skica*.

2.2.1 Grafični vmesnik

Na sliki 2.3 je prikazan grafični vmesnik omenjenega razvojnega okolja. Črno polje na dnu okna prikazuje podatke o prevajanju in nalaganju programov, v primeru napake pa izpiše, kaj je šlo narobe.

Belo polje v sredini je urejevalec besedila, znotraj katerega se piše program. Urejevalnik besedila omogoča prikaz številk vrstic, zlaganje (angl. folding) kode, barvanje kode, avtomatsko formatiranje kode in še druge funkcije.

Nad urejevalnikom besedila pa imamo gumbe v zeleni barvi. Najbolj pomembni gumbi, z leve strani, so prvi gumb, *verify*, ki odprto kodo prevede, drugi gumb, *upload*, ki kodo prevede in jo naloži na mikrokontroler, in zadnji gumb, čisto na desni strani okna, ki odpre *serijsko konzolo* [16].

Čisto na vrhu pa je menijska vrstica, ki omogoča odpiranje novih *skic*, urejanje nastavitev Arduino IDE itd.

2.2.2 Python

Za delovanje okolja Arduino IDE je potrebna tudi predhodna inštalacija okolja Python 2.3 [46].

2.2.3 Prevajanje kode

Razvojno okolje omogoča urejanje kode ter prevajanje in nalaganje te kode na mikrokontrolerje.

Po [6] lahko poenostavimo proces nalaganja kode:

1. predprocesiranje,
2. prevajanje C++ kode v strojno kodo oz objektne datoteke,
3. povezovanje kode s standadnimi Arduino knjižnicami v enotno „hex“ datoteko,
4. nalaganje „hex“ datoteke na mikrokontroler.

V stopnji prevajanja se prevajalnik ravna po navodilih za prevajanje kode glede na mikrokontroler, na katerega bomo kodo naložili [6].

2.2.4 Skice

Koda, ki se ureja v tem razvojnem okolju, se shrani v *skico* (angl. sketch) s končnico *.ino*. Ime *skica* prihaja iz programa *Processing*, ki služi kot ena od osnov Arduino IDE, ter pomeni *program* oz. *projekt* [41, 11, 14]. Vsaka *skica* se shrani v svojo mapo na disku [41].

Poleg *skic* lahko v razvojnem okolju ustvarimo in urejamo tudi datoteke *C++* z izvorno kodo in datoteke z definicijami (angl. header file)[15]. Vse dodatne datoteke se shranijo v isto mapo, kjer je shranjena *skica*, in se avtomatično naložijo v *Arduino IDE*, če *skico* odpremo v njem. V grafičnem vmesniku razvojnega okolja so dodatne datoteke prikazane kot zavihki urejevalnika besedila.

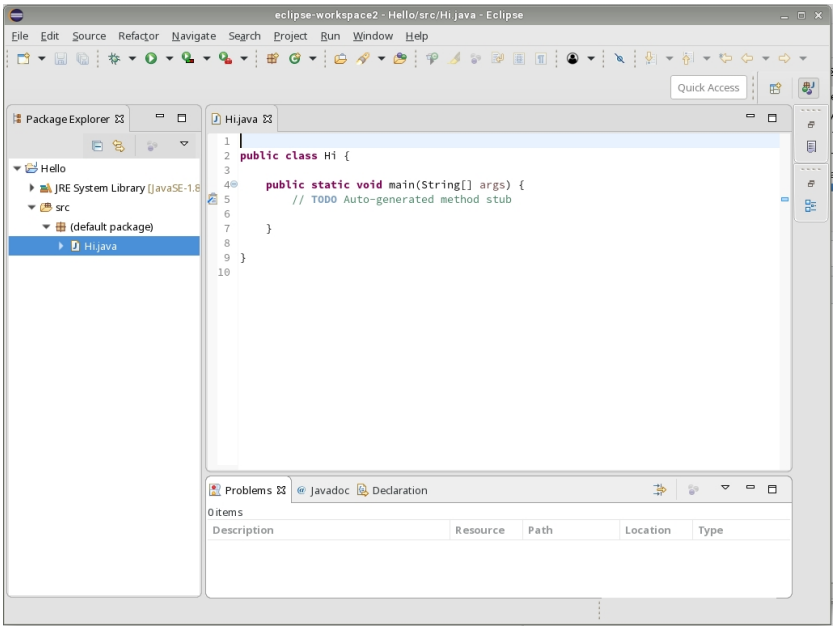
Zanimivost *skic* je, da programer vidi le del celotnega programa. V primeru Arduino IDE sta to funkciji *void setup()* in *void loop()*, kot je tudi prikazano na sliki 2.3. Funkcija *setup()* se kliče ob zagonu mikrokontrolerja, funkcija *loop()* pa se izvaja v neskončni zanki [17].

2.2.5 Odprtokodnost

Izvorna koda razvojnega okolja je na voljo na povezavi [11]. Ker ima odprtnokodno licenco, lahko vsakdo prenese izvorno kodo in jo po svoje uredi, spremenjeno različico pa lahko ponuja naprej, vendar pod isto licenco [8].

2.3 Eclipse IDE

Eclipse IDE je najbolj poznano kot *Javansko* integrirano razvojno okolje [30], vendar je z uporabo raznih dodatkov v njem možno razvijati v veliko drugih programskih jezikih [30, 31, 19].



Slika 2.4: Zaslonska slika okolja *Eclipse IDE*.

Poleg urejanja kode in njenega prevajanja *Eclipse* ponuja še mnogo drugih posebnosti, med katerimi so tudi povezovanje z reporzitoriji *Git*, prikaz večih tekstovnih datotek stran ob strani, zaznavanje napak ob pisanju kode itn.

Razvojno okolje izdaja *The Eclipse Foundation* pod odprtokodno licenco *Eclipse Public License* [19, 43].

Pri izdelavi diplomske naloge smo uporabili verzijo *Eclipse Oxygen*, ki je prikazana na sliki 2.4.

2.3.1 Orodje Ant

Apache Ant (*Another Neat Tool*) je orodje za avtomatizacijo processa povezovanja in prevajanja pogramske kode. Orodje je bilo razvito leta 2000 kot namestnik tedanjega orodja *make* [1, 27]. Spisano je v programskem jeziku *Java* in je namenjeno predvsem za uporabo na projektih *Java*[27].

Za razliko od orodij, ki so omejene na ukaze sistemske lupine, kot je na primer *make*, so ukazi oziroma *tarče* (angl. *Targets*) sestavljeni iz *nalog* (angl. *tasks*), ki so definirane kot javanski objekti, kar pomeni, da je *Ant* prenosljiv na vse operacijske sisteme, ki imajo podporo za javansko okolje [27].

Vse te *naloge* so javanski razredi, ki dedujejo razred *org.apache.tools.ant.Task* [29]. V primeru, da potrebovana *naloga* ni prisotna v *Ant*, jo lahko programer spiše sam [28, 29].

Tarče je potrebno definirati na nivoju vsakega projekta. Definira se jih v datoteki *build.xml*, kjer se z *značkami* definira *tarče*, njihove odvisnosti (angl. *dependencies*) ter katere *naloge* se bodo izvedle.

V sklopu naše diplomske naloge je v reporzitoriju izvorne kode *Arduino IDE* že prisotna datoteka *build.xml* [12], zato bo proces prevajanje projekta avtomatiziran s *tarčami*, ki so v tej datoteki že definirane.

Poglavje 3

Implementacija C++ knjižnjice *OTASerial*

3.1 Predpriprava

Najprej smo prenesli okolje Arduino IDE, ki je dostopno na [18]. Nato smo v mapo skic prenesli *jedro ESP8266/Arduino* [26] za *ESP8266* mikrokontrolerje, kot je opisano v [46]. To nam omogoči, da v okolju Arduino IDE izberemo *WEMOS D1 mini* kot tarčo za prevajanje kode.

ESP8266/Arduino jedro vsebuje prilagojene *Arduino* knjižnice, ki delujejo tudi na *ESP8266* mikrokontrolerjih, hkrati pa vsebuje knjižnice, ki so namenjene uporabi izključno za ta mikrokontroler.

3.2 Datoteke z izvorno kodo

V Arduino IDE smo odprli novo skico in v njeni mapi naredili datoteko z definicijo in datoteko z izvorno kodo. Ker smo knjižnico poimenovali *OTASerial*, smo ti dve datoteki poimenovali *OTASerial.h* in *OTASerial.cpp*.

V *OTASerial.h* bodo deklaracije razreda *OTASerial* in njegovih metod ter definicije njegovih krajših metod, v *OTASerial.cpp* pa bodo deklaracije ostalih metod razreda.

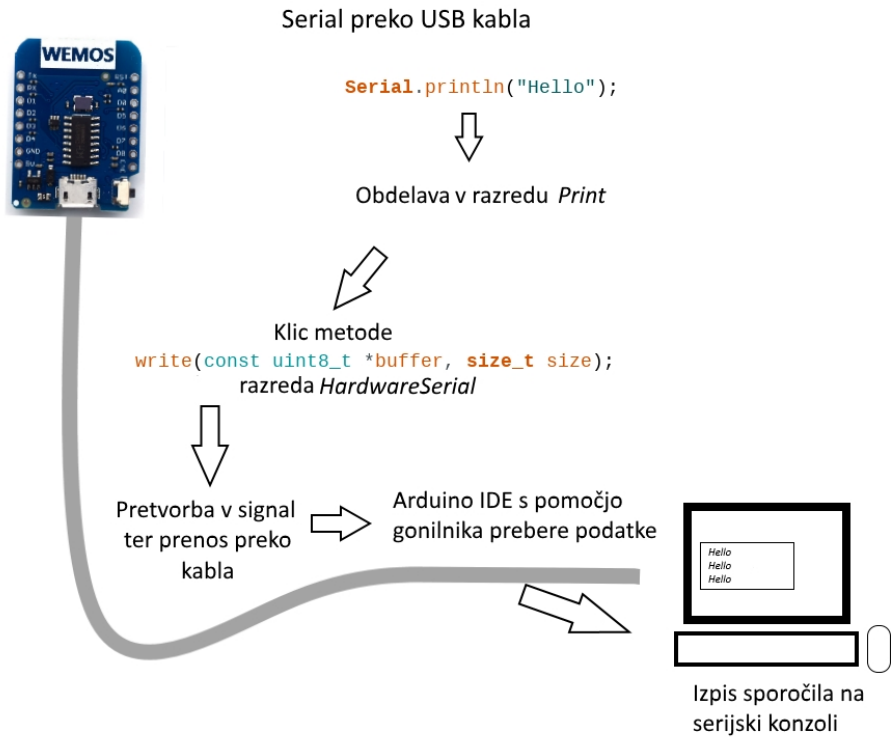
3.3 Pisanje na *serijsko konzolo*

Ko programer želi na *serijsko konzolo* poslati neke podatke, na primer niz *"Hello"*, v kodi to zapiše kot:

```
Serial.println("Hello");
```

3.3.1 Privzeto delovanje

Če si pogledamo izvorno kodo razreda *HardwareSerial*, katerega tip je objekt *Serial*, vidimo, da ta deduje razred *Stream*, ki pa deduje razred *Print* [22].



Slika 3.1: Povzetek potrebnih korakov za izpis sporočila na *serijski konzoli* brez knjižnice *OTASerial*.

Metode *print()*, *printf()* in *println()* so prvič definirane v razredu *Print*,

to je deklarirane v *Print.h* in definirane v *Print.cpp*[24, 23]. Te metode so definirane za izpisovanje vseh možnih podatkovnih tipov, ki se pojavijo v *Arduino* knjižnicah [24, 23].

Vsem tem metodam je skupno to, da na koncu kličejo virtualni metodi *write(uint8_t)* in *write(uint8_t buffer, size_t size)* [24, 23]. Metoda *write(uint8_t)* je nastavljena na 0 [24], kar pomeni, da smo jo prisiljeni implementirati, ko razred *Print* dedujemo.

