Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Ožbolt Menegatti

Vzorčni primeri programskih rešitev za Raspberry Pi

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

prof. dr. Miha Mraz
MENTOR

doc. dr. Miha Moškon
SOMENTOR

Ljubljana, 2014

Namesto te strani se vstavi *original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!*

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom prof. dr. Mihe Mraza in somentorstvom doc. dr. Mihe Moškona,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Ožbolt Menegatti, Ljubljana, september 2014.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Ožbolt Menegatti

Vzorčni primeri programskih rešitev za Raspberry Pi

POVZETEK

Ponudba ARM razvojnih ploščic na trgu danes sega od najcenejših 8-bitnih, pa vse do zmogljivejših, za namizno rabo uporabnih računal. Ploščica, ki izstopa po popularnosti je Raspberry Pi. Čeprav lahko zanjo dobimo množico primerov nalog na spletu, večini teh manjkajo priključki na vhodno/izhodne naprave, kar bi olajšalo učenje programiranja vgrajenih sistemov.

V okviru diplomskega dela smo razvili primerek vhodno/izhodne razširitvene ploščice, ki vsebuje osnovne kompontente, kot so LED diode, gumbi, zvočnik in svetlobni senzor. Naknadno smo razvili orodje za lažje programiranje Raspberry Pi, ki je zgrajeno s pomočjo HTML in javascript, kar dodatno olajša programiranje. Okoli teh dveh orodij je napisanih devet nalog, ki segajo od osnovnih vhodno/izhodnih operacij z gumbi in LED diodami, pa do dveh težjih nalog, ki se vrtita okoli PWM (angl. *pulse width modulation*) izhoda in I^2C komunikacije. Poleg C programskega jezika se dotaknemo tudi drugih programskih jezikov (npr. primer v Pythonu). Uporabnost Raspberry Pi sega tudi na področje zbirniškega jezika ARM, čemur je posvečena predzadnja izmed devetih nalog.

Ključne besede: Raspberry Pi, Vgrajeni sistemi, Osnove programiranja

University of Ljubljana
Faculty of Computer and Information Science

Ožbolt Menegatti

Case study examples for Raspberry Pi

ABSTRACT

Wide range of ARM development boards is currently available on the market. Their complexity spans from the simple 8-bit boards to more powerful ones that are also suitable to use as general purpose computers. The most popular ARM board currently available is without any doubt Raspberry Pi. Several study examples that demonstrate its usage are available on the internet. While the input/output devices available in the basic set are very fundamental, these examples are hard to use in the educational purposes directly.

We have developed our own input/output extension board with the basic components, such as LED module, buttons, speaker and light sensor. We additionally implemented HTML and javascript based development environment for Raspberry Pi programming. 9 educational case study examples, which demonstrate the basic tasks, such as buttons and LED module control, and also more complex tasks, such as PWM (Pulse Width Modulation) output control and I^2C communication, were written. Even though the majority of examples were written in C programming language, we also used Python and assembler in some of them.

Key words: Raspberry Pi, Embedded systems, Programming basics

ZAHVALA

Zahvala gre mentorju prof. Mihi Mrazu za strpnost in vzpodbudo, Jerneju Sorti za pomoč pri V/I ploščici, Dorijanu Morelu in Jožefu Stepanu iz FE za zastonj rezkanje ploščice in družini za vso podporo. Posebej pa bi se rad zahvalil še različnim organizacijam, ki razvijajo uporabljena odprtokodna orodja. To so Raspberry Pi Fundation, The Pocoo Team, Free Software Foundation, Python Software Foundation, KiCAD team, Twitter (Bootstrap) in jQuery Team.

— Ožbolt Menegatti, Ljubljana, september 2014.

KAZALO

1 Uvod

Diplomska naloga se osredotoča na delo s ploščico Raspberry Pi. Ta od konkurence ne odstopa po tehničnih specifikacijah, ampak po njeni razširjenosti in podpori, ki jo tako velika skupnost nudi. Vseeno si poglejmo ponudbo drugih ARM ploščic.

1.1 Ponudba ploščic

V letu 2014 lahko trg pri ponudbi ARM ploščic grobo razdelimo v dve kategorije. V prvo uvrstimo ploščice za razvoj specifičnih aplikacij, ki so prepredene z najrazličnejšimi komponentami. Drugo skupino predstavljajo ploščice za splošno uporabo, katere so večinoma zmogljivejše, a ne vsebujejo tako velikega števila posebnih komponent. Namesto tega omogočajo poganjanje kompleksnejših operacijskih sistemov in imajo dovolj računske moči za predvajanje filmskih posnetkov, brskanje po spletu in podobno. Kljub vsemu pa ohranjajo svojo ARM naravo preprostejših ploščic iz prve kategorije.

Prva skupina je namenjena inžinirjem, katerih namen je razvoj novih izdelkov. Ta skupina ima veliko konkurenco na trgu, z izbiro ponudnika pa nas pogosto prisilno omeji na ponudbo enega ponudnika (angl. *vendor lock-in*).

Druga skupina, katera se je pojavila skupaj s prenosom velikega števila programske opreme na platformo ARM, pa zgornjih lastnosti nima. Programska oprema, ki jo poganjamo na teh ploščicah (primer na sliki 3.1), je večinokrat odprta. Te ploščice praviloma vsebujejo USB in Ethernet priključek, nek slikovni in zvočni izhod ter napajalni vhod. Mnogo ploščic vsebuje še dodatne podatkovne vhodno/izhodne (angl. *input/output, I/O*) povezave. Ponujeni procesorji so po računski moči primerljivi s procesorji Pentium III iz začetka tega stoletja, čeprav v prihodnosti pričakujemo spremembe. Grafična moč je večja, kot je bila v grafičnih karticah iz istih let. To nam prinese možnost predvajanja zelo kakovostnih video posnetkov.

1.2 Razvojna okolja

Pri ponudbi komercialnih ARM ploščic s strani proizvajalcev komponent za vgrajene sisteme, so razvojna okolja lastniška, pogosto draga in nezdružljiva z drugimi napravami. Kot primer iz lastnih izkušenj lahko podam program Keil MDK-ARM. Njegova uporaba je brezplačna do neke meje velikosti programa, za kaj več pa cene uporabe skočijo do več tisoč evrov. Tehnični problem tu je ta, da je protokol nalaganja izvedljive datoteke na ploščico lastniški.

Veliko širša ponudba je na trgu močnejših ploščic. Tu je dostop do ploščice enak dostopu do kateregakoli Linux sistema. Tako lahko priredimo skorajda vsako obstoječe razvojno orodje programiranju za ARM ploščico. Drug način pa je uporaba spletnih razvojnih okolij, kot je denimo AdaFruit WebIDE. Odprti protokol na Raspberry Pi omogoča tudi, da lahko napišemo sami svoj nalagalnik in zaganjalnik za datoteke.

Razvojno okolje, ki je bilo uporabljeno za programiranje v tem delu, je spletni vmesnik razvit v Pythonu. Orodje podpira nalaganje, kompilacijo in poganjanje programov na Raspberry Pi.

1.3 Cilj diplomske naloge

V okviru diplomske naloge smo razvili vhodno/izhodno razširitveno ploščico, ki omogoča hitrejše in lažje učenje programiranja. Poleg tega smo razvili tudi orodje za programiranje Raspberry Pi, ki je popolnoma neodvisno od platforme, saj deluje preko HTML vmesnika. Glavni doprinos je seznam vzorčnih nalog, ki bralca popeljejo v svet iskanja pravih vrstic v stotinah straneh podatkovnih listov različnih naprav in v svet sledenja pravim linijam

v shemah različnih elementov. Nato te informacije prelijemo v programsko kodo, ki je poleg svoje poučnosti tudi zabavna z igranjem pesmic, prižiganjem luči in odzivanjem na svetlobo okolice.

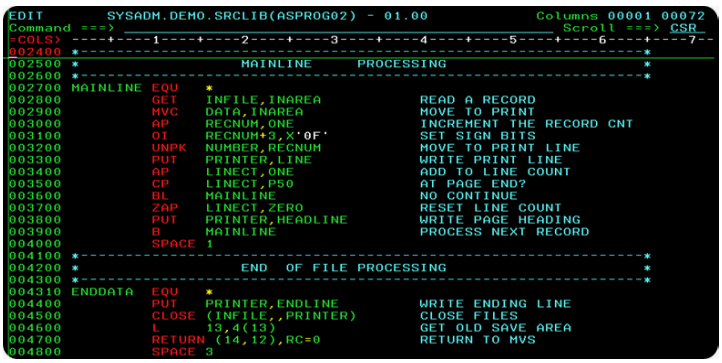
1.4 Členitev Dela

V poglavju 2 opišemo zgodovino razvojnih okolij za programiranje in razloge za odločitev za dotično ARM platformo. Poglavje 3 navede in pojasni izbor uporabljene strojne in programske izbire pri diplomski nalogi. V poglavju 4 so navedeni primeri vzorčnih izobraževalnih nalog. Vsaka izmed teh je ločena na njen opis in rešitev. Poglavji 3 in 4 dopolnjujeta še dodatka A in B.

2 Opis problema

Leto 1953 je v tehnološkem svetu zaznamoval prihod prvega komercialnega računalnika, IBM 701. IBM je s predajo 19-ih *mainframe* računalnikov načel svoj primat na tem področju. Preko serij IBM 7000 (1960) in kasneje IBM 360 (1964) se je v programskem svetu razvijalo orodja predvsem za to procesorsko arhitekturo. Navkljub obstoju drugih arhitektur so bila predavanja na Ameriških univerzah posvečena predvsem Fortranu in BAL-u (IBM Basic assembly language - slika 2.1). Čeprav so mnoga podjetja, kot sta CDC in DEC dobro kljubovala, nikoli niso uspeli prevzeti primata IBM-u.

Skorajda 20 let pozneje je Intel izdal svoj prvi procesor za domačo uporabo. Leta 1971 je tako na sceno prišel mikroprocesor Intel 4004, leto za njim pa še bolj znani Intel 8008. Kmalu so se na trgu začeli pojavljati računalniki za domačo uporabo, ki pa so le redkoma uporabljali Intelov procesor. Vendar Intel ni zaspal. Nadaljeval je svoj pohod na trg in uspel s procesorjem, ki ga je IBM uporabil pri izdelavi računalnika IBM PC (slika 2.2). Leta 1981 je ta PC, ki je uporabljal procesor Intel 8088, povzročil hiter vzpon trga osebnega računalništva. Po tem, ko so kloni IBM PC-ja preplavili trg, takratni glavni igralec na trgu IBM, ni več uspel ustaviti Intelove arhitekture X86. Za tem so šole



Slika 2.1: IBM-ov zbirniški jezik.

in univerze v večini za svojo vzele Intelovo strojno in Microsoftovo programsko opremo in jo pri učenju uporabljajo še danes.

Z razširitvijo manj zmogljivih računalnikov kot so mobilni telefoni in vgrajene naprave, pa je vedno več moči začelo pridobivati tudi podjetje ARM. Njihov trg se zadnja leta vse bolj prekriva s trgom Intela. Tako smo priča procesu, ko želi ARM, podobno kot Intel tri desetletja pred tem, s cenejšimi rešitvami spodkopati svojo konkurenco.

Začelo se je z zelo poceni ARM procesorji za naprave, kot so mikrovalovne naprave, avtomobilske naprave, pa vse do pametnih WC školjk iz Japonske. Nadaljuje se s postopnim predstavljanjem bolj in bolj zmogljivih Cortex različic ARM procesorjev, ki preko zmogljivejših telefonov, cenejših prenosnih računalnikov in varčnejših strežniških proce-

Slika 2.2: IBM-ov lastni pokop: originalni IBM PC.

sorjev vse bolj silijo v področje, ki je bilo še pred pol desetletja trdno pod nadzorom Intela in njegove X86 arhitekture.

Kot je bilo že nakazano, šolstvo vedno sledi trgu. In na trgu se pojavljajo sistemi, ki bazirajo na ARM in Linux tehnologijah. Učenje preprostejših in manj zmogljivejših ARM sistemov je mogoče in se izvaja z uporabo ploščic, kot je naprimer Discovery podjetja STM32 (slika 2.3). Za programiranje močnejših ARM čipov pa lahko uporabimo ali že obstoječe pametne telefone ali pa kupimo ploščico, ki je namenjena programiranju. Danes je na področju mobilne telefonije pri proizvajalcih praksa, da svoj produkt zaklenejo, tako da je možno programiranje teh naprav na le vnaprej določen način. Uporabiti moramo torej že naloženo programsko opremo in razvijalska orodja prilagojena za lete (kot st npr. Android ali Windows Phone). Ta orodja nam večinoma ne omogočajo polnega dostopa do strojne opreme, zato se odločimo za drugo opcijo.

Izbrati moramo razvojno ploščico in osnovno programsko opremo, na kateri bomo razvijali. Če želimo poln dostop, moramo biti sposobni spreminjati že naloženo programsko opremo. Zaradi zgoraj navedenih razlogov smo se odločili, da za svoje potrebe diplomske naloge izberemo ploščico Raspberry Pi in operacijski sistem Linux.

Razvojna ploščica sicer ni polno odprta, za nekatere dele strojne opreme ni na voljo celotne dokumentacije, a ta del je relativno majhen, tako da nas pri učenju programiranja Raspberry Pi ne ovira. Operacijski sistem Linux, na drugi strani je popolnoma odprt za spremembe v izvorni kodi.

Slika 2.3: Ploščica STM32f4 Discovery.

3 Izbira in uporaba orodij

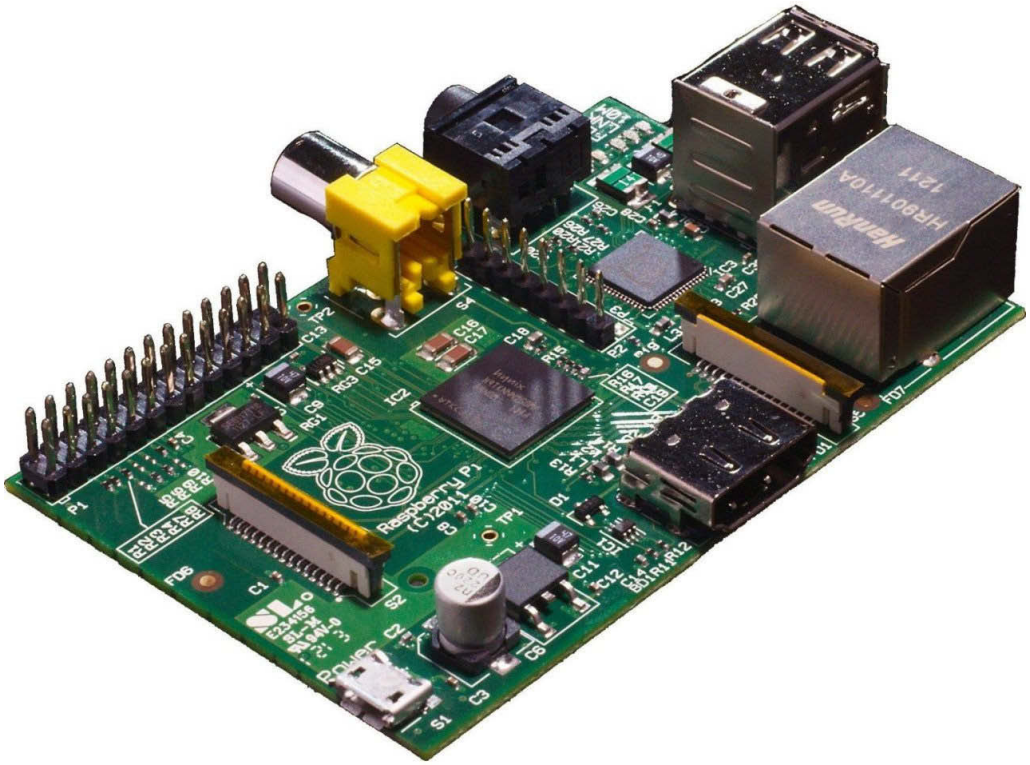
V pričujočem poglavju najprej opišemo potek izbire orodij ter temu pripadajoč razvoj dodatkov. Med te sodi vhodno/izhodna ploščica razvita za namene lažjega programiranja. Kasneje opišemo programsko opremo, ki je bila razvita za isti namen. V zadnjem delu opišemo še primer uporabe teh orodij.

3.1 Razvojna ploščica Raspberry Pi

Zadnje čase prodaja raste manjšim, tako imenovanim *credit-card-sized* računalnikom. Temelje je postavila fundacija Raspberry Pi z istoimensko ploščico. Že pred izidom končnega produkta so prednaročila preplavila podjetje. Podjetje je nato v prvem dnevu prodalo 100.000 produktov, do danes pa beležijo 2,5 milijona prodanih ploščic.

V pričakovanju naslednjika prve verzije ploščice na trg prihajajo konkurentje kot so

- Beagleboard,
- HummingBoard,
- Banana Pi itd.



Slika 3.1: Raspberry Pi.

Prav vsak izmed prej omenjenih ima kakšno prednost pred starejšim Raspberry Pi. Beagleboard ima veliko več GPIO (angl. *General-purpose input/output*) pinov, HummingBoard je na voljo v več verzijah (vsaka je hitrejša od prejšnje), Banana Pi pa je zaradi svojega izvora na Kitajskem izjemno poceni glede na specifikacije. Vendar pa nobena ploščica ne nudi tako zavzete skupnosti in podpore, ki pride le s tako veliko skupino ljudi, kot jo ima za seboj Raspberry Pi.

Specifikacije Raspberry Pi, ki so navedene v tabeli 3.1 pokažejo, da imamo za potrebe učenja programiranja na voljo ARM procesor, 256 ali 512 MB rama (odvisno od modela) in veliko GPIO pinov, ki služijo za povezovanje zunanjih naprav.

Interakcija s pini lahko poteka na več načinov. Najosnovnejši je ta, da pogledamo v specifikacije dotičnega ARM procesorja in nastavljamo smeri pinov, njihove funkcije, beremo in pišemo stanja preko pisanja po registrih na znanih naslovih.

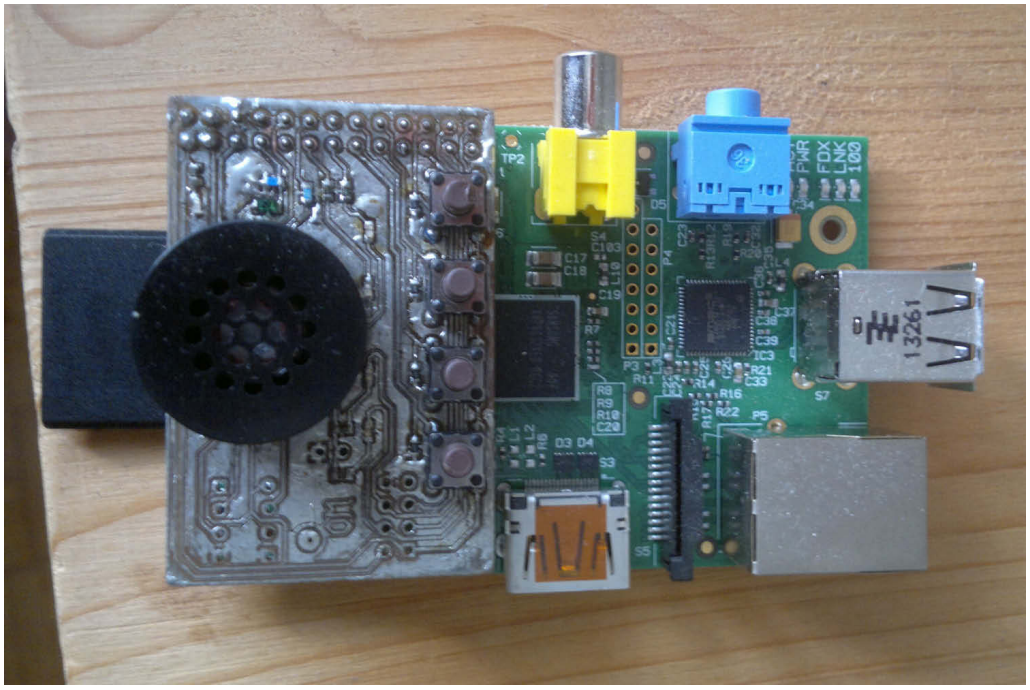
Lažji način je uporaba že obstoječih knjižnic, ki implementirajo konstante in funkcije, preko katerih lahko na primer le v nekaj vrsticah prižgemo ali ugasnemo LED diodo, ki

	Model A	Model B	Model B+
System-on-a-chip	Broadcom BCM2835 (CPU + GPU). SDRAM is a separate chip stacked on top)		
CPU:	700 MHz ARM11 ARM1176JZF-S core		
GPU:	Broadcom VideoCore IV,OpenGL ES 2.0,OpenVG 1080p30 H.264 high-profile encode/decode		
Memory (SDRAM)	256 MB	256 MB (until 15 Oct 2012); 512 MB (since 15 Oct 2012)	512 MB
USB 2.0 ports:	1 (provided by the BCM2835)	2 (via integrated USB hub)	4 (via intergrated USB hub)
Video outputs:	Composite video — Composite RCA, HDMI (not at the same time)		Composite video requires 4 Pole Adapter
Audio outputs:	TRS connector — 3.5 mm jack, HDMI		
Storage:	Secure DigitalSD / MMC / SDIO		Micro Secure Digital / MicroSD
Network	None	10/100 wired Ethernet RJ45	
Low-level peripherals:	26 General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface Bus (SPI), I2C, I2S, Universal asynchronous receiver/transmitter (UART)		40 General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface Bus (SPI), I2C, I2S, I2C IDC Pins, Universal asynchronous receiver/transmitter (UART)
Power ratings:	300 mA, (1.5 W)	700 mA, (3.5 W)	˜650 mA, (3.0 W)[3]
Size:	(85.0x56.0x15) mm	(85.0 x 56.0 x 17) mm	

Tabela 3.1: Strojne lastnosti Raspberry Pi [2].

je priključena na izhod. Primer knjižnice, ki jo nam ponuja skupnost, je *wiringPi* in je na voljo za C/C++ programski jezik. Obstajajo tudi ovojniki (angl. *wrappers*) za druge programske jezike. V tej diplomski nalogi bomo opisali uporabo ovojnika knjižnice za Python.