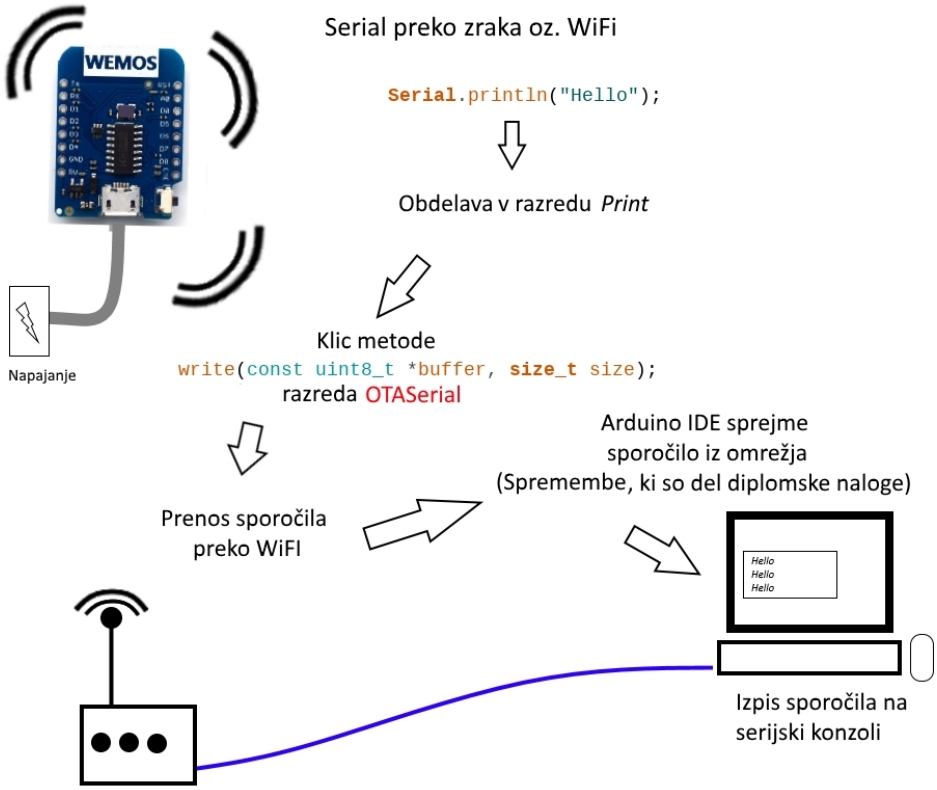
Metoda *write(uint8_t *buffer, size_t size)* pa je že implementirana in v zanki *while* kliče metodo *write(uint8_t)*, dokler *size* ne doseže 0 [24]. V implementaciji metode *write(uint8_t)* v razredu *HardwareSerial* se črka tipa *uint8_t* zapiše v pisalni medpomnilnik preko metode *uart_write_char(uart_t* uart, char c)*, ki je deklarirana v datoteki *uart.h* [24, 21, 25]. Na sliki 3.1 je skiciran povzetek postopka za izpis sporočila na serijski konzoli, kot smo ga opisali v tem razdelku.

3.3.2 Z uporabo knjižnice *OTASerial*

V naši knižnjici spremenimo način prenosa sporočila tako, da se prenese preko *WiFi*, nato pa preko lokalnega omrežja do razvojnega računalnika, kjer se prikažejo na *serijski konzoli* okolja *Aruino IDE*.

Glavni razred knjižnice je istoimenski razred *OTASerial*. Znotraj tega razreda smo redefinirali metodi *write(uint8_t)* in *write(uint8_t buffer, size_t size)* tako, da sporočilo pošljeta preko omrežja.

Povzetek postopka za izpis sporočila na serijski konzoli z uporabo knjižnice *OTASerial* je skiciran na sliki 3.2, na sliki 3.3 pa je omenjena sprememba prikazana v obliki protokolskih diagramov. Na diagramih so prikazani vsi protokoli, potrebni za dosego prenosa sporočila z razvojne ploščice do razvojnega računalnika.



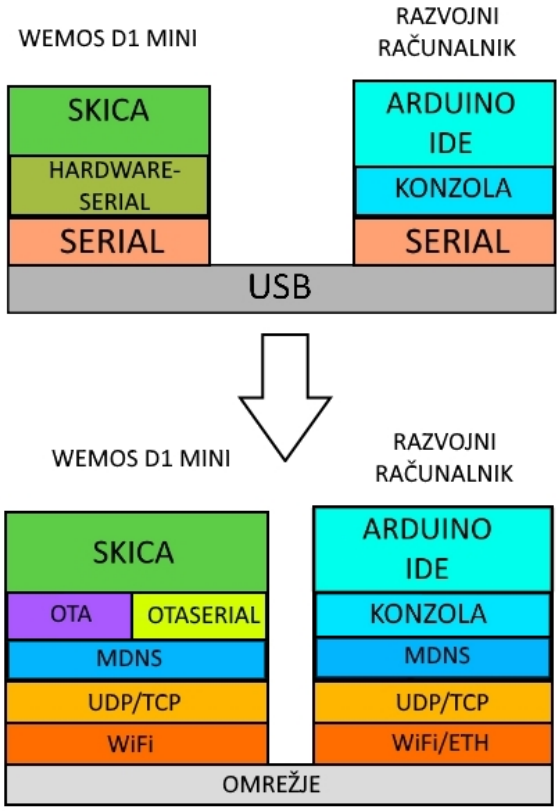
Slika 3.2: Povzetek potrebnih korakov za izpis sporočila na *serijski konzoli* z uporabo knjižnice *OTASerial*.

3.4 Razred *OTASerial*

Razred *OTASerial* je glavni razred naše knjižnice. Tako kot razred *HardwareSerial*, ki je razred objekta *Serial*, *OTASerial* deduje razred *Stream*, ki nam poleg pisalnih metod (metode *print()*) ponuja tudi metode za branje oziroma metode *read()*.

Ker je objekt *Serial* deklariran na skladu, ga naknadno ne moremo uničiti [22]. Zato smo izbrali ekstremno pot in smo na koncu datoteke *OTASerial.h* definirali *makro*:

```
#define Serial OTASerial_object
```



Slika 3.3: Protokolski diagrami, ki prikazujejo proces pošiljanja podatkov z razvojne ploščice do *Arduino IDE* pred in po naših spremembah.

Ta *makro* bo v času predprocesiranja kode vse omembe *Serial* v kodi zamenjal z *OTASerial_object*. To pomeni, da bo vrstica

```
Serial.println("Hello");
```

po predprocesiranju izgledala takole:

```
OTASerial_object.println("Hello");
```

Objekt *OTASerial_object* je objekt razreda *OTASerial*, ki ga deklariramo in definiramo znotraj naše knjižnice, tako da ga programerju ni potrebno dodatno deklarirati ob uporabi naše knjižnice. Ker kot argument v konstruktorju zahteva objekt *Serial* razreda *HardwareSerial*, smo na začetku datoteke *OTASerial.cpp* dodali makro:

```
#undefine Serial
```

Ta makro znotraj datoteke *OTASerial.cpp* omogoča uporabo originalnega objekta *Serial*.

3.4.1 Glavni gradniki razreda

Glavni gradniki razreda *OTASerial* so prikazani na izseku kode 3.1.

Zasebni gradniki (angl. ***private***)

- *debugServer*: kazalec na objekt razreda *WiFiServer*. Z uporabo tega objekta se postavi strežnik, na katerega se lahko poveže *serijska konzola* okolja *Arduino IDE*.

- *client*: objekt razreda *WiFiClient*. Ta objekt predstavlja povezavo med *odjemalcem*, ki je povezan prek *Arduino IDE* in med strežnikom, ki teče na razvojni ploščici.

- *_lastClientCheck:* nepredznačeno celo število tipa *long*, vrednost katerega prikazije čas zadnje preverbe prisotnosti novega odjemalca na strežniku. Čas je prikazan v milisekundah.

- *_connected*: logična zastavica, ki se uporabi za preverbo povezanosti *odjemalca* pri klicu metod, ki so del razreda *OTASerial*.

- *_initialised*: logična zastavica, ki se nastavi ob inicilizaciji *OTASerial* v metodi *begin()*. Uporablja se za preverjanje, ali je *OTASerial* že iniciliziran.

- *_ESSID*: Kazalec na niz črk, ki vsebuje ime brezžičnega omrežja. Uporabi se pri povezovanju na brezžično dostopno točko.

- *_password*: Kazalec na niz črk, ki vsebuje geslo brezžičnega omrežja. Uporabi se pri povezovanju na brezžično dostopno točko.

```
private:
  WiFiServer *debugServer = nullptr;
  WiFiClient client;
  unsigned long lastClientCheck = 0;
  bool _connected = false;
  bool _initialised = false;
  char * ESSID =  nullptr;
  char * password = nullptr;

public:
  HardwareSerial *USBSerial = nullptr;
  bool OTADefined = false;
  bool WiFiDefined = false;
  bool internalOTAHandle = true;
  unsigned long clientCheckInterval = 1000;
  unsigned int OTAPort = 8266;
  unsigned int debugPort = 23;
```

Izsek kode 3.1: Glavni gradniki razreda *OTASerial*.

Ker sta kazalca na niza *ESSID* in *password* zasebna, jih programer ne more neposredno nastaviti. Za ta namen je na voljo javna metoda *configWiFi(const char * essid , const char * pass)*, v kateri se vrednosti *essid* in *pass* prekopirata v zasebna niza *ESSID* in *password*.

Javni gradniki (angl. *public*)

- *USBserial*: kazalec na objekt Serial razreda *HardwareSerial*. Povezave do tega objekta ne želimo izgubiti, zato shranimo kazalec, ki kaže na njegov naslov.

- *OTADefined*: logična zastavica, ki se preveri ob inicilizaciji objekta *OTASerial* in nam pove, ali je programer objekt *ArduinoOTA* inicializiral sam. To zastavico mora programer nastaviti sam, preden kliče metodo *begin()* .

- *WiFiDefined*: logična zastavica, ki se preveri ob inicilizaciji objekta *OTASerial* in nam pove, ali je programer objekt *WiFi* inicilizira sam. To zastavico mora programer nastaviti sam, preden kliče metodo *begin()*.
- *internalOTAHandle*: logična zastavica, ki označuje, ali naj se znotraj metode *write()* kliče metoda *ArduinoOTA.hanlde()*.
- *clientCheckInterval*: nepredznačeno celo število tipa *long*, ki označuje časovni razmak med preverbami prisotnosti novega odjemalca na strežniku.
- *OTAPort*: nepredznačeno število, ki hrani številko vrat, na katerih bo dostopna storitev *ArduinoOTA*. To število je privzeto nastavljeno na *8266*, kot je tudi privzeta nastavitev ob inicilizaciji *ArduinoOTA*, programer pa ga lahko nastavi na drugo vrednost.
- *debugPort*: nepredznačeno število, ki hrani številko vrat strežnika *debugServer*. Privzeta vrenost je 23, kar so standardna vrata za *telnet*. To število lahko programer spremeni na drugo vrednost.

3.4.2 Metode *begin()* in *end()*

V svetu *Arduina* obstaja navada, da se znotraj konstruktorjev izvaja samo nastavljanje vrednosti, ostala inicilizacija objektov pa se izvede v metodah *begin()*[2]. Razred ima lahko več teh metod, ki pa lahko sprejmejo različne argumente.

Prav tako pa je ustavljanje delovanja objektov naloga metod *end()* in ne destruktorjev [2].

Metode *begin()*

Ker s knjižnico *OTASerial* želimo zamenjati objekt *Serial*, smo iz izvorne kode razreda *HardwareSerial.h* prepisali deklaracije metod *begin()*. Kot je razvidno iz izseka kode 3.2, so prve tri metode že definirane in vse kličejo zadnjo, še ne definirano metodo.

```
virtual void begin(uLong baud){
  begin(baud, SERIAL_8N1, SERIAL_FULL, 1);
}

virtual void begin(uLong baud, SerialConfig config) {
  begin(baud, config, SERIAL_FULL, 1);
}

virtual void begin(uLong baud, SerialConfig config, SerialMode
    mode) {
  begin(baud, config, mode, 1);
}

virtual void begin(uLong baud, SerialConfig config, SerialMode
    mode, uint8_t tx_pin);
```

Izsek kode 3.2: Ponovna deklaracija metod *begin()*, ki so deklarirane že v *HardwareSerial.h*

Prva polovica definicije četrte metode je prikazana na izseku kode 3.3, druga polvica pa na izseku kode 3.4. Najprej se s preverbo zastavice *_initialised* preveri, če je bil objekt *OTASerial* že iniciliziran. V primeru, da je bil, se izvajanje te metode z ukazom *return* preneha. Če objekt še ni bil iniciliziran, se izvajanje nadaljuje.

Ker ne želimo dokončno onemogočiti objekta *Serial* razreda *HardwareSerial*, ga v naslednjem koraku s pomočjo kazalca *USBserial* inicializiramo z uporabo njegove metode *begin()*. Kot argumente uporabimo argumente naše metode *begin()*.

V naslednjih korakih se inicializirata *WiFi* objekt, ki nadzoruje povezavo z brezžično dostopno točko in pa objekt *ArduinoOTA*, ki upravlja z nalaganjem programov preko *WiFi*. V primeru, da ima programer ta dva objekta že inicilizirana, lahko ta dva koraka preskoči z nastavitvijo zastavic *Serial.WiFiDefined* in *Serial.OTADefined*.

Za delovanje knjižnjice *OTASerial* pa je zahtevano, da sta oba objekta

```
void OTASerial::begin(uLong baud, SerialConfig config,
    SerialMode mode, uint8_t tx_pin) {
  if(_initialised) return;
  USBSerial->begin( baud, config, mode, tx_pin);
  if (!WiFiDefined) { ... }
  if (!OTADefined) { ... }
  debugServer = new WiFiServer(debugPort);
  debugServer->begin();
}
```

Izsek kode 3.3: Inicilizacija objektov *USBserial*, *WiFi*, *ArduinoOTA* in *debugServer*.

inicilizirana.

V naslednjem koraku se inicilizirata strežnik *debugServer*, ki začne poslušati na vratih *debugPort*.

Od tu naprej je postopek prikazan na izseku kode 3.4. K *MDNS* storitvi, ki jo predstavlja objekt *MDNS*, se dodata zapisa *"OTA_Serial_Port"*, ki hrani vrata strežnika *debugServer*, in *"OTA_Serial"*, ki z vrednostjo *"yes"* oznanja, da je na tej razvojni ploščici aktivirana knjižnica *OTASerial*.