3.2 Vhodno/izhodna ploščica

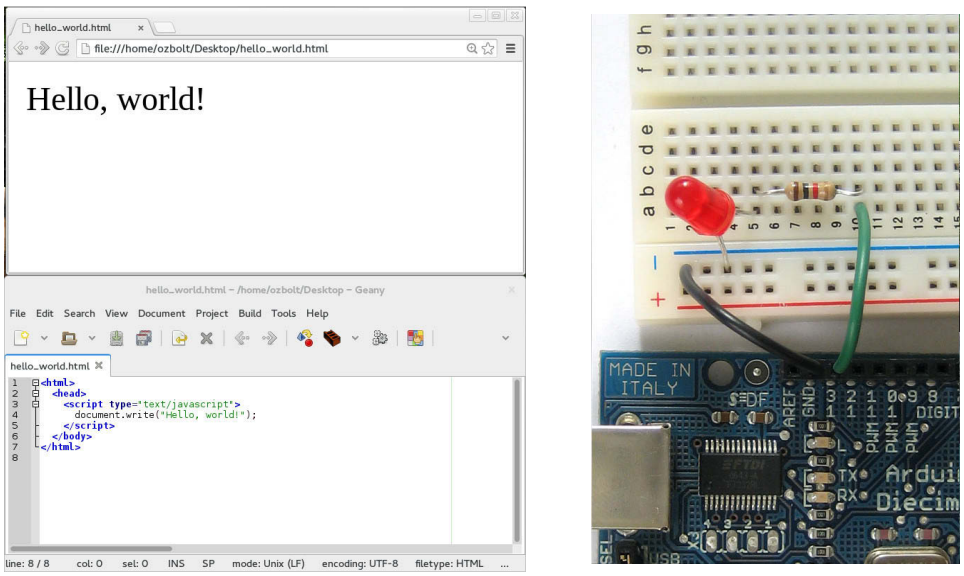


Slika 3.2: Slika razširitvene vhodno/izhodne ploščice v uporabi.

Dve izmed področij, kjer je ploščica Raspberry Pi uporabna, sta:

- programiranje v zbirniku (RISC ARM),
- programiranje vgrajenih naprav (zaradi svoje velikosti se mnogokrat vgrajuje v večje sisteme).

Prvi program, ki je napisan v vsakem priročniku za začetnike programiranja, je Pozdrav svetu (angl. *Hello World*, slika 3.3a). Pozdrav svetu je v svetu vgrajene elektronike ali programiranja v zbirniku dosti drugačen, saj izpisovanje besedila pogosto ni na voljo. Njegov nadomestek je lahko naprimer prižiganje LED diod ali branje stanja gumbov na



(a) Pozdrav svetu v brskalniku, spodaj koda.

(b) Pozdrav svetu z LED diodo.

Slika 3.3: Pozdravi svetu v različnih okoljih.

napravi (slika 3.3b). Ob pogledu na sliko 3.1 vidimo, da ploščica LED diode sicer ima, ki pa žal niso programabilna. Na ploščici so le zato, da sporočajo stanje naprave.

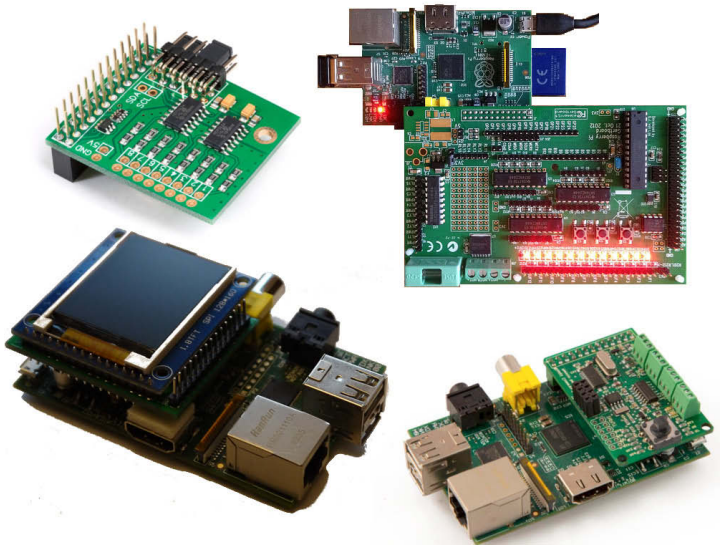
V izogib takim problemom obstajajo dodatne razširitvene ploščice (slika 3.4). Kljub temu smo se odločili, da ploščico, ki bi nam omogočila lažje programiranje izdelamo sami. Slika (3.2) prikazuje gumbe, večji zvočnik in štiri led diode ob gumbih, ki se nahajajo na razširitveni vhodno/izhodni ploščici. Njena uporaba je enostavna. Na ploščici se poleg omenjene periferije nahaja še senzor svetlobe.

Odprtokodno orodje, ki smo ga uporabili za izdelavo strojnega dela te diplomske naloge se imenuje KiCAD. Dodatne informacije o vezju lahko dobite v dodatku A.

3.3 Operacijski sistemi

Operacijski sistem (OS) je tako kot vsaka stvar, ki teče na računalniku, program. Njegova naloga je nadzor drugih programov, da se ti obnašajo po vnaprej določenih pravilih. Najbolj znan OS je Microsoft Windows, vendar obstajajo alternative. Ker želimo bralcu predstaviti čimbolj odprto rešitev, smo v naši rešitvi uporabili Linux OS, ki ima podporo za Raspberry Pi že vgrajeno.

Obstaja več Linux distribucij, ki v koordinaciji z ostalimi orodji iz GNU (angl. *GNU*



Slika 3.4: Razširitvene ploščice za Raspberry Pi.

is not Unix) kompilacije ustvarijo delujoč sistem. To nam omogoča, da lahko uporabimo široko paleto že obstoječih programov, ki nam bodo omogočili učenje. Najbolj znani distribuciji za uporabo v koordinaciji z Raspberry Pi sta Raspberian in Arch Linux ARM. Slednji velja za distribucijo z bolj posodobljenimi programi (angl. *bleeding edge*), medtem ko je Raspberian zgrajen na osnovi Debiana, kar prinaša večjo zanesljivost programov.

Za potrebe te naloge bi lahko uporabili katerikoli sistem. Zaradi zanesljivosti smo se odločili za distribucijo Raspberian.

3.4 Orodje za programiranje

Ko bo nekdo želel program naložiti na mini računalnik in ga pognati, bo to moral storiti preko določenega protokola. Največkrat se to izvaja preko protokola SSH, s katerim rokujemo preko ukazne vrstice. To za začetnike ni najprijaznejša rešitev, zato potrebujemo boljšo.

Danes več programskih orodij (IDE, primer na sliki 3.6) uporablja množico skript, ki v ozadju sicer uporabljajo SSH, vendar je to zakrito v skupek preprostih gumbov. Problem te izvedbe je vezava na določeno programsko orodje (angl. *vendor lock-in*). Zato smo se odločili za tretjo rešitev.

Rešitev je lasten HTTP strežnik nameščen na ploščici Raspberry Pi. Do strežniškega

3.3V	1	2	5V
I2C0 SDA	3	4	DNC
I2C0 SCL	5	6	GROUND
GPIO4	7	8	UART TXD
DNC	9	10	UART RXD
GPIO 17	11	12	GPIO 18
GPIO 21	13	14	DNC
GPIO 22	15	16	GPIO 23
DNC	17	18	GPIO 24
SP10 MOSI	19	20	DNC
SP10 MISO	21	22	GPIO 25
SP10 SCLK	23	24	SP10 CE0 N
DNC	25	26	SP10 CE1 N

Slika 3.5: Razpored GPIO pinov. DNC pomeni "do not connect". GPIO pin 19 podpira PWM modulacijo, 14 in 15 se lahko uporabita za UART komunikacijo, I^2C komunikacija je na 2 in 3, SPI komunikacija pa je možna po pinih na spodnjem delu plošče.

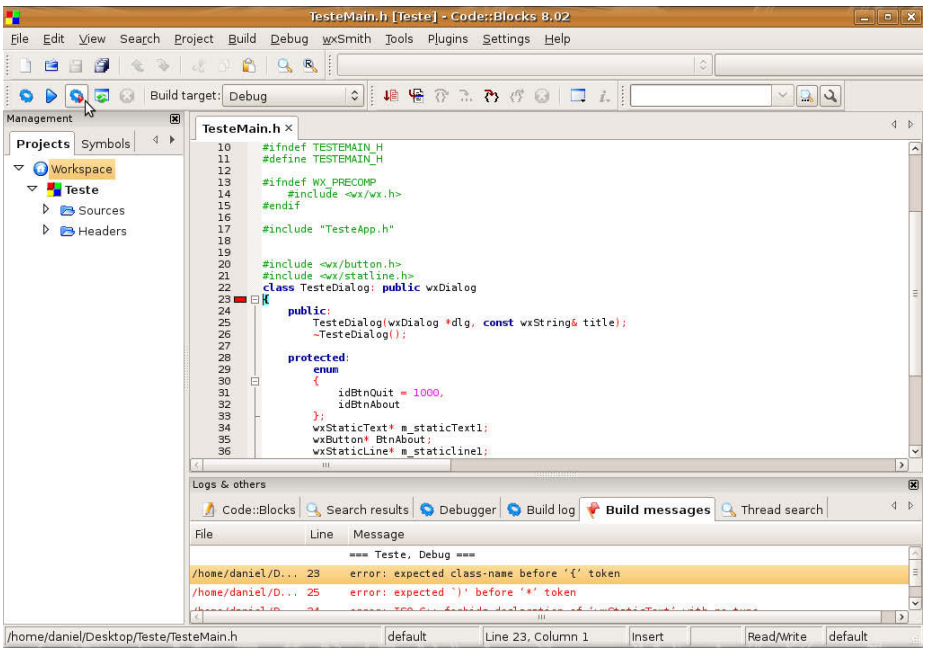
vmesnika bomo dostopali preko brskalnika. Za boljši občutek pri njegovi uporabi so uporabljene animacije. Le-te naj bi dale uporabniku lagodnejši občutek pri brskanju.

3.4.1 Nalaganje datoteke

Nalaganje nove datoteke poteka preko začetne strani (slika 3.7). Tu naložimo izvorno kodo programa, standardni vhod in argumente v ukazni vrstici. Vsako izmed teh treh stvari lahko naložimo preko tekstovnega vnosa ali kot datoteko. Če izvorno kodo nalagamo preko tekstovnega vnosa, moramo dodatno določiti tudi jezik izvorne kode. Drugi način nalaganja kode pa je, da si izberemo enega izmed primerov nalog v zato namenjeni strani (glej sliko 3.8).

3.4.2 Poganjanje datoteke

V primeru, da so v datoteki napake in je njeno nalaganje neuspešno, se na desni strani prikaže izpis prevajalnika in napake (glej sliko 3.9). Ob pritisku spodnje desne tipke lahko ponovimo in naložimo popravljeno verzijo. V primeru, da se je koda uspešno prevedla, se omogoči gumb *Start* in s pritiskom nanj zaženemo program. Slika 3.10 prikazuje primer poganjanja datoteke. Ker program bere argumente ukazne vrstice (ukaz `args[i]`) in standardni vhod (ukaz `scanf()`), mu pri izvajanju podamo naslednje vhodne podatke:



Slika 3.6: Programsko okolje za pomoč pri programiranju Code::Blocks.