V predzadnjem koraku se na *serijsko konzolo* izpišejo podatki o povezavi na brezžično dostopno točko in podatki o vratih strežnika *debugServer* ter o vratih, na katerih posluša *ArduinoOTA*.

Nazadnje se zastavica *_initialised* nastavi na *true*, kar pomeni, da je objekt *OTASerial* inicializiran.

Inicilizacija *WiFi*

Sledi bolj podroben opis inicilizacije objekta *WiFi* znotraj metode *begin()*.

Najprej se s klicem metode:

```
WiFi.mode(WIFI_STA);
```

nastavi delovanje *WiFi* modula v način *postaje* oziroma *WIFI_STA*. Ta način pomeni, da bo razvojna ploščica delovala kot naprava, ki se želi povezati v

```
  MDNS.addServiceTxt("arduino" , "tcp", "OTA_Serial_port",
    String(debugPort));
  MDNS.addServiceTxt("arduino" , "tcp", "OTA_Serial", "yes");

  USBSerial->println();
  USBSerial->print("WiFi connected to:");
  USBSerial->print(WiFi.SSID());
  USBSerial->print(" at IP: ");
  USBSerial->println(WiFi.localIP());
  USBSerial->print("OTA listening at port ");
  USBSerial->println(OTAPort);
  USBSerial->print("Server listening at port ");
  USBSerial->println(debugPort);
  _initialised = true;
}
```

Izsek kode 3.4: Dodajanje podatkov knjižnice v mDNS zapis in izpis informacij o povezanosti v *WiFi* omrežje.

WiFi omrežje.

V naslednjem koraku se preveri, če sta podana ime dostopne točke oz. *ESSID* in geslo za dostop do nje. Če sta *ESSID* in geslo podana, se kliče metoda *begin()* objekta *WiFi*:

```
WiFi.begin(ESSID, password);
```

Če *ESSID* in *geslo* nista podana, se izvede povpraševanje po njih.

Dokler *ESSID* in *geslo* nista podana, se izvaja zanka, v kateri se najprej poizve po prisotnih brezžičnih omrežjih, potem pa se zaporedno v svojih ločenih zankah sprašuje najprej po *ESSID* in nato po *geslu*.

Ko sta podana oba *ESSID* in *geslo*, se s poskusom povezave na dostopno točko preveri, če sta pravilna. Če *ESSID* in *geslo* nista pravilna, se povpraševanje nadaljuje.

Inicilizacija *ArduinoOTA*

Objekt *ArduinoOTA* smo inicializirali enako, kot je inicializiran v skici *BasicOTA* [20]. Za inicilizacijo smo definirali 4 povratne (angl. *callback*) metode s klicem naslednjih metod:

- *onStart(callback()):* funkcija *callback* se kliče ob začetku nalaganja skice preko *OTA*. V naši implementaciji na *serijsko konzolo* izpiše niz o začetku nalaganja nove skice.
- *onProgress(callback(progress, total)):* funkcija *callback* se kliče med potekom nalaganja programa. Kot argumente sprejme nepredznačeni števili *progress* in *total*, ki javljata podatke o poteku nalaganja programa. V naši implementaciji na *serijsko konzolo* napiše, kolikšen del skice je naložen v odstotkih.
- *onEnd(callback()):* funkcija *callback* se kliče, ko je nalaganje končano. V naši implementaciji na *serijsko konzolo* izpiše samo niz "End".
- *onError(callback(error)):* funkcija *callback* se kliče ob napaki. Vrednost argumenta *error* je odvisna od tipa napake. V naši implementaciji preveri vrednost argumenta *error* in v *serijsko konzolo* izpiše, katera napaka se je zgodila.

Nato smo z uporabo metode *setHostname()* spremenili ime ploščice oziroma *hostname* na *esp8266-OTASerial* in z uporabo metode *setPort()* spremenili vrata, na katerih posluša *ArduinoOTA*, na vrednost števila *OTAPort*. *ArduinoOTA* ponuja še nastavitev gesla za nalaganje *skic*, vendar tega nismo izkoristili.

Kot zadnji korak smo objekt *ArduinoOTA* inicilizirali s klicem njegove metode *begin()*.

Multicast *DNS*

Knjižnica *ArduinoOTA* in razvojno okolje *Arduino IDE* za oglaševanje in prepoznavanje razvojnih ploščic v lokalnem omrežju uporabljata protokol

Multicast Domain Name System (MDNS). Ta protokol se uporablja za razreševanje domenskih imen znotraj manjših omrežij brez potrebe po *DNS* strežniku. Standard določa, da se morajo vse *DNS* poizvedbe za domenska imena, ki se končajo z domeno *".local."*, razrešiti preko *MDNS*.

MDNS uporablja pakete, ki temeljijo na paketih *DNS*, pošilja pa se jih na *IPv4* naslov *224.0.0.251* oziroma *IPv6* naslov *FF02::FB* na *UDP* vratih *5353*. Ti paketi se razširijo po celotnem omrežju, zato jih lahko vidijo vse naprave.

Primeri implementacij *MDNS* so *Apple Bonjour*, *Avahi*, knjižnica *CC3000 Multicast DNS* za razvojne plošcice ter *JmDNS*, ki jo uporablja tudi *Arduino IDE*.

Ob zagonu razvojnega okolja *Arduino IDE* se izvede *MDNS* poizvedba za domensko ime *_arduino._tcp.local.* Če je v omrežju prisotna kakšna naprava, ki si je prisvojila iskano domensko ime, ta naprava odgovori z opisom svojih storitev ter s svojim naslovom *IP*.

Na izseku kode 3.5 je prikazan izpis programa *tcpdump* ob zagonu *Arduino IDE*. Na *IPv4* naslovu *192.168.0.108* se je najahalo razvojno okolje *Arduino IDE*, na *IPv4* naslovu *192.168.0.114* pa razvojna ploščica *WEMOS D1 mini*, na kateri je že tekla knjižnica *OTASerial*. Kot dokaz prisotnosti knjižnice sta zapisa *"OTA_Serial_port=23"* in *"OTA_Serial=yes"* v odgovoru na *MDNS* poizvedbo, dodatek katerih je bil omenjen v razdelku 3.4.2.

Metoda *end()*

Naloga metode *end()* je ustaviti delovanje objekta knjižnice. Potek ustavitve delovanja *OTASerial* je prikazan na izseku kode 3.6. V prvem koraku se preveri, ali je *OTASerial* že inicializiran, in če je, se izvajanje metode nadaljuje. Nato se po potrebi prekine komunikacijo z odjemalcem *client*, zastavico *_connected* pa se nastavi na *false*. V naslednjem koraku se z uporabo metode *close()* ustavi delovanje strežnika *debugServer*. V nadaljevanju se z uporabo operatorjev *delete* in *delete []* pobriše kazalce *debugServer*, *ESSID* in *password*, nato pa se vrednost teh kazalcev nastavi na *nullptr*. Nazadnje,

```
IP 192.168.0.108.mdns > 224.0.0.251.mdns:
  0 PTR (QM)? _arduino._tcp.local. (37)

IP 192.168.0.114.mdns > 224.0.0.251.mdns:
  0*- [0q] 4/0/0
  PTR esp8266-OTASerial._arduino._tcp.local.,
  TXT "tcp_check=no" "ssh_upload=no"
  "board=ESP8266_WEMOS_D1MINI" "auth_upload=no"
  "OTA_Serial_port=23" "OTA_Serial=yes",
  (Cache flush) SRV esp8266-otaserial.local.:8266 0 0,
  (Cache flush) A 192.168.0.114 (353)
```

Izsek kode 3.5: Izpis programa *tcpdump* v *bash terminalu* ob zagonu *Arduino IDE*.

```
void OTASerial::end(){
  if(!_initialised) return;
  if(_connected)client.stop();
  _connected = false;
  debugServer->close();
  delete debugServer;
  delete [] ESSID;
  delete [] password;
  ESSID = nullptr;
  password = nullptr;
  debugServer = nullptr;
  _initialised = false;
}
```

Izsek kode 3.6: Definicija metode *end()* razreda *OTASerial*.

```
virtual size_t write(uint8_t c){
  write((uint8_t *)c,1);
}
virtual size_t write(const uint8_t *buffer, size_t size);
```

Izsek kode 3.7: Declaracije *write()* metod v datoteki *OTASerial.h*.

se vrednost zastavice *_initialised* nastavi na *false*.

3.4.3 Metode *write()*

Razred *OTASerial* redefinira metodi *write(uint8_t c)* in *write(uint8_t *buffer, size_t size)*, njuni deklaraciji sta vidni na izseku kode 3.7. Metoda *write(uint8_t c)* je že definirana in kliče metodo *write(uint8_t *buffer, size_t size)*, kot *buffer* uporabi argument *c* tipa *char*, ki ga tretira kot tabelo tipa *uint8_t* dolžine 1.

Definicija metode *write(uint8_t *buffer, size_t size)* pa je prikazana na izseku kode 3.8.

Najprej se preveri vrednost zastavice *_initialised*, ki pove ali je *OTASerial* iniciliziran. Če je vrednost zastavice enaka *false* se izvajanje metode zaključi z vrnitvijo števila 0. V naslednjem koraku se v primeru, da objekt *ArduinoOTA* ni iniciliziran izven knjižnice in da je vrednost zastavice *internalOTAHandle* enaka *true*, izvede njegova metoda *handle()*. Ta metoda preverja za začetek nalaganja nove skice preko *OTA* in mora biti prisotna nekje v skici, če *OTA* želimo izkoristiti. Nato se kliče zasebna metoda *checkForClient()*, ki preveri prisotnost odjemalca, če ta ni povezan, v nasprotnem primeru pa nove odjemalce zavrne.

Nato sledi preverjanje odjemalčevega stanja s klicem metode *status()*. Če je odjemalčevo stanje enako 0, odjemalec ni več povezan na strežnik. V tem primeru se komunikacija z njim s klicem metode *stop()* ustavi, vrednost zastavice *_connected* pa se nastavi na *false*.

V primeru, da je odjemalec povezan, metoda vrne število poslanih bajtov, ki ga vrne metoda *write(buffer, size)*. S klicem te metode se odjemalcu pošlje

```
size_t OTASerial::write(const uint8_t *buffer, size_t size) {
  if(!_initialised) return 0;
  if(!OTADefined ) ArduinoOTA.handle();
  checkForClient();
  if (!_connected) return 0;

  if(oneClient.status() == 0){
    oneClient.stop();
    _connected = false;
    return 0;
  }
  return oneClient.write( buffer, size);
}
```

Izsek kode 3.8: Definicija metode *write(uint8_t *buffer, size_t size)* v datoteki *OTASerial.cpp*.

besedilo *buffer* velikosti *size*. Na odjemalčevi strani pa naj bi se besedilo *buffer* izpisalo na *konzoli OTA*.

Preverjanje prisotnosti novega odjemalca

Preverjanje prisotnosti novega odjemalca se izvaja znotraj metode *checkForClient()*, ki je prikazana na izseku kode 3.9.

Najprej se preveri ali je od zadnjega preverjanja prisotnosti odjemalca, ki ga označuje spremenljivka *lastClientCheck*, že preteklo *clientCheckInterval* milisekund. V primeru da to drži, se najprej v spremenljivko *lastClientCheck* z uporabo funkcije *millis()* shrani trenutno število milisekund od zagona razvojne ploščice.

Nato pa se s klicem metode *hasClient()* objekta na kazalcu *debugServer* preveri, če je strežnik zaznal novega odjemalca in se ga s klicem metode *available()* shrani v lokalni objekt *newComer* razreda *WiFiClient*.

V primeru, da je na strežnik že povezan odjemalec, kar sporoča zastavica *_connected*, se odjemalcu *newComer* pošlje niz:

```
void OTASerial::checkForClient() {
  if (lastClientCheck > 0 && (millis() - lastClientCheck) <
   clientCheckInterval) return;
  lastClientCheck = millis();
  if (debugServer->hasClient()) {
    WiFiClient newComer = debugServer->available();
    if(_connected){
      newComer.print("FULL:");
      newComer.print(newComer.remoteIP());
      newComer.println();
      newComer.stop();
      return;
    }
    newComer.println("HELLO:OTASERIAL");
    char *response = nullptr;
    size_t len = readClient(newComer,response);
    if (strcmp(response, "HELLO:ARDUINO:OTAMONITOR\n") != 0) {
      delete [] response;
      newComer.stop();
      return;
    }
    delete [] response;
    client = newComer;
    client.println("RESPONSE:OK");
    _connected = true;
  }

}
```

Izsek kode 3.9: Definicija metode *checkForClient()*.

```
"FULL:<IP že povezanega odjemalca>"
```

potem pa se komunikacija z njim konča, izvajanje metode *checkForClient()* pa se s klicem *return* konča.

Če pa na strežniku še ni povezanega odjemalca, se prične zelo preprosta avtentikacija, ki ni namenjena varnosti, ampak omejevanju povezovanja samo na tiste, ki poznajo postopek. Odjemalcu *newComer* se pošlje niz *"HELLO:OTASERIAL"*, ki mu sledi znak za novo vrstico. Če odjemalec ne odgovori z nizom *"HELLO:ARDUINO:OTAMONITOR"*, ki mu sledi znak za novo vrstico, se komunikacija z njim prekine, izvajanje metode pa se konča. V primeru, da je odgovorni niz pravilen, se odjemalec *newComer* shrani kot odjemalec *client*, nanj pa se pošlje niz *"RESPONSE:OK"*, ki mu sledi znak za novo vrstico.

Na koncu se vrednost zastavice *_connected* nastavi na *true*. Sedaj je povezovanje odjemalca končano.

3.4.4 Ostale redefinirane metode

Nadrazred *Stream* nas je prisilil še k redefiniciji metod *available()*, *read()*, *peak()* in *flush()*. Kot je razvidno iz definicij, prikazanih na izseku kode 3.10, te metode za dosego rezultata uporabijo istoimenske metode objekta *client*. Za vse te metode smo dodali še preverjanje, ali je *OTASerial* inicializiran in ali je odjemalec *client* povezan, kar zaznamujeta zastavici *_initialised* in *_connected*, hkrati pa te metode sprožijo preverbo za prisotnost novega odjemalca Če je rezultat preverjanja enak *false*, se izvajanje metode prekine.

Metoda *available()* vrne število *bajtov*, ki so na voljo za branje iz bralnega medpomnilnika. Metoda *peek()* vrne vrednost trenutnega prvega *bajta* v bralnem medpomnilniku. Metoda *read()* vrne vrednost trenutnega prvega *bajta* v bralnem medpomnilniku in ga iz njega odstrani. Metoda *flush()* pa sproži čakanje, dokler se pošiljanje *bajtov* z razvojne ploščice do odjemalca ne konča.

```
int OTASerial::available() {
  if(!_initialised || !(_connected || checkForClient())) return
    -1;
  return client.available();
}

int OTASerial::peek() {
  if(!_initialised || !(_connected || checkForClient())) return
    -1;
  return client.peek();
};

int OTASerial::read() {
  if(!_initialised || !(_connected || checkForClient())) return
    -1;
  return client.read();
};

void OTASerial::flush() {
  if(!_initialised || !(_connected || checkForClient())) return;
  client.flush();
};
```

Izsek kode 3.10: Definicije metod *available()*, *read()*, *peak()* in *flush()*.

Poglavje 4

Urejanje Arduino IDE za podporo brezžične serijske komunikacije

4.1 Predpriprava

Za urejanje in uspešno prevajanje izvorne kode okolja *Arduino IDE* na sistemih *Linux* potrebujemo naslednje pakete [13]:

```
git
make
gcc
ant
openjdk-8-jdk
```

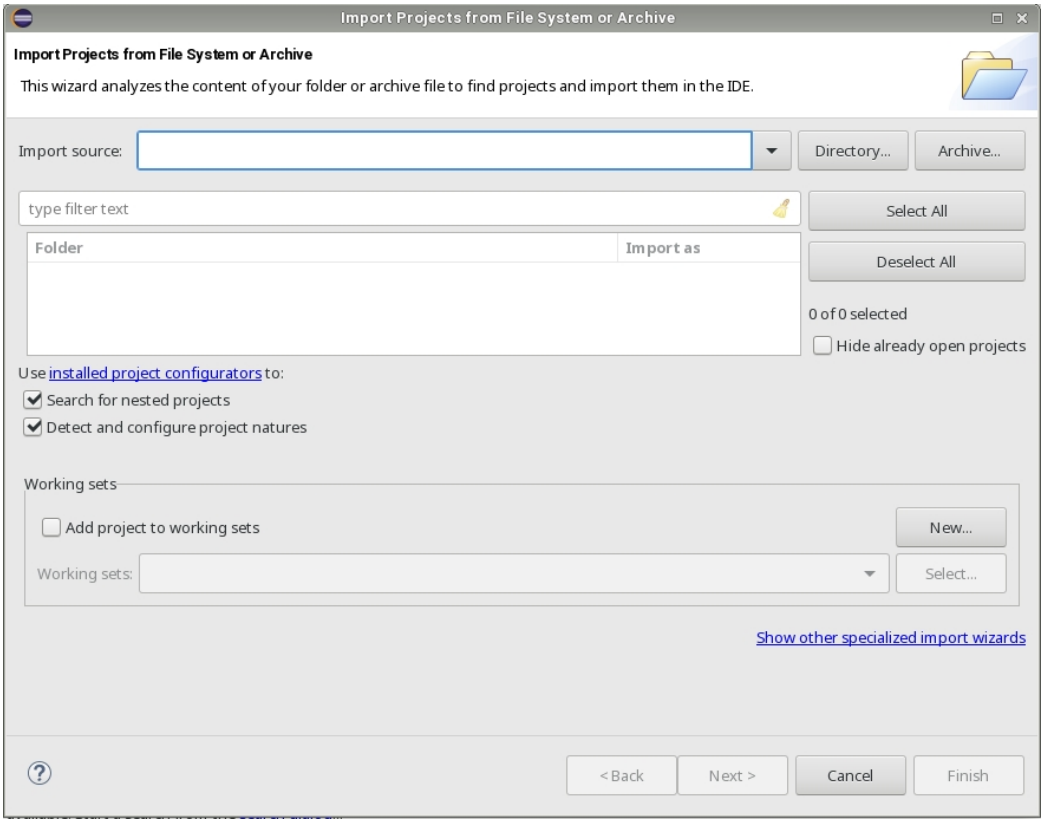
Ker pa je izvorne kode veliko in smo med razvojem morali iskati posamezne datoteke, smo namestili tudi razvojno okolje *Eclipse IDE*.

Ob zagonu *Eclipse IDE* smo si izbrali ustrezen *delovni prostor* (angl. *workspace*). Nato smo z uporabo ukaza *git clone* z *GitHub* reporsitorija [11] v *delovni prostor* prenesli izvorno kodo razvojnega okolja *Arduino IDE*.

V naslednjem koraku smo izvorno kodo *Arduino IDE* vnesli v *Eclipse IDE* kot nov, že obstoječi projekt. V menijski vrstici smo izbrali:

`File > Open Projects from File System`

Pojavilo se je okno, prikazano na sliki 4.1, v katerem smo s klikom na gumb *Directory* izbrali mapo *Arduino* znotraj *delovnega prostora* in potrdili izbiro.



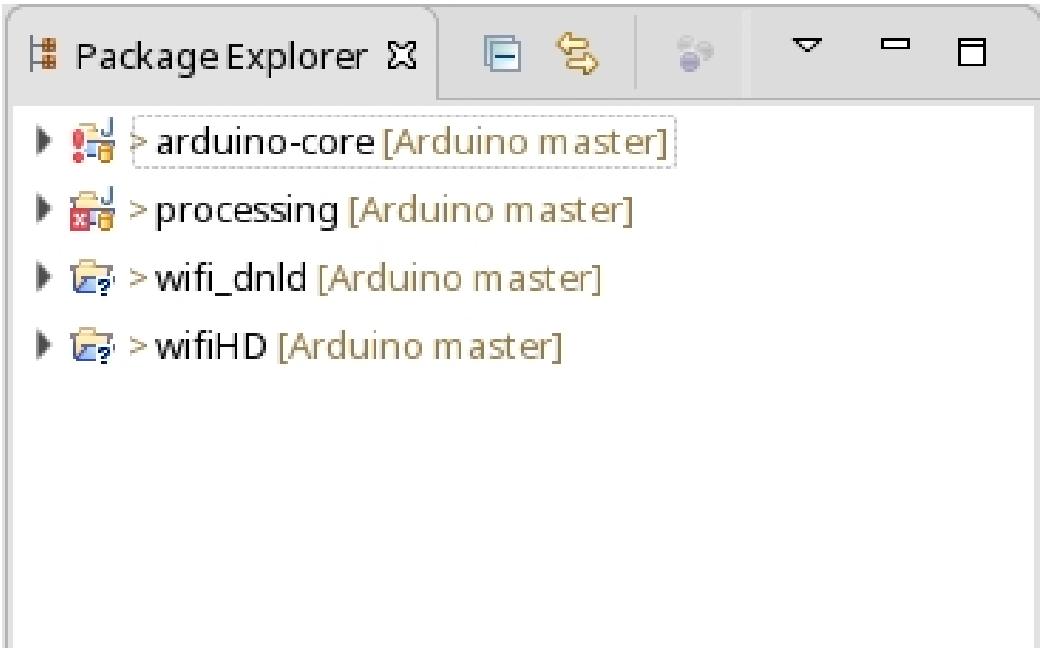
Slika 4.1: Zaslonska slika okna za izbiro projektov iz datotečnega sistema.

V *raziskovalcu paketov* (angl. *Package Explorer* se potem prikaže več projektov, ki so prikazani na sliki 4.2.

4.2 Datoteke z izvorno kodo

V mapi

`delovni prostor/Arduino/app/src/processing/app/`



Slika 4.2: Zaslonska slika projektov v *raziskovalcu paketov*.

smo na novo ustvarili datoteko *OTAMonitor.java*, v kateri je realizirana *serijska konzola preko WiFi* oziroma *konzola OTA*.

Prav tako smo spremenili datoteko *NetworkDiscovery.java* v mapi:

```
delovni prostor/Arduino/...
 ...arduino-core/src/cc/arduino/packages/discoverers/
```

in datoteko *MonitorFactory.java* v mapi:

```
delovni prostor/Arduino/app/src/cc/arduino/packages/
```

4.3 Delovanje serijske konzole pred spremembami

Kadar v menijski vrstici pod menijem *Tools* izberemo možnost *Serial Monitor* ali kliknemo na gumb *Serial Monitor*, se v javanski kodi v datoteki

Editor.java kliče metoda *handleSerial()* [7]. Ta metoda poskrbi, da se javni objekt *serialMonitor* prikaže, kadar ga potrebujemo, in se zapre, ko ga ne potrebujemo. Objekt *serialMonitor* je tipa *AbstractMonitor* [7]. Ta razred je abstraktni razred, ki ne sme biti inicializiran neposredno [36]. Zato končni objekt, ki se inicializira na tem *kazalcu*, deduje razred *AbstractMonitor*. Tip tega objekta se dokončno izbere v metodi *newMonitor* razreda *MonitorFactory* [9].

Ta metoda sprejme kot argument objekt trenutno povezane razvojne ploščice tipa *BoardPort*. V tem objektu sta med shranjenimi podatki protokol povezave, ki je v primeru *OTA* nastavljen na *network*, in pa podatki, ki se prenesejo preko *mDNS* povpraševanja [9]. Metoda kot rezultat vrne razred, ki deduje *AbstractMonitor* [9].

```
public AbstractMonitor newMonitor(BoardPort port) {
  if ("network".equals(port.getProtocol())) {
    if ("yes".equals(port.getPrefs().get("ssh_upload"))) {
      // the board is SSH capable
      return new NetworkMonitor(port);
    } else {
      // SSH not supported, no monitor support
      return null;
    }
  }

  return new SerialMonitor(port);
}
```

Slika 4.3: Zaslonska slika nespremenjene metode *newMonitor()*. Vir kode [9].

Metoda *newMonitor*, prikazana na sliki 4.3, najprej pogleda, če je protokol povezave razvojne ploščice nastavljen na *"network"*.

Če protokol ni enak *"network"*, potem metoda vrne objekt tipa *SerialMonitor*, ki se uporablja za povezavo preko *USB* kabla.

Če je protokol enak *"network"*, potem metoda preveri, ali ploščica podpira *SSH*. Če ploščica podpira *SSH*, potem metoda vrne objekt tipa *NetworkMonitor*. Če ploščica ne podpira *SSH*, potem metoda vrne *null* [9], kar se v metodi *handleSerial* interpretira, kot da serijska povezava preko omrežja ni mogoča [7]. V tem primeru se na črnem polju razvojnega okolja *Arduino IDE* pojavi opozorilo formata:

```
Serial monitor is not supported on network port such as...
...<IP ploščice> for the <ime ploščice> for this release
```

Primer takšnega opozorila za *WEMOS D1 mini*je prikazan na sliki 4.4.

Slika 4.4: Primer neuspelega poskusa vzpostavitve *serjske konzole* za razvojno ploščico *WEMOS D1 mini*, ki ne podpira *SSH*.

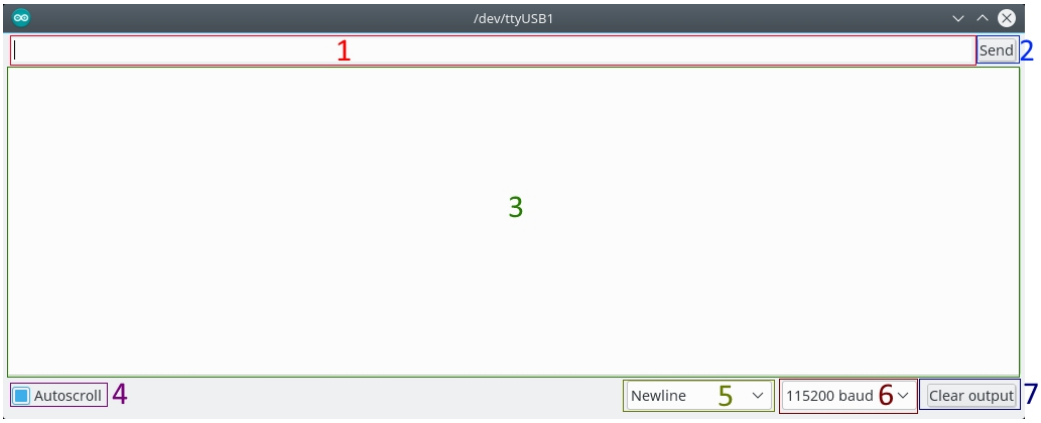
4.3.1 Izgled serijske konzole razreda *Serial Monitor*

Slika 4.5 prikazuje izgled okna *serijske konzole*. Zaradi lažje predstave smo obkrožili in oštevilčili komponente, ki *serijsko konzolo* sestavljajo.