Argument ukazne vrstice:

```
"Katarina Brencic"
```

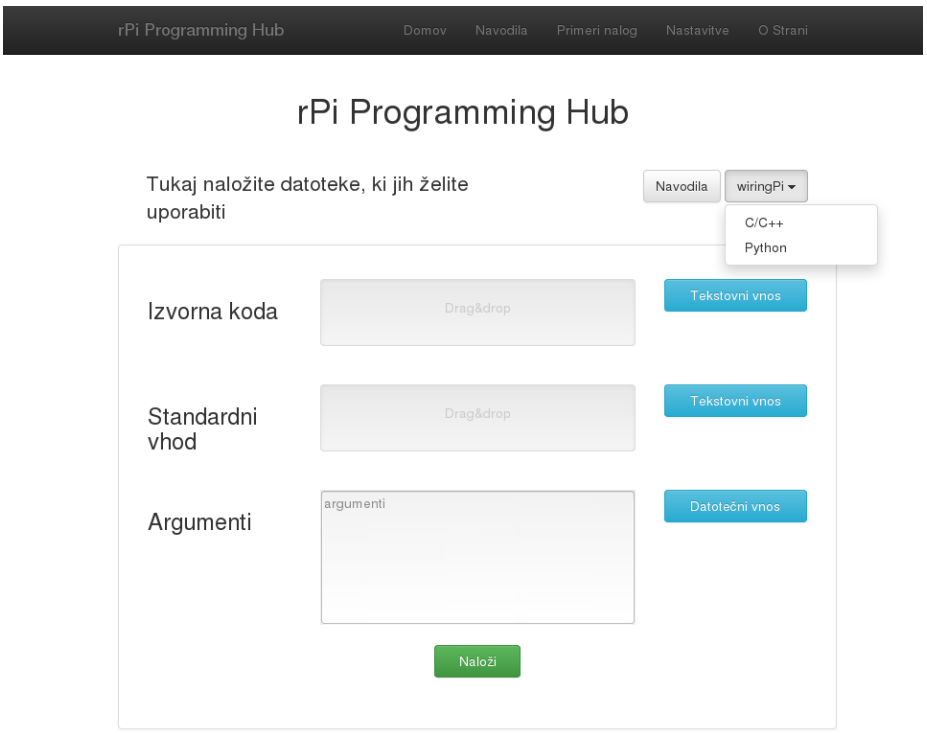
Standardni vhod:

```
Menegatti
22
0x2A
```

Poganjanje je privzeto omejeno na deset sekund. Po tem času se program zasilno zaustavi. Če želimo to ali kakšne druge nastavitve spreminjati, obiščemo stran z nastavitvami (slika 3.11).

3.4.3 Tehnične podrobnosti

Strežnik, ki je napisan v programskem jeziku Python, je sestavljen iz treh datotek. V prvi datoteki se nahajajo opisi odzivov na HTTP zahteve. Vsi URL naslovi, ki vrnejo status 200, so: `/`, `/settings`, `/examples`, `/about` in `/reference`. Zadnji dve strani sta statični in nimata večjega pomena. V datoteki se nahaja še razred

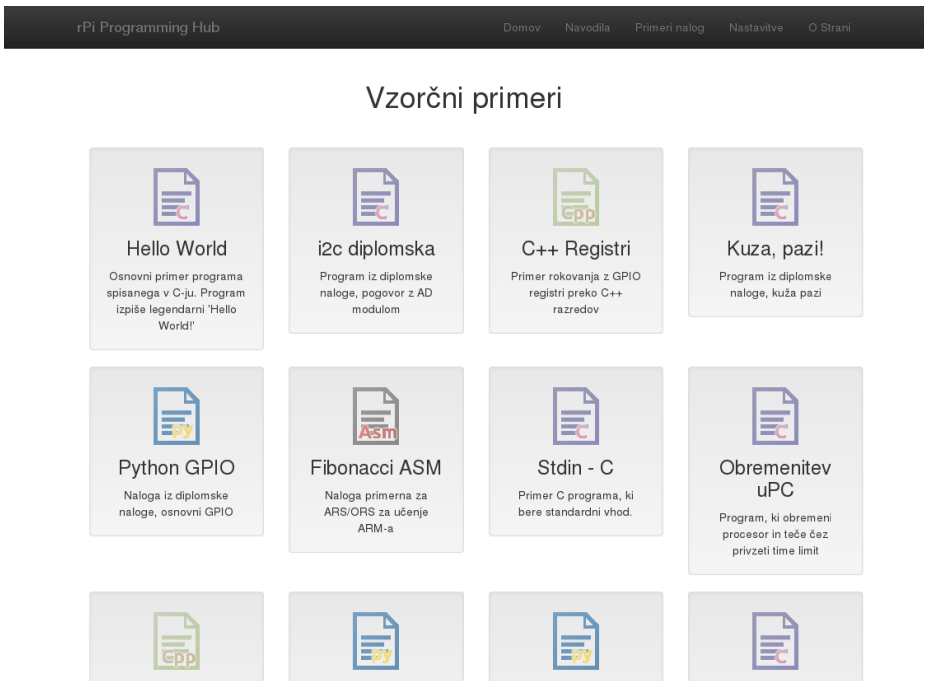


Slika 3.7: Začetna stran, ko dostopamo do HTTP strežnika preko modernega brskalnika. Stran deluje le v brskalnikih s podporo *HTML5 File API*.

Settings, ki hrani nastavitve (posnetek strani prikazuje slika 3.11) ter funkcija, ki sestavi stran s primeri (na sliki 3.8 je prikazan osnovni primer delovanja).

V drugi datoteki se nahaja razred Colorizer, katerega namen je polepšanje raznih delov strani. Izvorno kodo pobarvamo s pomočjo knjižnice *pygments*, ki barvno označuje izvorne kode, ki jih uporabnik naloži. Poročilo o morebitnih napakah pri kompilaciji, ki ga prevajalnik `gcc` vrne lahko od verzije `gcc 4.9` naprej prav tako obarvamo (uporaba zastavice `-fdiagnostics-color=always`). Vendar pa ta vrne le *esc* kode barv [3], tako da je druga naloga tega razreda prenos informacije o barvi iz *esc* kode do html/css stila.

Tretja datoteka vsebuje same razrede, ki zgradijo in poganjajo datoteke. Glavni razred Runner vsebuje vse možne informacije in metode o trenutni datoteki, ki jo je uporabnik naložil. Runner vsebuje podrazreda RunnerThread in ThreadData. Prvi razširi

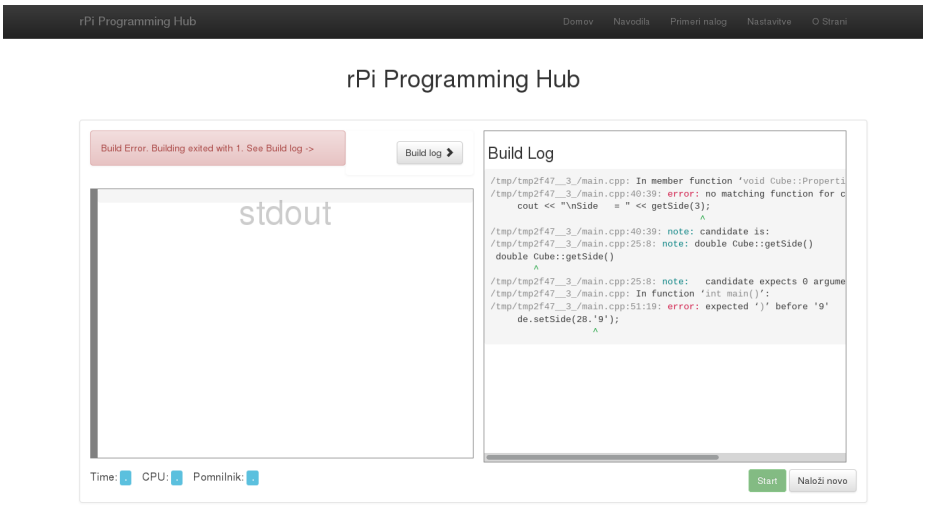


Slika 3.8: Podstran spletnega strežnika s seznamom primerov nalog.

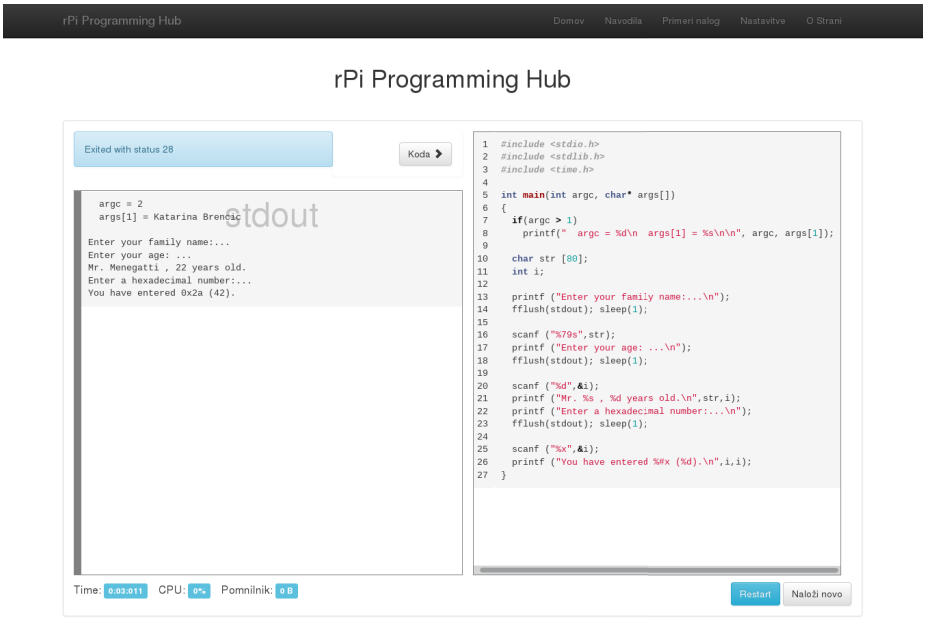
vgrajeni razred Thread. Ko poženemo to nit, se podatki o izhodu programa beležijo v drugi podrazred, ta pa te spremembe na zahtevo uporabnika sporoča naprej uporabniku. Za lažjo komunikacijo med HTTP strežnikom in razredom Runner ima slednji vgrajenih še več metod. Prav tako ne manjka podpora za nasilno končanje programa, za časovno omejitev izvajanja, za beleženje podatkov o uporabi procesorja ter pomnilnika ipd.

Python strežnik za izvajanje uporablja knjižnice *flask* in *jinja2*. Verzija popularnega programskega jezika Python je 3.4.

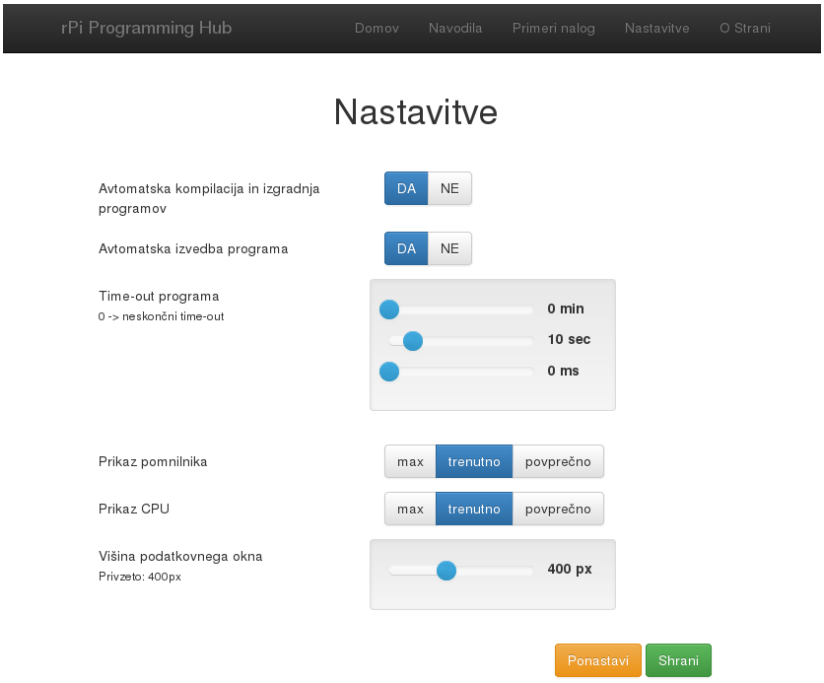
Klient je napisan s pomočjo knjižnice *jQuery*, ki jo uporabljamo za vse AJAX operacije, ki se izvedejo ali ob naložitvi datoteke ali ob njenem poganjanju (denimo bloka stdout na sliki 3.10). Skrbi tudi za preverjanje pravilnosti vnosa, prenos sprememb nastavitev in mnogo drugih manjših opravil. Za izgled strani uporabljamo knjižnico *Boostrap*, animacije pa so izvedene s pomočjo standarda CSS3.