V naslednjem seznamu v istem številčnem vrstnem redu kot na sliki 4.5 predstavimo poimenovanje komponent v kodi in njihove funkcije.

1. Besedilno polje *textField* razreda *JTextField*. V to polje vpišemo besedilo, ki ga želimo poslati na razvojno ploščico.

2. Gumb *Send*, *sendButton* razreda *JButton*. Ob pritisku na ta gumb se besedilo z besedilnega polja *textField* pošlje na razvojno ploščico, potem pa se *textField* počisti. Isti efekt dosežemo s pritiskom tipke *Enter* na tipkovnici.



Slika 4.5: Zaslonska slika *serijske konzole* z oštevilčenimi komponentami.

3. Besedilna površina *textArea* razreda *TextAreaFIFO*. Na to besedilno površino se izpisuje besedilo, ki ga pošlje razvojna ploščica. Ta površina vsebuje tudi drsnik.

4. Potrditveno stikalo *autoscrollBox* razreda *JCheckBox*. To stikalo določa, ali naj se ob izpisu besedila na besedilno površino *textArea* njen drsnik pomakne do dna. S tem lahko sledimo najnovejšemu besedilu z razvojne ploščice ali pa premikanje drsnika prekinemo in si ogledamo starejše zapise.

5. Spustni kombinirani meni *lineEndings* razreda *JComboBox*. S tem spustnim menijem izberemo, kateri simbol za končanje vrstice se bo dodal besedilu iz besedilnega polja *textField* potem, ko bomo to besedlo poslali s pritiskom na gumb *sendButton* oziroma s pritiskom tipke *Enter* na tipkovnici. Na voljo imamo možnosti:

   - *No line ending* — besedilu se simbol za končanje vrstice ne doda.
   - *Newline* — besedilu se doda simbol $\backslash n$.
   - *Carriage return* — besedilu se doda simbol $\backslash r$.
   - *Both NL and CR* — besedilu se dodata oba simbola - $\backslash r$ $\backslash n$.

6. Spustni kombinirani meni *serialRates* razreda *JComboBox*. S tem menijem izberemo *baudno hitrost*, to je hitrost prenosa v bitih na sekundo. S tem izberemo kako hitro bo naš računalnik sprejemal podatke iz razvojne ploščice. *Baudna hitrost* mora biti nastavljena na isto vrednost tako na razvojni ploščici kot na *serijski konzoli*, v nasprotnem primeru dobimo napake pri branju podatkov.

7. Gumb *Clear output*, *clearButton* razreda *JButton*. Ko pritisnemo na ta gumb, se počisti vsebina besedilne površine *textArea*.

4.4 Uveljavljanje sprememb

4.4.1 Razred *NetworkDiscovery*

Ker smo v knjižnjici *OTASerial* iz poglavja 3 izkoristili *mDNS* za oglaševanje vrat strežnika, smo morali tudi okolje *Arduino IDE* spremeniti tako, da te nove informacije nekje shrani. Razreševanje odgovora *MDNS* poteka v metodi *serviceResolved* razreda *NetworkDiscovery*. V tej metodi se informacije, ki jih je prinesel odgovor *MDNS*, shranijo v tabelo znotraj objekta *port* razreda *BoardPort*.

Izsek kode 4.1 prikazuje metodo *serviceResolved*, kjer se nahajajo naši dodatki. Prav tako sta na isti sliki prikazana objekta *port* in *info*.

Objekt *port* zaznamuje razvojno ploščico. V ta objekt želimo vnesti informacije iz odgovora *MDNS*. Te informacije vnesemo s klicem metode:

```
port.getPrefs().put(ključ,vrednost);
```

kjer je *ključ* niz, po katerem vnašamo, *vrednost* pa vrednost, ki jo želimo vnesti.

Objekt *info* pa hrani podatke, ki jih je prinesel odgovor *MDNS*. Po teh podatkih povprašujemo s klicem metode:

```
vrednost = info.getPropertyString(ključ);
```

```
String ota_ser = info.getPropertyString("OTA_Serial");
String ota_ser_port;
if (ota_ser == null || !ota_ser.equals("yes")) {
    ota_ser = "no";
    ota_ser_port = "no";
}else {
    ota_ser_port = info.getPropertyString("OTA_Serial_port");
    if (ota_ser_port == null) {
        ota_ser = "no";
        ota_ser_port = "no";
    } else {
        try {
            int tmpPort = Integer.parseInt(ota_ser_port);
            if (tmpPort < 0 || tmpPort > 65535) {
                ota_ser="no";
                ota_ser_port = "no";
            }
        } catch (NumberFormatException e212) {
            ota_ser="no";
            ota_ser_port = "no";
        }
    }
}
port.getPrefs().put("OTA_Serial", ota_ser);
port.getPrefs().put("OTA_Serial_port", ota_ser_port);
```

Izsek kode 4.1: Naši dodatki znotraj metode *serviceResolved* razreda *serviceResolved*.

ki za željeni *ključ* vrne *vrednost*, ki je niz tipa *String*

Po kodi iz izseka kode 4.1 v prvih dveh vrsticah definiramo lokalni spremenljivki *ota_ser* in *ota_ser_port*. V *ota_ser* se bo shranila vrednost za *mDNS* zapis s ključem *"OTA_Serial"* z možnima vrednostnima *"yes"* ali *"no"*, v *ota_ser_port* pa se bo shranila vrednost za *mDNS* zapis s ključem *"OTA_Serial- _port"* z možnimi vrednostimi 0—65535, kar je dovoljen razpon za *tcp / udp* vrata. Pravilnost *ota_ser_port* se preverja s klicem metode *parseInt* razreda *Integer*, ki niz prevede v število. Če pretvorba ne upe ali če je pretvorjeno število izven razpona 0—65535, se spremenljivkama *ota_ser* in *ota_ser_port* vrednost nastavi na *"no"*. Na koncu pa vrednosti spremenljivk vnesemo v objekt *port*, spremenljivko *ota_ser* pod ključ *"OTA_Serial"* z vrednostjo *"no"* ali *"yes"*, spremenljivko *ota_ser_port* pa pod ključ *"OTA_Serial_port"* z vrednostjo *"no"* ali niz, ki vsebuje številko na razponu 0—65535.

4.4.2 Razred *MonitorFactory*

Končni tip objekta na kazalcu *serialMonitor* se izbere v metodi *newMonitor* razreda *MonitorFactory* [9].

Kot je prikazano na izseku kode 4.2, smo kodo spremenili tako, da se v primeru, ko razvojna ploščica ne podpira *SSH*, preveri, če je vrednost, ki jo vrne klic:

```
port.getPrefs().get("OTA_Serial");
```

enaka vrednosti *"yes"* oziroma, če ploščica podpira *OTA_Serial*. Če ploščica to podpira, metoda *newMonitor* vrne objekt tipa *OTAMonitor*, v nasprotnem primeru pa vrne *null*.

4.4.3 Razred *OTAMonitor*

V tem razredu se nahaja vsa logika za prikaz *OTA konzole* in povezavo s strežnikom na razvojni ploščici.

```
public AbstractMonitor newMonitor(BoardPort port) {
 if ("network".equals(port.getProtocol())) {
  if ("yes".equals(port.getPrefs().get("ssh_upload"))) {
    // the board is SSH capable
    return new NetworkMonitor(port);
    // Dodatki diplomske naloge
  } else if ("yes".equals(port.getPrefs().get("OTA_Serial"))) {
    String debug_port = port.getPrefs().get("OTA_Serial_port");
    return new OTAMonitor(port, Integer.parseInt(debug_port));
  } // konec dodatkov
  else {
    // SSH not supported, no monitor support
    return null;
  }
 }
 return new SerialMonitor(port);
}
```

Izsek kode 4.2: Spremenjena metoda *newMonitor* razreda *AbstractMonitor*.

```
public OTAMonitor(BoardPort port, int debugServerPort) {
 super(port);

 this.serialRates.setVisible(false);
 this.serialRates = null;

 this.ip = port.getAddress();
 this.port = debugServerPort;
 this.worker = new Thread(this);
 this.run = true;
 this.worker.start();

 this.onSendCommand(new ActionListener() {...});
 this.onClearCommand(new ActionListener() {...});
 this.addWindowListener(new WindowAdapter() {...});
}
```

Izsek kode 4.3: Konstruktor razreda *OTAMonitor*.

Tako kot *SerialMonitor* tudi ta razred deduje razred *AbstractTextMonitor* [10], ki deduje *AbstractMonitor* [5]. Za sam izgled *serijske konzole*, od tu naprej imenovana *OTA konzola*, je že poskrbljeno v nadrazredih *AbstractTextMonitor* in *AbstractMonitor*[10, 5].

Za povezavo s strežnikom na razvojni ploščici smo uporabili objekt *printSocket* razreda *Socket* paketa *java.net* [39]. Ta razred se poskusi povezati na naslov *IP* in *vrata* ter vzpostavi podatkovni pretok, v katerega se lahko piše in iz njega bere [39]. Za pisanje v ta pretok podatkov, to je pisanje na strežnik, smo uporabili objekt *sender* razreda *PrintWriter*, ki sprejme izhod pretoka, za branje iz pretoka podatkov, to je branje s strežnika, pa smo uporabili objekt *receieved* razreda *BufferedReader*, ki sprejme vhod pretoka.

Konstruktor razreda *OTAMonitor*

Na sliki 4.3 je prikazana definicija konstruktorja razreda *OTAMonitor*. Ta konstruktor sprejme objekt *port* razreda *BoardPort* ter celo število *debugSer-*

verPort, ki hrani vrata, na katerih je dostopen strežnik na razvojni ploščici.

Najprej se s klicem rezervirane besede *super(port)*, kot je prikazano na sliki 4.4, kliče konstruktor nadrazreda *AbstractTextMonitor*, ki s klicem iste rezervirane besede kliče konstruktor njegovega nadrazreda *AbstractMonitor* [5], v katerem se inicializira okno ter se kliče abstraktna metoda *onCreateWindow*, ki v podrazredu *AbstractTextMonitor* vzpostavi izgled *serijske konzole* [5]. Ker nam izgled serijske konzole *AbstractTextMonitor* ugaja, metode *onCreateWindow* nismo na novo definirali.

Ker spustni kombinirani menij za *baudno hitrost* (ang. *baud rate*) pri brezžični povezavi ni potreben, smo objekt *serialRates* s klicem metode *setVisible(false)* skrili, nato pa nastavili na *null*, kot je prikazano na izseku kode 4.4.

```
super(port);
this.serialRates.setVisible(false);
this.serialRates = null;
```

Izsek kode 4.4: Klic konstrutorja nadrazreda in onemogočenje spustnega kombiniranega menija *serialRates*.

Nato se v spremenljivki *ip* in *port* shranita *IP* naslov ter *vrata* strežnika na razvojni ploščici, nato pa, kot je prikazano na izseku kode 4.5,

```
this.ip = port.getAddress();
this.port = debugServerPort;
this.worker = new Thread(this);
this.run = true;
this.worker.start();
```

Izsek kode 4.5: Nastavitev vrednosti spremenljivk *ip*, *port*, *worker* in *run* v konstruktorju.

se v spremenljivko *worker* shrani kazalec na objekt *Thread*, ki označuje nov objekt nitenja oziroma *nit* [34]. Ta *nit* bo vzporedno izvajala metodo *run()* razreda *OTAMonitor* [34]. Nato se na vrednost *true* nastavi zastavica *run*, ki se uporabi v pogojih zank znotraj metode *run()*, nato pa se s klicem metode *worker.start()* začne izvajanje niti.

V naslednjem koraku pa se poženeta metodi *onSendCommnad()* in *onClearCommand*, ki jih definira nadrazred *AbstractTextMonitor* [5]. Kot je prikazano na izseku kode 4.6 metodi, kot argument sprejmeta objekt razreda *ActionListener* oziroma *poslušalca dogodkov*, v katerem definiramo, kaj se bo zgodilo ob kliku na gumb ali izbiri predmeta iz spustnega menija itn [38].

```
this.onSendCommand(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    send(textField.getText());
    textField.setText("");
  }
});
this.onClearCommand(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    textArea.setText("");
  }
});
```

Izsek kode 4.6: Klic metod *onSendCommand()* in *onClearCommand()* v konstruktorju.

Metoda *onSendCommnad()* objekt *poslušalca dogodkov* iz argumenta doda gumbu *Send* ter *besedilnem polju* na *konzoli OTA* [5], metoda *onClearCommand* pa ta objekt doda gumbu *Clear* na *konzoli OTA* [5]. Ob pritisku na gumb *Send* se besedilo iz *besedilnega polja* pošlje kot argument v metodo *send()* in se iz *besedilnega polja* pobriše. Ob pritisku na gumb *Clear* pa se pobriše besedilo v *besedilni površini konzole OTA*.

```
this.addWindowListener(new WindowAdapter() {
  @Override
  public void windowClosing(WindowEvent e) {
    run = false;
    waitForEnable = false;
  }
});
```

Izsek kode 4.7: *Poslušalec oken*, ki je definiran za *konzolo OTA*.