Slika 3.9: Napaka pri prevajanju datoteke.



Slika 3.10: Primer poganjanja datoteke.



Slika 3.11: Nastavitve.

4 Vzorčne izobraževalne naloge

V pričujočem poglavju je nanizanih več vzorčnih izobraževalnih nalog. Te segajo od osnovnega prižiganja LED diod in branja stanj gumbov, preko programiranja v drugih jezikih, natančneje v jeziku Python in ARM zbirniku, do komunikacije preko I^2C protokola in uporabe PWM izhoda za generacijo zvoka.

4.1 GPIO nadzor brez uporabe knjižnice *wiringPi*

4.1.1 Opis naloge

Naloga se glasi: *Implementiraj prižiganje LED diod glede na stanje gumbov. Pri tem knjižnice wiringPi nimaš na voljo.*

Namen naloge je seznaniti se s procesorsko arhitekturo na strojni opremi in razširitveni vhodno/izhodni ploščici. Nalogo lahko logično razdelimo na tri dele. Prvi del je inicializacija strojne opreme in operacijskega sistema. Drugi del je realizacija funkcij, ki bodo omogočale branje in pisanje različnih stanj diod in gumbov. Tretji del pa je sama logika programa. Glavni vir pri reševanju bo specifikacija procesorja [4], vezava GPIO pinov na ploščici (slika 3.5) in shema V/I ploščice (slika A.1).

4.1.2 Rešitev naloge

Koda 4.1: setup(void)

```
1  #define PERI_BASE 0x20000000
2  #define GPIO_OFFSET 0x00100000
3  #define BLOCK_SIZE (4*1024)
4  static volatile uint32_t *gpio ;
5
6  void setup (void) {
7    int fd ;
8    fd = open ("/dev/mem", O_RDWR | O_SYNC );
9    if (fd < 0) {
10     printf("Ne uspem odpreti /dev/mem , napaka: %s\n", strerror (errno));
11     exit(1);
12   }
13   gpio = (uint32_t *)mmap(0, BLOCK_SIZE,
14                           PROT_READ|PROT_WRITE,
15                           MAP_SHARED, fd,
16                           PERI_BASE+GPIO_OFFSET) ;
17   if (gpio < 0) {
18     printf("Funkcija mmap (GPIO) neuspesna , napaka %s\n",
19             strerror (errno));
20     exit(1);
21   }
22 }
```

Binarno pisanje in branje sta operaciji, ki ju ponavadi v specifikacijah najdemo pod poglavjem *General Purpose I/O*. Poglavje se začne na strani 89 v knjigi [4]. Funkcija setup (koda 4.1) opravi začetno inicializacijo za kasnejše uspešno pisanje in branje registrov.

V programskem jeziku C je pisanje na določeni naslov v pomnilniku izvedeno s kazalci. Denimo, da bi želeli na naslov $0x1234$ napisati vrednost 1984. To bi izvedli na naslednji način:

```
*(0x1234) = 1984;
```

Le-ta se lahko pravilno prevede, a v večjih operacijskih sistemih, kakršen je tudi Linux,

ne bo deloval. Moramo si namreč rezervirati pravice za pisanje in branje na pomnilniške naslove. Poskus rezervacije se skriva v vrsticah 8 in 13.

V Linux-u imamo znakovno napravo /dev/mem, ki predstavlja sliko pomnilnika v računalniku. Kot tako napravo jo lahko odpremo s sistemskim klicem open() (vrstica 8). Kot parametre priložimo O_ RWDR, ki nam omogoči bralni/pisalni dostop, O_ SYNC pa OS-u naroči, naj bo pisanje sinhrono. To pomeni, da bo proces počakal z nadaljevanjem izvajanja, dokler zapis ne bo končan.

Naslednja zanimiva vrstica je vrstica 13. Klic mmap() sporoči OS-u, da program želi dostop do nekega pomnilniškega prostora, na katerega bo kazal deskriptor datoteke. Le tega smo pridobili s klicem open(). Rezerviramo si torej prostor velikosti BLOCK_ SIZE, na katerega lahko beremo in pišemo. Zastavica MAP_ SHARED pove, da se bodo spremembe dejansko shranjevale na napravo. Pomnilniški naslov, od katerega naprej si prilaščamo prostor, je določen s PERI_ BASE + GPIO_ OFFSET. PERI_ BASE je omenjen v poglavju 1.2.3 [4], GPIO_ OFFSET pa se ne nahaja v uradni dokumentaciji, najdemo ga lahko v knjižnici *wiringPi*.

Koda 4.2: setMode()

```
1  #define INPUT 0
2  #define OUTPUT  1
3  void setMode(int pin, int mode) {
4    int fSel, shift;
5
6    fSel = pin / 10;
7    shift = (pin % 10) * 3;
8
9    *(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift)) | ((mode & 0x7) << shift) ;
10 }
```

Naslednji sestavek izvorne kode je 4.2. Funkcija setMode določa namembnost nekega pina. Naj bo to vhod, izhod ali kaj drugega.

Tabela T6-1 GPIO Register Assignment na strani 91 in naprej iz [4] opisuje registre GPIO Function Select Registers (GPFSELn). Ta nam pove dve pomembni stvari:

- V enem 32 bitnem pomnilniškem naslovu se nahaja prostor za nastavitve desetih naprav. Vsak pin ima dodeljene svoje tri bite, imenovane FSELn (denimo FSEL2

: [6-8]).

- Zatorej lahko vsak pin deluje na sedem načinov, pri katerem je v načinu izhod vrednost treh bitov 0b001, vhod pa 0b000.

Vzemimo sedaj vrstici, kjer določamo vrednosti obeh integer-jev. Iz zgornjih točk bi morali biti operaciji dodelitve vrednosti očitni. V zadnji vrstici bite registra postavimo tako, da se aktivira želeni način delovanja. Register je določen s gpio+fsel kazalcem. Cilj te ne ravno preproste vrstice je ta, da ohranimo vrednost bitov, ki jih ne želimo spreminjati, prepišemo pa samo tiste tri bite registra, ki so določeni s spremenljivko shift.

Koda 4.3: setOutput()

```
1  #define LOW  0
2  #define HIGH  1
3
4  void setOutput(int pin, int value) {
5    int regNo;
6    if (value == LOW)
7      regNo = pin < 32 ? 10:11;
8    else
9      regNo = pin < 32 ? 7: 8;
10
11   *(gpio + regNo) = 1 << (pin & 31);
12 }
```

Po določitvi smeri pinov, moramo prebrati nivoje napetosti na vhodnih pinih (gumbi) in določanje napetosti na izhodnih (LED diode). Posvetimo se prvemu. Zopet se obrnimo k podatkovnemu listu našega procesorja. V prvi tabeli omenjenega poglavja lahko najdemo nahajališče registrov GPSETn (registra številka 7 in 8) in GPCLRn (registra številka 10 in 11). Na strani 95 vidimo, da imajo ti registri namen nastavljanja pinov v visoko stanje (GPSETn) ali na napetost mase (GPCLRn). Vsak pin ima določen svoj bit pri vnosu, ker pa je registrov več kot 32, se pini, ki so večji od 31, prelijejo v naslednji register.

Iz zbranih informacij v prejšnjem odstavku dobimo izvor vseh vrstic v funkciji setOutput iz kode 4.3. Zanimiva razlika med to kodo in tisto iz 4.2 je ta, da smo se pri setMode

potrudili ohranjati vrednosti bitov v registru, za katere se nismo zanimali. V funkciji setOutput nam tega ni potrebno storiti, saj vrednosti registra niti ne moramo prebrati (in s tem ohranjati), saj so to izhodni registri.

Koda 4.4: readInput()

```
1 int readInput(int pin) {
2   int regNo = pin < 32 ? 13:14 ;
3   int value = *(gpio + regNo) & (1 << (pin & 31));
4   if (value != 0)
5     return HIGH ;
6   else
7     return LOW ;
8 }
```

Logično nadaljevanje po pisanju je branje. Iz določitve vrednosti spremenljivke regNo lahko po analogiji iz prejšnjih funkcij ugotovimo, da bomo brali iz registrov 13 in 14. Ti registri (stran 96 [4]) se imenujejo GPIO PIN LEVEL REGISTERS (GPLEVn) in so vhodni. Realizacija funkcije readInput je v kodi 4.4.

Ostala nam je le še glavna main funkcija, ki pa je napisana v odseku kode 4.5.

Koda 4.5: main()

```
1 int main (void) {
2   int buttons[4] = {18, 23, 24, 25};
3   int leds[4] = { 7, 8, 17, 27};
4   int i, pressed;
5   setup();
6   for(i=0;i<4;i++) {
7     setMode(buttons[i], INPUT);
8     setMode(leds[i], OUTPUT);
9   }
10  for (;;delay(500)) {
11    for(i=0;i<4;i++) {
12      pressed = readInput(buttons[i]);
13      setOutput(leds[i], pressed);
14    }
```

```
15   }
16   return 0 ;
17 }
```

Logike delovanja na tem mestu ne bomo razlagali, saj bi morala biti jasna tudi začetniku programiranja. Razlago pa vseeno potrebuje nekaj detajlov. Prvi naj bo izvor GPIO pinov za gumbe in diode. Te lahko določimo s pomočjo slike A.1. Drugo vprašanje je, kaj nam funkcija readInput vrne, ko je gumb pritisnjeno? Iz sheme lahko razberemo, da pritisnjeni gumbi električni potencial pina dvignejo na 3.3V (logična enica). Upori R5-R8 tu služijo kot Pull down upori, kar pomeni da vhod povlečejo k potencialu mase v času, ko tipka ni pritisnjena. S študijo sheme lahko ugotovimo tudi, da LED diode svetijo ob logični enici na izhodih 7, 8, 17 in 18. Zadnja stvar, ki še ni pojasnjena, pa je implementacija funkcije delay; le to lahko vidite v prilogi B.4.

4.2 GPIO nadzor z uporabo knjižnice *wiringPi*

4.2.1 Opis naloge

Naloga se glasi: *Prižigaj LED diode glede na stanje senzorja svetlobe. Na spremembo se odzovi z uporabo prekinitev. Za pomoč uporabi knjižnico wiringPi.*

Za spremembo od naloge iz poglavja 4.1 tokrat nimamo opraviti z branjem specifikacij strojne opreme, saj uporabljamo knjižnico, ki je to za nas že opravila. Tokrat bomo poleg sheme vhodno/izhodne ploščice (slika A.1) za pomoč pri programiranju uporabili tudi opis knjižnice *wiringPi* [5].

4.2.2 Rešitev naloge

Ker tokrat uporabljamo knjižnico *wiringPi*, moramo najprej preveriti, kakšne so označitve pinov v tej knjižnici. Dokumentacija knjižnice vsebuje tabelo s števili pinov (tabela 4.1). Razliko med številkami LED pinov z ali brez *wiringPi* lahko opazimo, če primerjamo kodi 4.5 in 4.6.