Kot zadnje pa oknu *OTA konzole* dodamo *poslušalca oken* oziroma *WindowListener*, ki je prikazan na izseku kode 4.7. Tega poslušalca smo definirali tako, da se ob zaprtju okna *OTA konzole* vrednosti zastavic *run* in *waitForEnable* nastavita na *false*. S tem zagotovimo, da se bo ob zaprtju okna končala tudi *nit*, ki izpisuje besedilo z razvojne ploščice.

Metoda *run*

Ker je za *konzolo OTA* potrebno neprestano preverjanje za prisotnost besedila za izpis, smo morali logiko izpisovanja prestaviti v drugo nit, ki teče vzporedno z oknom *konzole OTA*. Razred *OTAMonitor* zato implementira tudi vmesnik *Runnable*, ki nudi metodo *run()*, znotraj katere koda lahko teče v drugi niti [37, 35].

Znotraj metode *run()* se vrši povezovanje na strežnik in branje ter prikazovanje besedila, ki ga strežnik pošlje. Prva zanka v tej metodi je zanka *while* s pogojem:

```
waitForEnable || run
```

To pomeni, da se bo zanka izvajala v primeru, da je vsaj ena od zastavic resnična oziroma enaka *true*.

```
while ((waitForEnable || run) ) {
  while (waitForEnable) {
    try {
      Thread.sleep(200);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  if(!run) return;
```

Izsek kode 4.8: Začetek metode *run()*: glavna zanka in čakalna zanka.

Če je zastavica *waitForEnable* enaka *true*, je okno *konzole OTA* onemogočeno. V tem primeru se bo v naslednjem koraku začela izvajati čakalna zanka *while*, ki bo izvajala *Thread.sleep(200)*, to je spanje za približno 200

```
BufferedReader received = null;

try {
 printSocket = new Socket(ip, port);
 sender = new PrintWriter(printSocket.getOutputStream(), true);
 received = new BufferedReader(
    new InputStreamReader(printSocket.getInputStream()));
```

Izsek kode 4.9: Poskus povezovanja na strežnik na *IP* naslovu *ip* in *vratih port*.

milisekund, dokler je vrednost *waitForEnable* enaka *true*. Ta zanka je vidna na sliki 4.8. Vrstica kode *if(!run) return*, sproži prenehanje izvajanja *niti* v primeru, da smo okno zaprli v trenutku, ko je bila omenjena *nit* v čakalni zanki. S tem se izognemo nepotrebnemu povezovanju na strežnik.

Nato sledi povezovanje na strežnik, šele potem pa zanka *while* za branje podatkov s strežnika, izvajanje katere je odvisno tudi od zastavice *run*. Ko se branje s strežnika zaključi, se izvrši zapiranje podatkovnih pretokov ter povezave s strežnikom.

Podrobnejši opis povezovanja in branja besedila s strežnika sledi v naslednjih razdelkih.

Povezovanje na strežnik na razvojni ploščici

Na strežnik se poskusimo povezati z inicilizacijo objekta *printSocket*. Nato iz objekta izluščimo vhodni in izhodni pretok z uporabo lokalnega objekta *received* ter globalnega objekta *sender*. Ta dva koraka sta prikazana na izseku kode 4.9.

V naslednjem koraku počakamo na pozdravno sporočilo iz strežnika ter preverimo, če je enako nizu:

```
"HELLO:OTASERIAL"
```

Če je sporočilo pravilno, se pošlje odgovor:

```
String OTASerialHello = received.readLine();

if (OTASerialHello.equals("HELLO:OTASERIAL")) {
  sender.println("HELLO:ARDUINO:OTAMONITOR");
```

Izsek kode 4.10: Preverjanje pozdravnega sporočila in odgovor.

```
"HELLO:ARDUINO:OTAMONITOR"
```

Ta pa se preveri na strani strežnika. Ta korak je prikazan na izseku kode 4.10.

Nato se posluša za potrdilno sporočilo s strežnika. To sporočilo se glasi:

```
"RESPONSE:OK"
```

in oznanja, da je povezava vzpostavljena. Ta postopek je prikazan na izseku kode 4.11.

```
String resp = received.readLine();
if (resp.equals("RESPONSE:OK") ) {
  hasServer = true;
} else {
// OTAMonitor se ni uspel povezati
  hasServer = false;
  run = false;
  System.err.println("Wrong \"RESPONSE\" message received ..." )
    ;
}
```

Izsek kode 4.11: Preverjanje potrdilnega sporočila.

Če je povezovanje uspešno, se vrednost zastavice *hasServer* nastavi na *true*. V primeru napačnega pozdravnega ali potrditvenega sporočila, se vrednost zastavic *hasServer* in *run* nastavi na *false*, nato pa se izpiše opozorilno sporočilo o napaki.

Kot je vidno na izseku kode 4.9, je celoten proces povezovanja na strežnik vgnezden v blok *Try-Catch*. Ta blok lovi *izjeme* (angl. *Exceptions*). To

so napake, ki jih lahko *izvržejo* (angl. *throws*) posamične metode oziroma razredi. Z blokom nam *Try-Catch* lahko preprečimo sesutje programa, hkrati pa program lahko na *izjeme* reagira.

```
} catch (NoRouteToHostException e) {
  System.err.println("OTASerial at address " + ip + ":" + port
   + " is not reachable. \n      ... ");
  hasServer = false;
  run = false;
} catch (IOException e) {
  System.err.println("An I/O error has occured ..." );
  hasServer = false;
}
```

Izsek kode 4.12: Lovljenje *izjem NoRouteToHostException* in *IOException*. Opozorilni sporočili sta okrajšani zaradi preglednosti.

V procesu povezovanja na strežnik lovimo izjemi *NoRouteToHostException* in *IOException*. Prvo izvrže razred *Socket*, kadar povezovanje na njegov cilj ni uspel, drugo pa izvrže metoda *readLine()* razreda *BufferedReader*, kadar ze zgodi napaka pri *Vhodno-Izhodnem* prenosu. V primeru, da *Try-Catch* ujame izjemo *NoRouteToHostException*, se izpiše opozorilo o napaki pri povezovanju, potem pa se vrednost zastavic *hasServer* in *run* nastavi na *false*. Če je ujeta izjema *IOException*, se izpiše opozorilo o napaki pri *Vhodno-Izhodnem* prenosu in o poskusu ponovnega povezovanja na strežnik, nato pa se na *false* nastavi samo vrednost zastavice *hasServer*, tako da se glavna zanka lahko ponovno izvede. Koda, v kateri je prikazano reagiranje na ti dve *izjemi*, je vidna na izseku kode 4.12.

Branje besedila s strežnika in prikaz na serijski konzoli

Branje besedila s strežnika se izvaja znotraj zanke *while* s pogojem:

```
run && hasServer
```

Zastavica *run* se nastavi na *true* ob zagonu niti, v kateri teče metoda *run()*. Uporablja se kot indikator, ali naj branje poteka. Zastavica *hasServer*

pove, ali je povezovanje na strežnik uspelo. Ta zanka je prikazana na izseku kode 4.13.

```
char buffer[] = new char[256];
while (run && hasServer) {
  try {
    int l = received.read(buffer);

    if (l >= 0) {
      message("" + new String(buffer, 0, l));
    }else {
      Thread.sleep(1);
    }
  } catch (IOException e1) {
    e1.printStackTrace();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Izsek kode 4.13: Zanka za branje podatkov iz strežnika.

Če je pogoj neresničen, se zanka ustavi. Nato pa v primeru, da *received*, *sender* in *printSocket* obstajajo oziroma niso enaki *null*, se v istem vrstnem redu zaprejo njihovi pretoki s klicem metode *close()*, kot je prikazano na sliki 4.14.

```
if (received != null) received.close();
if (sender != null) sender.close();
if (printSocket != null) printSocket.close();
```

Izsek kode 4.14: Zapiranje podatkovnih pretokov za objekte *received*, *sender* in *printSocket*.

Znotraj bralne zanke se z metodo *read(buffer)* objekta *received* prebere besedilo iz pretoka, ki ga je ustvaril *printSocket*, kot je prikazano na izseku kode 4.13. Ta metoda prekopira vsebino pretoka v tabelo *buffer* tipa *char* in vrne število prekopiranih črk l [32]. Nato iz tabele skonstruiriramo niz, ki prebere črke iz tabele od indeksa 0 do izključno indeksa l:

```
new String(buffer, 0, l)
```

Nato se ta niz uporabi kot argument ob klicu metode *message()*, kot je prikazano na sliki 4.13.

Metodo *message()* nam ponuja nadrazred *AbstractTextMonitor* [5]. Znotraj te metode se besedilo doda (angl. *append*) že obstoječemu besedilu na besedilni ploskvi, potem pa se, če je na *serijski konzoli* aktivirano potrditveno kazalo *Autoscroll*, navpični drsnik premakne na konec besedila [5].

Pošiljanje besedila z metodo *send()*

Ta metoda se kliče ob pritisku na gumb *Send*, kot je opisano v opisu konstruktorja.

Kot prikazuje izsek kode 4.15, se v metodi *send()* z izbiro s stavkom *switch* besedilu pripne simbol za končanje vrstice. Ta simbol izberemo s spustnim menijem *lineEndings* na *konzoli OTA*.

To spremenjneno besedilo se uporabi kot argument metode *print()* objekta *sender*. Metoda *print()* nato besedilo v argumentu pošlje po pretoku, ki ga je ustvaril *printSocket* [33].

Čakanje na omogočanje okna serijske konzole

Nadrazred *AbstractTextMonitor* nam ponuja tudi metodo *onEnableWindow(boolean enable)* [5], ki se kliče ob onemogočanju oziroma omogočanju *serijske konzole* [4].

To metodo smo redefinirali v razredu *OTAMonitor*, saj jo definira tudi nadrazred *AbstractTextMonitor*. Kot je prikazano na izseku kode 4.16, se v tej metodi za vse grafične komponente kliče metoda *setEnabled()*, ki sprejme argument *enable*, ki določi, ali je komponenta omogočena. Najprej smo odstranili *serialRates.setEnabled(enable)*, saj smo spustni meni *serialRates* znotraj konstruktorja nastavili na *null* .

Kadar je onemogočeno okno *konzole OTA* naj bi bila onemogočena tudi njena logika. To smo dosegli s spreminjanjem vrednosti zastavic *run* in *waitForEnable* v delu kode, ki je prikazan na izseku kode 4.17. Če je vrednost

```
private void send(String s) {
 try {
  switch (lineEndings.getSelectedIndex()) {
   case 1:
     s += "\n";
     break;
   case 2:
     s += "\r";
     break;
   case 3:
     s += "\r\n";
     break;
   default:
     break;
  }
  sender.print(s);
  sender.flush();
 } catch (Exception e) {
  e.printStackTrace();
 }
}
```

Izsek kode 4.15: Metoda *send()* razreda *OTAMonitor*.

```
  protected void onEnableWindow(boolean enable) {
    textArea.setEnabled(enable);
    clearButton.setEnabled(enable);
    scrollPane.setEnabled(enable);
    textField.setEnabled(enable);
    sendButton.setEnabled(enable);
    autoscrollBox.setEnabled(enable);
    lineEndings.setEnabled(enable);

    if (enable) { ... }
  }
```

Izsek kode 4.16: Metoda *onEnableWindow()* razreda *OTAMonitor*.