Koda 4.6: celotna druga naloga

```
1 #include <wiringPi.h>
2
3 #define LED0 10
```

WiringPi: GPIO Pin Numbering Tables

http://wiringpi.com/

P1: The Main GPIO connector							
WiringPi Pin	BCM GPIO	Ime	Header		Ime	BCM GPIO	WiringPi Pin
		3.3v	1	2	5v		
8	2	SDA	3	4	5v		
9	3	SCL	5	6	0v		
7	4	GPIO7	7	8	TxD	14	15
		0v	9	10	RxD	15	16
0	17	GPIO0	11	12	GPIO1	18	1
2	7	GPIO2	13	14	0v		
3	22	GPIO3	15	16	GPIO4	23	4
		3.3v	17	18	GPIO5	24	5
12	10	MOSI	19	20	0v		
13	9	MISO	21	22	GPIO6	25	6
14	11	SCLK	23	24	CE0	8	10
		0v	25	26	CE1	7	11
WiringPi Pin	BCM GPIO	Ime	Header		Ime	BCM GPIO	WiringPi Pin

Tabela 4.1: Zgornja tabela je razširitev tabele pinov (slika 3.5) z oznakami v knjižnici *wiringPi*. Številka verzija ploščice, ki jo uporabljamo, je dva. Detekcijo verzij *wiringPi* opravi samodejno.

```
4  #define LED1 11
5  #define LED2 0
6  #define LED3 2
7  #define LIGHT 7
8  #define DELAY_TIME 10 //milisekunde
9
10 void interruptHandlerUp (void) { temperature_sensor = HIGH; }
11 void interruptHandlerDown (void) { temperature_sensor = LOW; }
12 static volatile int temperature_sensor;
13
14 void main (void) {
15   wiringPiSetup () ;
16   pinMode (LED0, OUTPUT) ;
17   pinMode (LED1, OUTPUT) ;
18   pinMode (LED2, OUTPUT) ;
19   pinMode (LED3, OUTPUT) ;
20   pinMode (LIGHT, INPUT) ;
21
22   wiringPiISR (LIGHT, INT_EDGE_FALLING, &interruptHandlerUp) ;
23   wiringPiISR (LIGHT, INT_EDGE_RISING, &interruptHandlerDown) ;
24
25   for (;;) {
26     digitalWrite( LED0, temperature_sensor );
27     digitalWrite( LED1, temperature_sensor );
28     digitalWrite( LED2, temperature_sensor );
29     digitalWrite( LED3, temperature_sensor );
30     delay(DELAY_TIME);
31   }
32 }
```

V vrstici 12 opazimo zanimivo definicijo globalne spremenljivke, static volatile int. Static lastnost pomeni, da je spremenljivka privatna tej datoteki, oziroma da do nje ne moremo dostopati iz drugih datotek. To tu ni tako pomembno, a pri pisanju knjižnic je spremenljivke, ki so jasno namenjene za interno uporabo, dobro označiti tako. Lastnost volatile povzroči, da prevajalnik ne izvaja optimizacij za spremenljivko. V tem primeru

bi lahko slab prevajalnik napačno predvideval, da prekinitveno servisna programa nista nikoli klicana in tako vzel spremenljivko temperature_ sensor za konstanto in izvedel temu primerne optimizacije. Sledita funkciji, ki se bosta klicali ob prekinitvah. Prvi trije ukazi v funkciji main, nam pokažejo kako inicializiramo knjižnico *wiringPi* in kako nastavimo smeri pinov. *WiringPi* nam omogoča tudi preprost način nadzora prekinitev. Za spremembo namembnosti funkcije v prekinitveno servisni program le pokličemo funkcijo wiringPiISR. Na koncu naloge v neskončni zanki zapišemo stanje temperaturnega senzorja tako, da ob detekciji pomanjkanja svetlobe prostor osvetlimo, v nasprotnem primeru pa ne.

Dobra praksa pisanja prekinitveno servisnih programov (PSP) je ta, da jih poskušamo ohranjati čim bolj kratke in jedrnate. Daljše prekinitve delovanja procesa v časovno kritičnih sistemih mnogokrat privedejo do neželenih posledic. Zato dejansko pisanje po registrih opravljamo v zanki, medtem ko PSP samo postavlja vrednost zastavice.

Opis delovanja temperaturnega senzorja je v dodatku A.3.

4.3 Igranje pesmice z uporabo knjižnice *pthread*

4.3.1 Opis naloge

Naloga se glasi: *S pomočjo gumbov in zvočnika omogoči igranje treh tonov, ki jih potrebujemo, da zaigramo pesmico Kuža pazi. Namesto knjižnice wiringPi, uporabi knjižnico pthread. Za tone uporabi frekvence* 531, 596 in $669Hz$.

Za realizacijo naloge bomo uporabili funkcije setup, setMode, readInput in setOutput iz poglavja 4.1. Za generacijo zvoka bomo uporabili knjižnico *pthread*. Zopet se bomo za poizvedbo GPIO pinov obrnili na shemo vhodno/izhodne ploščice (slika A.1). Več informacij o zvočniku najdemo v [6]. Izbor uporabljenih frekvenc je razložen v dodatku B.1.

4.3.2 Rešitev naloge

Zvok je fizikalna sprememba tlaka v prostoru in času, za katero smo skozi evolucijo razvili sposobnost zaznavanja. To spremembo tlaka lahko z membrano realiziramo tako, da jo periodično nihamo in s tem povzročamo nadtlake in podtlake. Ti potujejo po prostoru in jih človeško uho zazna, če je frekvenca nihanja znotraj intervala med 20 in 20000 Herzi. Zvočnik nadziramo na naslednji način:

1. Programsko določimo napetost na izbranem pinu.

2. Razlika v napetosti med pinom in maso povzroči premik električnih nosilcev (električni tok). Ta steče čez vse elemente, ki so vezani med pinom in maso.

3. Na tej poti v zvočniku imamo vezano tuljavico. Sprememba toka čez tuljavico ustvari magnetno polje.

4. Membrana v zvočniku je postavljena blizu tuljavice in je narejena na tak način, da se odziva na smer magnetnega polja.

Za večjo možnost ponovne uporabe v nadaljevanju napisane kode, bomo to funkcionalnost implementirali kot neodvisno nit v programu. S tem imamo še vedno možnost, da program ob izvajanju opravlja tudi druga dela.

Koda 4.7: Glasba, pthread

```
1  static int frequency;
2  static int pin;
3  static pthread_t threadptr;
4
5  void *music_thread( void * bv ) {
6    int halfPeriod ;
7    for (;;) {
8      halfPeriod = 500000 / frequency ;
9      setOutput (pin, HIGH) ;
10     delayMicroseconds (halfPeriod) ;
11     setOutput (pin, LOW) ;
12     delayMicroseconds (halfPeriod) ;
13   }
14   return NULL ;
15 }
```

V odseku kode 4.7 je glavni del programa: na vsako polovico periode spreminjamo stanje izhoda in to izvajamo v svoji niti. Kar nam še manjka je inicializacija pina in niti, zagon le-te, dinamična nastavitev frekvence in main funkcija, ki nam bo omogočila zaigrati pesmico. Koda je brez dodatnega komentarja podana v odseku kode 4.8.

Koda 4.8: Glasba, pthread drugič

```
1  void music_init (int pin) {
2    pthread_t myThread ;
3    frequency = 0;
4    setMode(pin, OUTPUT) ;
5    pthread_create (&myThread, NULL, music_thread, NULL) ;
6    threadptr = myThread ;
7  }
8
9  void music_frequency(int freq) {
10   frequency = freq;
11 }
12
13 void music_stop ( void ) {
14   pthread_cancel (threadptr) ;
15   pthread_join (threadptr, NULL) ;
16 }
17
18 int scale [3] = { 531, 596, 669 } ;
19 int swtch [4] = { SW_0, SW_1, SW_2, SW_3 } ;
20
21 void main( void ) {
22   int i ;
23   setup () ;
24   music_init (SPEAKER);
25   for(i = 0; i<4; i++)
26     setMode(swtch[i], INPUT);
27
28   for (;;) {
29     for (i = 0 ; i < 3 ; i++) {
30       delay (10) ;
31       if (readInput(swtch[i]) == HIGH) {
32         music_frequency (scale [i]) ;
33         break;
34       }
```

```
35      }
36      if (readInput(swtch[3]) == HIGH) {
37        music_stop() ;
38        break;
39      }
40    }
41  }
```

4.4 Igranje pesmice s pulzno širinsko modulacijo (PWM)

4.4.1 Opis naloge

Naloga se glasi: *S pomočjo gumbov in zvočnika omogoči igranje treh tonov, ki jih potrebujemo za igranje pesmice Kuža pazi. Za generacijo tonov uporabi strojno opremo, ki jo ima na voljo procesor na Raspberry Pi. Uporabi iste frekvence kot pri prejšnji nalogi.*

Za realizacijo naloge bomo dodelali funkcijo setup iz poglavja 4.1, za generacijo zvoka pa bomo uporabili pulzno širinsko modulacijo. Naloga bo podobno kot prva zahtevala veliko vpogledov v specifikacijo procesorja [4]. Ker je naloga zelo obširna, bo del povezovanja interne ure procesorja in PWM periferije rešen vnaprej.

4.4.2 Rešitev naloge

Pulzno širinska modulacija (angl.*Pulse Width Modulation, PWM*) je drugi naziv za tisto, kar smo naredili s pomočjo niti v prejšnji nalogi. Gre torej za periodično spreminjanje izhoda v pravokotni obliki. Lastno ime je dobila zaradi svoje izjemne uporabnosti v elektrotehniki. Ker omogoča širok nabor aplikacij, je danes PWM vgrajena v večino modernih procesorskih arhiterkur pričemer ARM ni izjema.

Uporaba niti iz prejšnjega poglavja zaseda procesorski čas, ki bi ga lahko uporabili veliko bolje. Mnogi procesorji v manjših in cenejših sistemih pa niti niso dovolj hitri za višje frekvence preklapljanja. To je zadosten razlog, da si PWM zasluži svojo nalogo.

Koda 4.9: Glasba - PWM inicializacija

```
1 #define PWM_OFFSET 0x0020C000
2 #define CLOCK_OFFSET 0x00101000
3 ...
```

	PWM0	PWM1
GPIO 12	Alt fun 0	
GPIO 13		Alt fun 0
GPIO 18	Alt fun 5	
GPIO 19		Alt fun 5
GPIO 40	Alt fun 0	
GPIO 41		Alt fun 0
...	...	...

Tabela 4.2: PWM na GPIO pinih procesorja na Raspberry Pi. Omogočimo ga kot alternativno funkcijo na določenih GPIO pinih. Izmed teh je samo pin 18 vezan na dostopno letvico, na katero lahko priključujemo zunanje naprave.

```
4  pwm = (uint32_t *)mmap(0, BLOCK_SIZE,
5                         PROT_READ|PROT_WRITE,
6                         MAP_SHARED, fd,
7                         PERI_BASE+PWM_OFFSET) ;
8  clk = (uint32_t *)mmap(0, BLOCK_SIZE,
9                         PROT_READ|PROT_WRITE,
10                        MAP_SHARED, fd,
11                        PERI_BASE+CLOCK_OFFSET) ;
12 ...
```

Odsek izvorne kode 4.9 je dopolnitev kode 4.1. Pri tej dopolnitvi smo iz podatkovnega lista izbrskali lokacijo PWM registrov in omogočili njihovo spreminjanje. Poleg tega smo omogočili tudi spreminjanje nastavitev ure, katere potrebujemo za PWM.