```
if (enable) {
  if (waitForEnable) {
    anzePrintln("[DEBUG: ENABLING WINDOW");
    run = true;
    waitForEnable = false;
  }
} else {
  anzePrintln("[DEBUG: stopping worker - disabled window");
  waitForEnable = true;
  run = false;
}
```

Izsek kode 4.17: Omogočanje in onemogočanje logike *konzole OTA*.

zastavice *enable* enaka *false*, se vrednost zastavice *waitForEnable* nastavi na *true*, nato pa se vrednost zastavice *run* nastavi na *false*. Če pa je vrednost zastavice *enable* enaka *true*, se najprej preveri, ali je vrednost *waitForEnable* enaka *true*. V primeru, da to drži, se najprej na *true* nastavi vrednost zastavice *run*, potem pa se na *false* nastavi vrednost zastavice *waitForEnable*. Vrstni red spreminjanja vrednosti je pomemben zaradi načina njune uporabe v metodi *run()*.

Tako se zagotovi, da se ob nalaganju programa preko *OTA*, kar sproži metodo *onEnableWindow()*, prekine povezava s strežnikom na razvojni ploščici. Ko je nalaganje programa končano, se metoda *onEnableWindow()* sproži še enkrat ter posledično omogoči komponente in ponovno sproži proces povezovanja *serijske konzole* na strežnik.

Poglavje 5

Nastavitve požarnega zidu

Ker ob uporabi *OTA* ter knjižnice *OTASerial* komunikacija poteka preko omrežja, je potrebno preveriti, če požarni zid operacijskega sistema, na katerem teče *Arduino IDE*, dopušča komunikacijo na naslednjih vratih:

- UDP vrata 5353 za *MDNS*;
- UDP vrata za *ArduinoOTA*, privzeto vrata 8266;
- TCP vrata za *OTASerial*, privzeto vrata 23;
- TCP vrata 1000 do 65535, preko katerih se izvede prenos *skice* z uporabo *ArduinoOTA*.

Glede vrat za prenos *skic* se zdi, da se izberejo naključna vrata med 1000 in 65535. Dokumentacije glede izbire vrat nismo našli, do te hipoteze smo prišli s pomočjo uporabe programa *tcpdump*.

Priporočamo, da v primeru znanega *IP* naslova razvojne ploščice v požarnem zidu dovolite komunikacijo preko prej omenjenih vrat samo med razvojnim računalnikom, kjer teče *Arduino IDE*, ter med razvojno ploščico.

Poglavje 6

Primer uporabe OTASerial ter spremenjenega okolja *Arduino IDE*

Za namen primera uporabe predpostavimo, da imamo nameščene vse potrebne pakete oziroma gonilnike, da uporabljamo našo spremenjeno verzijo okolja *Arduino IDE* in da na razvojni ploščici *WEMOS D1 mini*še ni nameščena skica, ki omogoča *ArduinoOTA*.

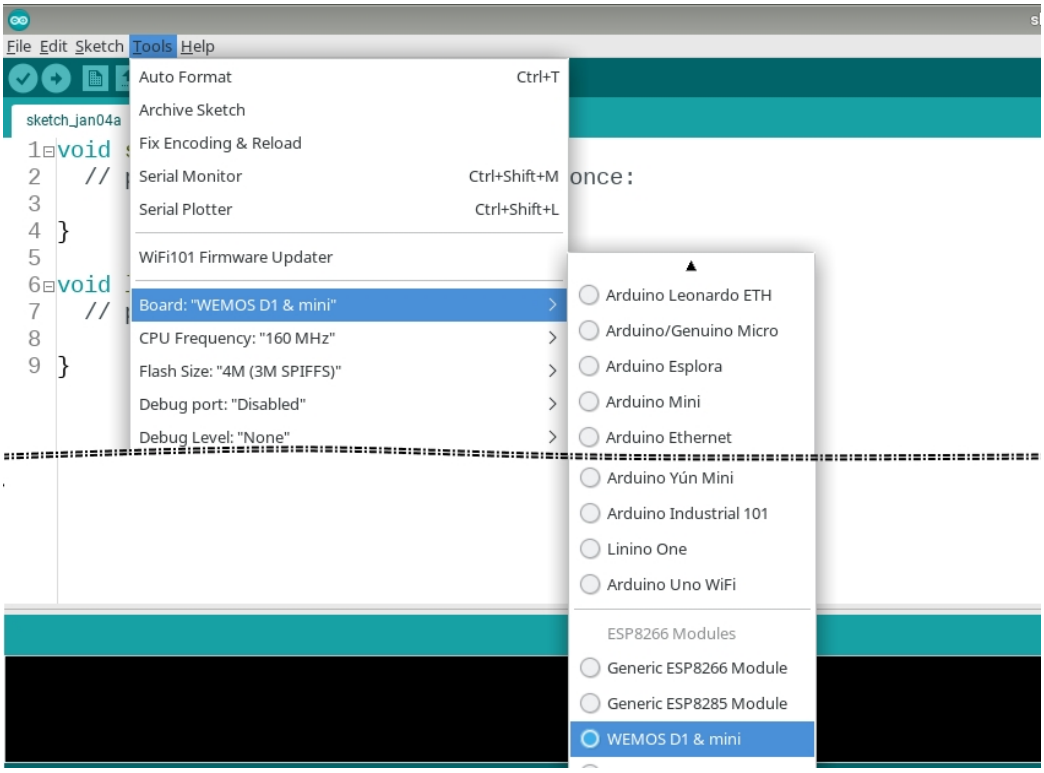
6.1 Nameščanje skice z omogočenim *ArduinoOTA*

Ker na razvojni ploščici trenutno knjižnica *ArduinoOTA* ni aktivna, je potrebno preko *USB* kabla nanjo namestiti skico, ki omogoča *ArduinoOTA*. Najprej poženemo razvojno okolje *Arduino IDE* in pod menijem:

```
Tools > Board
```

izberemo *"WEMOS D1 & mini"*, kot je prikazano na sliki 6.1.

To razvojnemu okolju pove, naj naloži jedrne knjižnice ter nastavitve prevajanja in nalaganja za to razvojno ploščico.



Slika 6.1: Izbira razvojne ploščice znotraj *Arduino IDE*. Zaradi dolgega seznama razvojnih ploščic je slika prirezana, prikazano z vzporednima črtama.

Nato z *USB* kablom razvojno ploščico *WEMOS D1 mini*povežemo z računalnikom in počakamo, dokler se znotraj *Ardino IDE* v meniju

```
Tools > port
```

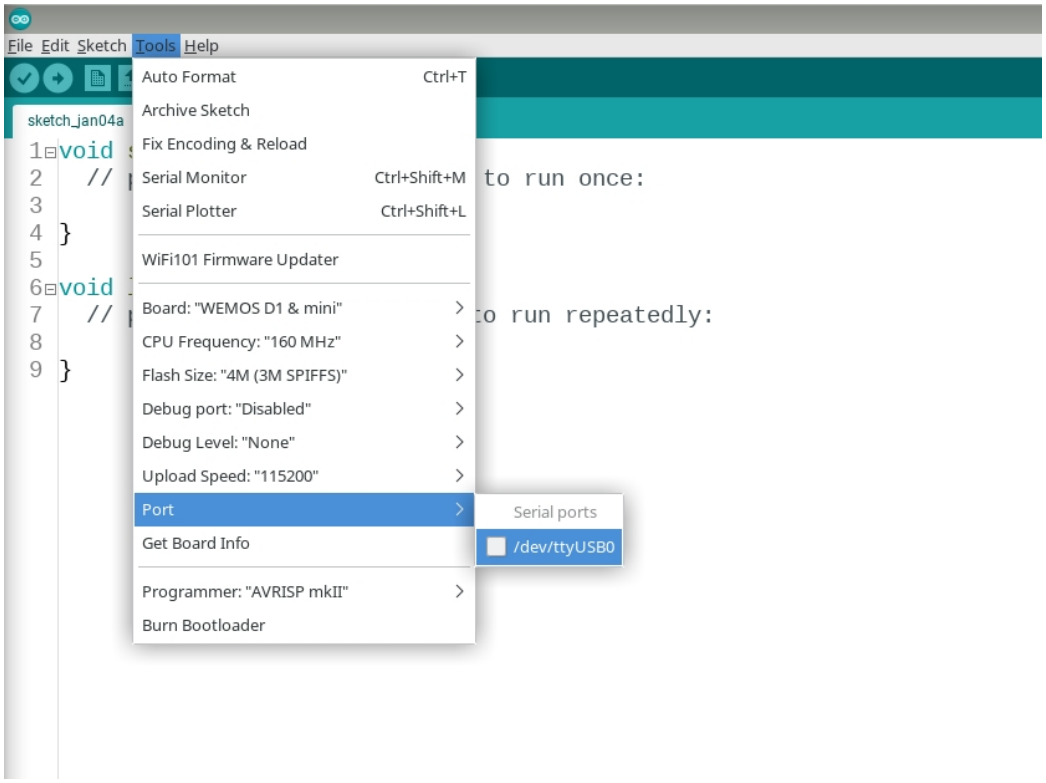
pod rubriko *Serial ports* ne prikaže zapis za razvojno ploščico. V *OpenSUSE* se prikaže možnost:

```
/dev/ttyUSB0,
```

kot je prikazano na sliki 6.2.

Nato s kombinacijo tipk *CTRL + N* odpremo novo skico, v katero prekopiramo vsebino skice *BasicOTA*, ki jo dobimo znotraj menija:

```
File > Examples > ArduinoOTA
```



Slika 6.2: Izbira *vhoda* za komunikacijo z razvojno ploščico.

ali na povezavi [20]. V niz *ssid* vnesemo ime željene dostopne točke, v niz *password* pa vnesemo njeno dostopno geslo. S klikom na gumb *Upload* začnemo nalaganje. Po končanem nalaganju je razvojno ploščico priporočljivo s pritiskom tipke *RESET* ponovno zagnati.

Nato pod menijem:

```
Tools > port
```

čakamo, dokler se pod rubriko *Network ports* ne prikaže nov zapis, ki označuje *omrežni vhod* za *ArduinoOTA*, naslednjega privzetega formata:

```
esp8266-<ID čipa> at <IP naslov razvojne ploščice>
```

S klikom na gumb *Serial Monitor* lahko opazujemo serijski izpis z razvojne ploščice. V skici *BasicOTA* se ob uspešni postavitvi na *serijsko konzolo* izpiše naslednje:

```
Ready
IP address: <IP naslov razvojne ploščice>
```

Če po izpisu tega besedila *omrežni vhod* ni prikazan, svetujemo ponovni zagon okolja *Arduino IDE*, da se povpraševanje *MDNS* še enkrat izvede.

Ko je *omrežni vhod* na voljo, lahko z izbiro tega *vhoda* in pritiskom na gumb *Upload* začnemo *skice* nalagati preko brezžičnega omrežja. Pomembno je omeniti tudi to, da se mora za ohranjanje delovanja *ArduinoOTA* nekje v *skici* klicati metodo *ArduinoOTA.handle()*, v nasprotnem primeru nalaganje *skic* preko brezžičnega omrežja ne bo mogoče.

6.2 Omogočenje knjižnice *OTASerial*

Sedaj imamo omogočeno nalaganje *skic* preko brezžičnega omrežja, vendar se ob pritisku na gumb *Serial Monitor* v črnem polju pojavi sporočilo o nepodprtem *omrežnem vhodu*, ki ga prikazuje slika 4.4. Če razvojna ploščica ni povezana z računalnikom, iz nje ne moremo dobiti serijskega izpisa.

Za rešitev tega problema moramo uporabiti našo knjižnico *OTASerial*. Datoteki *OTASerial.h* in *OTASerial.cpp* moramo prenesti oziroma prekopirati v isto mapo, v kateri se nahaja naša skica oziroma *.ino* datoteka. Na izseku kode 6.1 je prikazana *skica*, ki jo bomo uporabili v tem primeru.

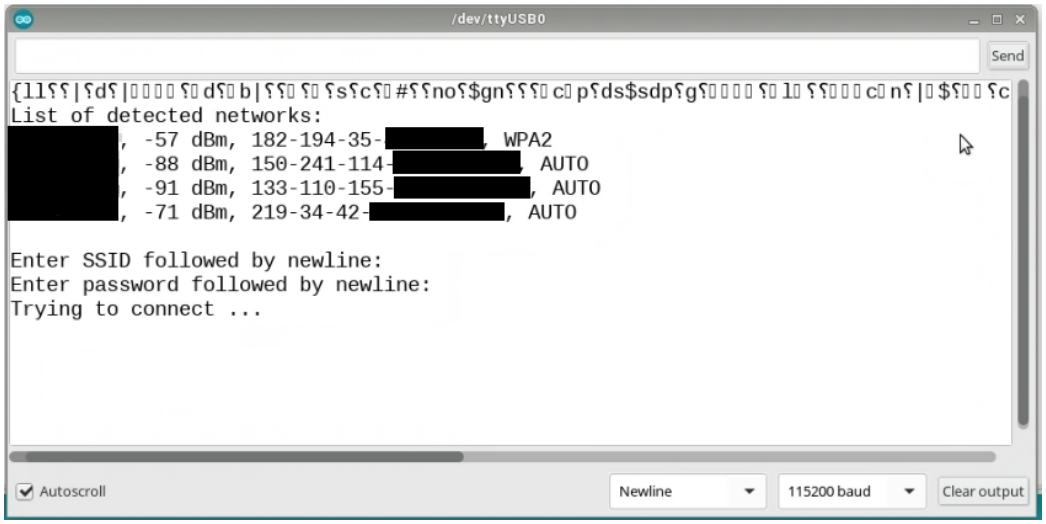
Da knjižnica začne delovati, je potrebno dodati vrstico:

```
#include "OTASerial.h"
```

Vsa koda po tej vrstici bo začela uporabljati naš razred *OTASerial*.

Če se sprehodimo po izseku kode 6.1, se najprej v funkciji *setup()* s klicem *Serial.begin()* inicializira *OTASerial*. Ker pred tem klicem nismo nastavili imena dostopne točke in gesla, se bo v *serijski konzoli* začelo poizvedovanje za ime dostopne točke in za geslo. Primer poizvedovanja prikazuje slika 6.3.

Če poizvedovanja po dotopni točki ne želimo izvesti vsakič, ko ponovno zaženemo razvojno ploščico, potem je pred *Serial.begin()* potrebno dodati še klic:



Slika 6.3: Primer izbire brezžične dostopne točke preko *serijske konzole*.