Koda 4.10: Glasba - PWM setPwmMode()

```
1  #define PWM_CONTROL 0
2  #define PWM_RANGE 4
3  #define PWM_MS_MODE 0x0080
4  #define PWM_ENABLE 0x0001
5  #define FSEL_ALT0  0b100
6
```

```
7  setupPwmMode(int pin) {
8    int fSel, shift, alt;
9    ...
10   alt = FSEL_ALT0;
11   *(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift)) | (alt << shift) ;
12   delayMicroseconds (110) ;
13   *(pwm + PWM_CONTROL) = PWM_ENABLE | PWM_MS_MODE ;
14   *(pwm + PWM_RANGE) = 1024;
15   setPwmClk(32);
16   delayMicroseconds (10) ;
17 }
```

Kot smo že povedali, imajo nekateri procesorji vgrajeno podporo za PWM in med njimi je tudi BCM2835. Vendar ta podpora ni dodana na vseh izhodnih pinih na dosegljivi letvici. Zvočnik je, gledano shemo, povezan na pin 18 (glej tabelo 4.2 [4]). Isti pin je povezan na strojni del procesorja, ki omogoča PWM (označen s PWM0).

V kodi 4.10 izvedemo več korakov. Za to, da omogočimo PWM, je potrebno funkcijo pina 18 nastaviti na ALT_ FUNC_ 0 (tabela 4.2). Nato ga omogočimo v registru PWM_ CONTROL in dodamo še nastavitev MS_ MODE, ki naredi delovanje modula preprostejše [4]. Namen funkcije setPwmClk (izsek 4.10) je, da omogoči povezavo interne ure (katera frekvenca je 19.2MHz) s PWM modulom. Delilnik 32 nam prinese osnovno uro za PWM hitrosti

$$\frac{19.2MHz}{32} = 600kHz\ .$$

Časovni zamiki v kodi se pojavljajo, ker je zahteva procesorja, da ob nastavljanju nekaterih nastavitev počakamo določen čas. O spremenljivki int range pa bomo več povedali po odseku izvorne kode 4.11.

Koda 4.11: Nastavitev frekvence

```
1  #define PWM_DATA 5
2  void setPwmFreq(int freq) {
3    int range = 600000 / freq ;
4    *(pwm + PWM_RANGE) = range;
5    *(pwm + PWM_DATA) = freq / 2 ;
```

```
6 }
```

Pri PWM lahko nastavljamo tudi čas, ko je PWM signal v visokem stanju v primerjavi s časom, ko je v nizkem (angl. *Duty Cycle*). Igranje z nastavitvijo PWM_ DATA vrne različne barve tonov različnih intenzitet. Z drugim registrom, tj. PWM_ RANGE določimo frekvenco. Ta se izračuna z enačbama:

$$f_{base} = 600kHz \ ,$$
$$f_{speaker} = \frac{range}{f_{base}} \ .$$

Glavna funkcija je podobna kodam 4.8 in 4.12. Razlika je le v tem, da se kliče funkcije napisane v tem poglavju. Ta rešitev nam ne obremeni procesor.

4.5 Igranje pesmice z uporabo knjižice *wiringPi*

4.5.1 Opis naloge

Naloga se glasi: *S pomočjo gumbov in zvočnika omogoči igranje treh tonov, ki jih potrebujemo, da zaigramo pesmico Kuža pazi. Za generacijo tonov uporabi strojno opremo, ki jo ima procesor na ploščici Raspberry Pi. Tokrat uporabi funkcije v knjižnici wiringPi.*

4.5.2 Rešitev naloge

Tudi ta rešitev ne obremenjuje procesorja. Pri rešitvi moramo v grobem samo zamenjati imena funkcij v tiste, ki so implementirane v wiringPi. Dodatno delo nam predstavlja le vključitev funkcije PWM, katero izvedemo s klicem v vrstici 8 (koda 4.12).

Koda 4.12: wiringPi PWM

```
1 #include <wiringPi.h>
2 #define SPEAKER 1
3
4 void main( void ) {
5   int i ;
6   int scale [3] = { 531, 596, 669 } ;
7   int swtch [4] = { SW_0, SW_1, SW_2, SW_3 } ;
8   wiringPiSetup () ;
```

```
9   pinMode(SPEAKER, PWM_OUTPUT);
10  for(i = 0; i<4; i++)
11    pinMode(swtch[i], INPUT);
12
13  for (;;) {
14    for (i = 0 ; i < 3 ; i++) {
15      delay (10) ;
16      if (digitalRead(swtch[i]) == HIGH) {
17        pwmToneWrite (SPEAKER, scale [i]) ;
18        break;
19      }
20    }
21    if (digitalRead(swtch[3]) == HIGH) {
22      pwmToneWrite(SPEAKER, 10) ; //cannot hear
23      break;
24    }
25  }
26 }
```

4.6 GPIO nadzor v Pythonu

4.6.1 Opis naloge

Naloga se glasi: *Na novo napiši nalogo iz poglavja 4.1, z razliko, da tu uporabiš jezik Python in knjižnico wiringPi za ta jezik.*

4.6.2 Rešitev naloge

Cilj naloge je prikaz Python ovojnika za že uporabljene funkcije iz prejšnjih nalog. Uporaba je za znalce tega programskega jezika izjemno intuitivna. Tako brez dodatne razlage prilagamo rešitev v odseku izvorne kode 4.13.

Koda 4.13: Python wiringPi

```
1 import wiringpi2 as wpi
2 buttons = [ 1, 4, 5, 6 ]
3 leds = [ 10,11, 0, 2 ]
```

```
4
5 wpi.wiringPiSetup()
6 for b in buttons:
7   wpi.pinMode(b, 0) #intput
8 for l in leds:
9   wpi.pinMode(l, 1) #output
10 while(True):
11   in_btn = [wpi.digitalRead(b) for b in buttons]
12   for i,l in enumerate(leds):
13     wpi.digitalWrite(l,i)
```

4.7 Fibonaccijevo zaporedje v čistem zbirnem jeziku

4.7.1 Opis naloge

Naloga se glasi: *V zbirniku arhitekture ARM napiši metodo, ki izračuna N-ti člen Fibonaccijevega zaporedja. Vhodni podatek N naj se prebere iz spremenljivke na pomnilniškem naslovu, na isti prostor pa naj se zapiše rešitev zbirniške rutine. Pri tem naj se vrednost registrov ohrani.*

4.7.2 Rešitev naloge

Koda 4.14 prikazuje rešitev naloge v zbirniku. Ukaza stmfd in ldmfd shranita in obnovita stanja registrov z uporabo sklada. Ukaza ldr in str opravljata operacije s pomnilnikom. Na voljo imamo še operacije mov za premik registra, cmp za postavljanje zastavic zero in carry, add in sub za aritmetične operacije in bxx za pogojne skoke.

Koda 4.14: Assembler naloga.

```
1         .align 2
2 N: .word 10
3 /* r0 -> N */
4 /* r0 <- fib(N) */
5 .start: stmfd sp!, {r1, r2, r3}
6         ldr r0, N
7         mov r1, #0
8         mov r2, #1
```

```
9           cmp r0, #2
10          blt .end_1
11  .loop: add r1, r1, r2
12          mov r3, r2
13          mov r2, r1
14          mov r1, r3
15          sub r0, r0, #1
16          cmp r0, #0
17          beq .end
18          b .loop
19  .end_1: mov r2, #1
20  .end: str r2, N
21          ldmfd sp!, {r1, r2, r3}
```

4.8 Fibonaccijevo zaporedje - izvedljiva koda v zbirniku

4.8.1 Opis Naloge

Naloga se glasi: *Kodo 4.14 zavij v C kodo tako, da jo bo mogoče prevesti in izvajati na Raspberry Pi. Zopet naj bo vhodni (zaporedna številka člena) in izhodni (vrednost tega člena) podatek shranjen v pomnilniškem prostoru N.*

4.8.2 Rešitev naloge

Ukaze v zbirniku je najlažje poganjati tako, da jih dodamo kot klice funkcije asm() (glej kodo 4.15). Vhodni podatek N je s tem definiran znotraj pomnilniškega prostora, ki je namenjen kodi in ne podatkom. Medtem, ko je branje podatka še uspešno, se pri poskusu pisanja v ta pomnilniški prostor zgodi napaka pri izvajanju. Linux javi napako *Segmentation fault*.

Koda 4.15: Assembler naloga v (ne)izvedljivi obliki

```
1  #include <stdlib.h>
2  int main() {
3  asm(" .align 2");
4  asm("N: .word 10");
5  asm("/* r0 -> N */");
```

```
asm("/* r0 <- fib(N) */");
asm(".start: stmfd sp!, {r1, r2, r3}");
asm(" ldr r0, N");
asm(" mov r1, #0");
...
asm(".end: str r2, N");
asm(" ldmfd sp!, {r1, r2, r3}");
return 0;
}
```

Problem rešimo tako, da označimo del s kodo kot kodo in del s podatki kot podatke (glej kodo 4.16). Drug problem se pojavi z direktnim naslavljanjem spremenljivke N. Ker se lahko zgodi, da se N nahaja daleč stran od kode, je potrebno naslavljanje (ukaza ldr in str v kodi 4.14) opraviti v dveh korakih. Datoteka z zbirnikom je podana v 4.16, datoteka s C kodo pa v 4.17

Koda 4.16: Assembler datoteka

```
.data
N: .word 10
.text
.global start
start: stmfd sp!, {r0, r1, r2, r3, lr}
        mov r1, #7
        ldr r0, =N
        ldr r0, [r0]
        mov r1, #0
        mov r2, #1
...
        b .loop
.end_1: mov r2, #1
.end: ldr r0, =N
        str r2, [r0]
        add r2, #1
        ldmfd sp!, {r0, r1, r2, r3, pc}
```

Koda 4.17: Assembler naloga - C main()

```
1 int main() {
2   start();
3   return 0;
4 }
```

Da je ta zbirniška koda izvedljiva, smo dopisali še funkcijo main. Da smo lahko klicali funkcijo start, smo morali v datoteki z zbirniško kodo (4.16) rutino označiti kot globalno. Pri povezovanju (angl. *linking*) moramo seveda podati obe datoteki. Primer izgradnje datoteke je v odseku kode 4.18.

Koda 4.18: Zaporedje GCC ukazov

```
1 gcc -c main.c -o main.o
2 gcc -c asm.S -o asm.o
3 gcc main.o asm.o -o a.run
```

S pomočjo orodij iz 3. poglavja ne moramo poganjati in razhroščevati zbirniških ukazov. Ta naloga je pokazala, kako narediti program izvedljiv, v dodatku B.2 pa je opisano kako lahko tako kodo kodo s klasičnimi metodami razhroščujemo s pomočjo orodja GDB.