```
Serial.configWiFi(essid,password);
```

S klicem te metode že znotraj *skice* knjižnici povemo, na katero dostopno točko se želimo povezati.

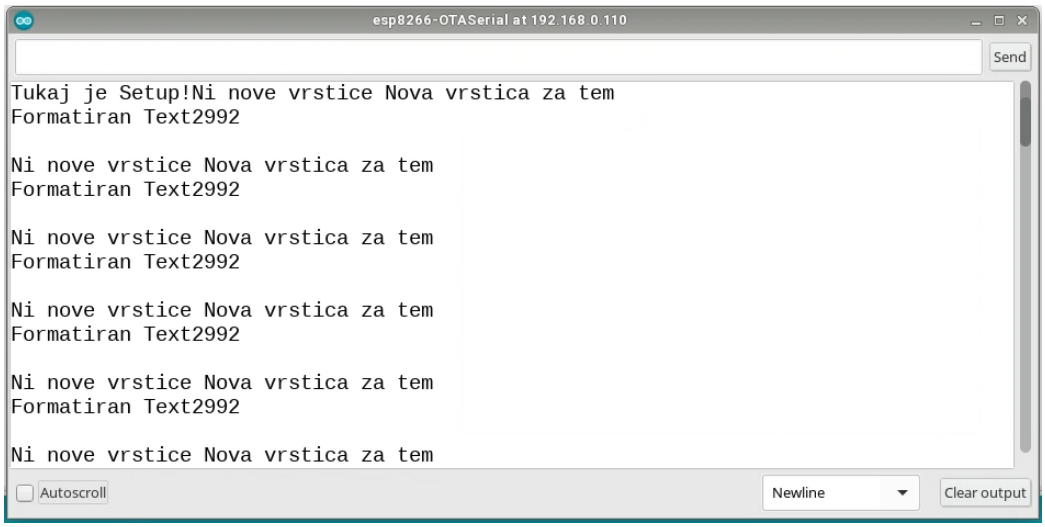
V naslednjem koraku se v zanki s preverjanjem *Serial.available()* zaustavi izvajanje, dokler se na strežnik ne poveže *odjemalec*. Potem pa se *odjemalcu* enkrat pošlje niz:

```
"Tukaj je Setup!"
```

nato pa se v neskončni zanki, ki jo predstavlja funkcija *loop()*, pošiljajo posamezni nizi, vsak uporablja eno od metod *print()*. Slika 6.4 prikazuje rezultat izpisan na *konzoli OTA*.

```
#include "OTASerial.h"

void setup() {
  Serial.begin(115200);

  while(Serial.available()<0){
    delay(1);
  }

  Serial.print("Tukaj je Setup!");

}

void loop() {

  Serial.print("Ni nove vrstice ");
  Serial.println("Nova vrstica za tem ");
  Serial.printf("Formatiran %s%u\n\n","Text",2992);
  delay(100);
}
```

Izsek kode 6.1: *Skica* uporabljena za primer uporabe.



Slika 6.4: Rezultat *skice* iz izseka kode 6.1 izpisan na *OTA konzoli*.

Poglavje 7

Sklepne ugotovitve

Če povzamemo, smo v diplomski nalogi najprej predstavili problem onemogočene serijkse komunikacije med razvojno ploščico ter razvojnim računalnikom v primeru uporabe *OTA*, nato pa smo opisali razvojno ploščico *WEMOS D1 mini*ter razvojni okolji *Arduino IDE* in *Eclipse IDE*.

V naslednjem poglavju smo opisali izdelavo *C++* knjižnice *OTASerial*, ki razvojni ploščici omogoča serijsko komunikacijo preko brezžičnega omrežja. Opisali smo redefiniciji metod *write()*, preko katerih vse metode *print()* pošljejo podatke na *serijsko konzolo* oziroma z uporabo naše knjižnice preko *WiFi* na *konzolo OTA*.

Nato pa smo opisali, kako smo spremenili razvojno okolje *Arduino IDE* tako, da sprejme sporočila, poslana z uporabo knjižnice *OTASerial* in jih prikaže v *konzoli OTA*.

Sledili sta poglavji z navodili za nastavitev požarnega zidu ter primer uporabe našega izdelka, v katerem smo podali preprost primer *skice*, ki uporablja knjižnico *OTASerial*.

7.1 Težave med izdelavo diplomske naloge

Med izdelavo diplomske naloge do večjih težav ni prišlo. Nekaj preglavic nam je povzročala slabo označena izbira načina prenosa *skice*, zaradi katere smo

skico večkrat začeli nalagati preko *USB* kabla in ne preko *OTA*. Rešitev za to je, da razvojne ploščice na napajanje ne povežemo na računalnik, kjer jo programiramo. Tako prenos *skic* preko *USB* kabla sploh ne bo mogoč.

7.2 Možne izboljšave rešitve diplomske naloge

Knjižnica *OTASerial*

Sama knjižnica *OTASerial* je zelo preprosta. Izboljšav brez dolgotrajnejše uporabe knjižnice ne moremo podati.

Spremenjeno okolje *ArduinoIDE*

Pri spremenjenem okolju *ArduinoIDE*, še posebej pa pri *konzoli OTA* oziroma razredu *OTAMonitor*, bi bila možna izboljšava dodatek vizuelne informacije, kdaj se *OTA konzola* povezuje na strežnik, kdaj ta povezava ni uspela itn. Trenutno je programerju povratna informaciji vidna samo kot opozorilno sporočilo v črnem polju okolja *Arduino IDE*.

Literatura

[1] Apache ant - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Apache_Ant. [Dostopano: 7. 12. 2017].

[2] Arduino. Arduino reference - arduino style guide for writing libraries. Dosegljivo: https://www.arduino.cc/en/Reference/APIStyleGuide. [Dostopano: 8. 12. 2017].

[3] Arduino. Arduino trubleshooting - why i can't upload my programs to the arduino board? Dosegljivo: https://www.arduino.cc/en/Guide/Troubleshooting#toc1. [Dostopano: 6. 1. 2018].

[4] arduino. Github - arduino/arduino - abstractmonitor.java. Dosegljivo: https://github.com/arduino/Arduino/blob/master/app/src/processing/app/AbstractMonitor.java. [Dostopano: 6. 12. 2017].

[5] arduino. Github - arduino/arduino - abstracttextmonitor.java. Dosegljivo: https://github.com/arduino/Arduino/blob/master/app/src/processing/app/AbstractTextMonitor.java. [Dostopano: 6. 12. 2017].

[6] arduino. Github - arduino/arduino - building arduino. Dosegljivo: https://github.com/arduino/Arduino/wiki/Building-Arduino. [Dostopano: 6. 1. 2018].

[7] arduino. Github - arduino/arduino - editor.java. Dosegljivo: https://github.com/arduino/Arduino/blob/master/app/src/processing/app/Editor.java. [Dostopano: 5. 12. 2017].

[8] arduino. Github - arduino/arduino - license. Dosegljivo: https://github.com/arduino/Arduino/blob/master/license.txt. [Dostopano: 9. 9. 2017].

[9] arduino. Github - arduino/arduino - monitorfactory.java. Dosegljivo: https://github.com/arduino/Arduino/blob/master/app/src/cc/arduino/packages/MonitorFactory.java. [Dostopano: 5. 12. 2017].

[10] arduino. Github - arduino/arduino - serialmonitor.java. Dosegljivo: https://github.com/arduino/Arduino/blob/master/app/src/processing/app/SerialMonitor.java. [Dostopano: 6. 12. 2017].

[11] arduino. Github - arduino/arduino: open-source electronics prototyping platform. Dosegljivo: https://github.com/arduino/Arduino. [Dostopano: 9. 9. 2017].

[12] arduino. Github - arduino/arduino/build. Dosegljivo: https://github.com/arduino/Arduino/tree/master/build. [Dostopano: 12. 6. 2017].

[13] arduino. Github wiki - arduino/arduino wiki : Building arduino. Dosegljivo: https://github.com/arduino/Arduino/wiki/Building-Arduino. [Dostopano: 5. 12. 2017].

[14] arduino. Github wiki - arduino/arduino wiki : Development policy. Dosegljivo: https://github.com/arduino/Arduino/wiki/Development-Policy. [Dostopano: 6. 12. 2017].

[15] Arduino - environment: Multiple files. Dosegljivo: https://www.arduino.cc/en/Guide/Environment#toc8. [Dostopano: 9. 9. 2017].

[16] Arduino - environment: Writing sketches. Dosegljivo: https://www.arduino.cc/en/Guide/Environment#toc1. [Dostopano: 9. 9. 2017].

[17] Arduino - sketch. Dosegljivo: https://www.arduino.cc/en/tutorial/sketch. [Dostopano: 9. 9. 2017].

[18] Arduino - software. Dosegljivo: https://www.arduino.cc/en/Main/Software. [Dostopano: 15. 9. 2017].

[19] Eclipse (software) - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Eclipse_(software). [Dostopano: 7. 12. 2017].

[20] esp8266. Github - esp8266/arduino - basicota.ino. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/libraries/ArduinoOTA/examples/BasicOTA/BasicOTA.ino. [Dostopano: 5. 1. 2018].

[21] esp8266. Github - esp8266/arduino - hardwareserial.cpp. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/cores/esp8266/HardwareSerial.cpp. [Dostopano: 6. 1. 2018].

[22] esp8266. Github - esp8266/arduino - hardwareserial.h. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/cores/esp8266/HardwareSerial.h. [Dostopano: 8. 12. 2017].

[23] esp8266. Github - esp8266/arduino - print.cpp. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/cores/esp8266/Print.cpp. [Dostopano: 8. 12. 2017].

[24] esp8266. Github - esp8266/arduino - print.h. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/cores/esp8266/Print.h. [Dostopano: 8. 12. 2017].

[25] esp8266. Github - esp8266/arduino - uart.h. Dosegljivo: https://github.com/esp8266/Arduino/blob/master/cores/esp8266/uart.h. [Dostopano: 6. 1. 2018].

[26] esp8266. Github - esp8266/arduino: Esp8266 core for arduino. Dosegljivo: https://github.com/esp8266/Arduino. [Dostopano: 6. 1. 2017].

[27] Apache Software Foundation. Apache ant™ user manual. Dosegljivo: http://ant.apache.org/manual/index.html. [Dostopano: 7. 12. 2017].

[28] Apache Software Foundation. Writing a simple buildfile. Dosegljivo: http://ant.apache.org/manual/using.html. [Dostopano: 7. 12. 2017].

[29] Apache Software Foundation. Writing your own task. Dosegljivo: ant.apache.org/manual/develop.html#writingowntask. [Dostopano: 7. 12. 2017].

[30] The Eclipse Foundation. Eclipse - ide. Dosegljivo: https://www.eclipse.org/ide/. [Dostopano: 6. 12. 2017].

[31] The Eclipse Foundation. Eclipse - packages. Dosegljivo: https://www.eclipse.org/downloads/packages/. [Dostopano: 7. 12. 2017].

[32] Oracle. Java api - class bufferedreader. Dosegljivo: https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html. [Dostopano: 6. 12. 2017].

[33] Oracle. Java api - class printwriter. Dosegljivo: https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html. [Dostopano: 6. 12. 2017].

[34] Oracle. Java api - class thread. Dosegljivo: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html. [Dostopano: 6. 12. 2017].

[35] Oracle. Java api - interface runnable. Dosegljivo: https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html. [Dostopano: 6. 12. 2017].

[36] Oracle. The java™ tutorials - abstract methods and classes. Dosegljivo: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html. [Dostopano: 5. 12. 2017].

[37] Oracle. The java™ tutorials - defining and starting a thread. Dosegljivo: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html. [Dostopano: 6. 12. 2017].

[38] Oracle. The java™ tutorials - how to write an action listener. Dosegljivo: https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html. [Dostopano: 6. 12. 2017].

[39] Oracle. The java™ tutorials - what is a socket? Dosegljivo: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html. [Dostopano: 6. 12. 2017].

[40] Ota introduction. Dosegljivo: http://esp8266.github.io/Arduino/versions/2.0.0/doc/ota_updates/ota_updates.html. [Dostopano: 15. 9. 2017].

[41] Processing - sketches and sketchbook. Dosegljivo: https://processing.org/reference/environment/#Sketchbook. [Dostopano: 9. 9. 2017].

[42] Sparkfun - espressif esp 8266. Dosegljivo: https://www.sparkfun.com/products/13678. [Dostopano: 15. 9. 2017].

[43] Matt Ward. Eclipse foundation - legal resources. Dosegljivo: https://www.eclipse.org/legal/. [Dostopano: 7. 12. 2017].

[44] Wemos d1 mini wiki. Dosegljivo: https://wiki.wemos.cc/products:d1:d1_mini. [Dostopano: 15. 9. 2017].

[45] Wemos d1 mini wiki - drivers. Dosegljivo: https://wiki.wemos.cc/downloads. [Dostopano: 9. 9. 2017].

[46] Wemos d1 mini wiki - get started in arduino. Dosegljivo: https://wiki.wemos.cc/tutorials:get_started:get_started_in_arduino. [Dostopano: 15. 9. 2017].

[47] Wemos d1 mini wiki - get started in nodemcu. Dosegljivo: https://wiki.wemos.cc/tutorials:get_started:get_started_in_nodemcu. [Dostopano: 15. 9. 2017].

[48] Wemos d1 mini na aliexpress. Dosegljivo: https://www.aliexpress.com/store/product/D1-mini-Mini-NodeMcu-4M-bytes-Lua-WIFI-Internet-of-Things-development-board-based-ESP8266/1331105_32529101036.html. [Dostopano: 15. 9. 2017].