4.9 I^2C komunikacija

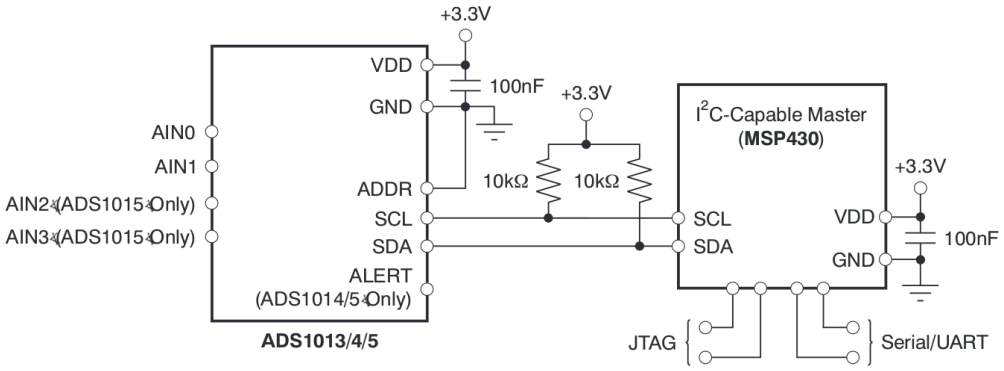
4.9.1 Opis Naloge

Naloga se glasi: *Na napravo je priključen A/D pretvornik TI ADS1013, do katerega komuniciramo preko vmesnika I^2C. Na analogni strani pretvornika je priključen temperaturni senzor, katerega stanje želimo brati v neskončni zanki.*

A/D pretvornik, ki je priključen, je pretvornik podjetja Texas Instrument, model ADS1013. Njegova vezava z okolico je prikazana na sliki 4.1, več informacij o čipu pa najdete v [8]. Vmesnik I^2C je opisan v prilogi B.3, uporabimo pa ga s pomočjo knjižnice wiringPi.

4.9.2 Rešitev naloge

Najprej moramo dobiti naslov naprave. Slika 4.1 nam prikazuje vezavo čipa ADS1013 na SDC in SDL priključka na Raspberry Pi (glej sliko 3.5).



Slika 4.1: Povezava na A/D modul preko I²C [8].

Na strani 14 podatkovnega lista [8] dobimo tabelo 4.3 iz katere razberemo naslov naprave (0x90).

ADDR pin	SLAVE address
GROUND	1001000
VCC	1001001
SDA	1001010
SCL	1001011

Tabela 4.3: Naslavljanje ADS1013

Naprej si poglejmo dosegljive registre na napravi. V [8] vidimo, da se za nas zanimiva registra *Conversion* in *Config* nahajata na naslovih 0 oziroma 1. Sledi natančnejši pregled najglobje vgnezdene zanke (vrstica 26 iz kode 4.19). Denimo, da si želimo, da naprava deluje v načinu hranjenja energije in mi od naprave le po nekem časovnem zamiku periodično zahtevamo pretvorbo analognega signala v digitalni.

Za dosego tega moramo ob zahtevi po pretvorbi počakati, da je le ta končana. To informacijo za nas hrani najvišji bit *Config* registra. Sedaj imamo vse informacije, da spišemo uporaben program. Ta se z dodanimi komentarji nahaja v kodi 4.19.

Koda 4.19: I2C komunikacija

```
#include <wiringPiI2C.h>
#include <stdlib.h>
//naslov naprave
#define dID 0x90
//set convert on cmd
#define CFG_DAT 0x88
#define CFG_REG 0x01
#define CONV_REG 0x02

int main (void)
{
  int fd, err, transf;
  fd=wiringPiI2CSetup(dID);
  if(fd < 0)
    exit(1);
  err = wiringPiI2CWriteReg16 (fd, CFG_REG, CFG_DAT);
  if(err < 0)
    exit(1);

  while(1) {
    //set command to start conversion
    err = wiringPiI2CWriteReg16 (fd, CFG_REG, CFG_DAT);
    if(err < 0)
      exit(1);
    //wait until conversion is done
    while(1) {
      transf = wiringPiI2CReadReg16 (fd, CFG_REG);
      if(transf < 0)
        exit(0);
      //conversion done
      else if(transf >> 15)
        break;
    }
    //read data
```

```
35      transf = wiringPiI2CReadReg16 (fd, CONV_REG);
36      if(transf < 0)
37        exit(0);
38      delay(100);
39    }
40    return 0;
41  }
```

V prejšnjih nalogah mnogokrat nismo izvajali pravega preverjanja napak. Spodnja naloga le to vsebuje. S tem prilagamo nasvet, da bralec vsako nalogo dopolni s takim in podobnim preverjanjem.

5 Zaključek

V okviru diplomske naloge smo razvili razširitveno vhodno/izhodno ploščico, ki ponuja dodatne vhodno/izhodne elemente za pomoč pri učenju programiranja na ploščici RaspberryPi. Poleg tega smo razvili tudi orodje za programiranje Raspberry Pi, ki je popolnoma neodvisno od platforme, saj deluje preko HTML vmesnika. Glavni doprinos diplomskega dela je seznam vzorčnih nalog, ki bralca popeljejo v svet iskanja pravih vrstic v stotinah straneh podatkovnih listov različnih naprav in sledenja pravim linijam v shemah različnih elementov. Nato te informacije prelijemo v programsko kodo, ki je poleg svoje poučnosti tudi zabavna z igranjem pesmic, prižiganjem luči in odzivanjem na svetlobo okolice. Pri tem je bralec izpostavljen vsesplošno uporabni knjižnjici wiringPi, ki mu lahko delo zelo olajša. Podan je še primer nalog v programskih jezikih Python in ARM zbirnik.

Čeprav so orodja, ki jih uporabljamo v diplomski nalogi razvita do neke mere uporabnosti, pa se lahko razvoj nadaljuje v marsikatero smer. Prva stvar, ki jo lahko dopolnimo, je vhodno/izhodna razširitvena ploščica. Le ta bi za varnejše delo potrebovala varnostne mehanizme in druge izboljšave, ki so omenjene v dodatku A.4.

Prav tako lahko dopolnjujemo spletni vmesnik. Čeprav ta vsebuje osnovne operacije ki jih potrebujemo za programiranje, pa ne omogoča dodajanja različnih parametrov pri prevajanju in ne omogoča pavziranja pri poganjanju. To sta le dve manjši funkcionalnosti izmed mnogih, ki jih ponujajo večji IDE-ji. Kot manjkajoče v poglavju nalog lahko omenim naloge o SPI (angl. *SCSI Parallel Interface*) in/ali RS-232 komunikaciji. Slednji nista realizirani zaradi pomanjkanjanja strojne opreme, pa tudi zaradi obsežnosti. O teh komunikacijskih protokolih si lahko več preberete na spletu, implementacije na Raspberry Pi pa dobite na spletnih straneh projekta wiringPi [5].

Literatura

[1] P. Stakem, 16 bit Microprocessors, History and Architecture, PRRB Publishing, 2013.

[2] eLinux.org, RPi Hardware, http://elinux.org/RPi_Hardware, [Uporabljen 15.7.2014] (2014).

[3] Wikipedia, ANSI escape code, https://en.wikipedia.org/wiki/ANSI_escape_code, [Uporabljen 19.8.2014] (2014).

[4] B. Corporation, BCM2835 ARM Peripherals, 406 Science Park Milton Road Cambridge CB4 0WW, 2012.

[5] G. Henderson, Wiring Pi Reference, http://wiringpi.com/reference/, [Uporabljen 3.9.2014] (2014).

[6] P. F. group, Multicomp Speaker, http://www.farnell.com/datasheets/1768514.pdf, [Uporabljen 15.7.2014] (2012).

[7] S. S. Richard Stallman, Roland Pesch, Debugging with gdb, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, 2014.

[8] T. Instruments, ADS101x A/D Converter Datasheet, http://www.farnell.com/datasheets/1768514.pdf, [Uporabljen 15.7.2014] (2009).

[9] V. Semiconductors, TEMT6200FX01 light sensor Datasheet, http://www.vishay.com/docs/81317/temt6200.pdf, [Uporabljen 15.7.2014] (2014).

[10] Sparkfun, 3.3V TTL Levels, https://learn.sparkfun.com/tutorials/logic-levels/33-v-cmos-logic-levels, [Uporabljen 19.7.2014] (2014).

[11] E. DIN, 12464-1 licht und beleuchtung, beleuchtung von arbeitsstatten, teil 1: Arbeitstatten in innenraumen (2003).

[12] Wikipedia, Root mean square, https://en.wikipedia.org/wiki/Root_mean_square/, [Uporabljen 15.8.2014] (2014).

[13] Wikipedia, I^2C, http://en.wikipedia.org/wiki/I%C2%B2C, [Uporabljen 19.8.2014] (2014).

[14] N. Semiconductors, UM10204 I2C-bus specification and user manual, PO Box 80073 5600 KA Eindhoven, 2014.

A Vhodno/izhodna ploščica: tehnične podrobnosti

A.1 Shema ploščice

Shema na sliki A.1 prikazuje vezavo naprav, ki skupaj tvorijo pet digitalnih vhodov (štiri gumbi in senzor svetlobe) in izhodov (štiri LED diode in zvočnik) . Opazimo lahko tudi dodatne podaljške za vse tri vrste komunikacijskih vodil; RS232, I^2C in SPI. Poleg omenjenih je tu še LED dioda, ki sporoča status naprave. Podatke o povezavah med oznakami vhodov in napravami podajamo v tabeli A.1

LED Diode si svoje izhode delijo s SPI komunikacijo. Da se ob SPI komunikaciji izognemo utripanju LED diod izključimo njihovo napajanje, ki je izvedeno preko mostiča JP1.

A.2 Tiskano vezje ploščice

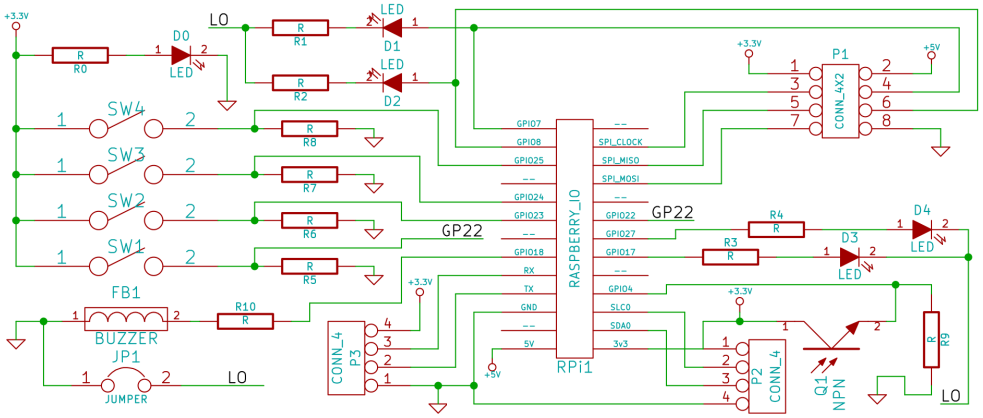
Oglejmo si razporeditev elementov po ploščici ter ostale lastnosti (skiki A.2 in A.3):

- GPIO letvica se nahaja na desni, kar pomeni, da ploščica prekrije spodnjo polovico Raspberry Pi. To se lepo vidi na sliki 3.2.

	gpio pin	wiringPi pin
LED0	7	10
LED1	8	11
LED2	17	0
LED3	27	2
SW1	22	3
SW2	23	4
SW3	24	5
SW4	25	6
BUZZER	18	1
LIGHT	4	7

Tabela A.1: Pini vhodno/izhodnih naprav.

- Zgoraj desno je z oznako D0 označena LED dioda, ki nam sporoča, če je naprava napajana.
- Pod njo imamo NPN foto tranzistor (model TEMT6200FX01 [9]), ki v tandemu z R9 tvori digitalni vhod.
- Levo se nahajata dva bolj osamljena pina. Na ta dva je zacinjen zvočnik [6].
- Malo pod sredino se od leve proti desni nahajajo 4 LED diode. Te so na pine vezane preko uporov R1-R4.
- Levo zgoraj nad diodami je mostič JP1. Ta skrbi za napajanje LED diod. Ob razklenitvi LED diode niso več povezane v električni krog.
- Spodaj so skoraj po celotni širini razporejeni gumbi SW1-SW4.
- Na levi strani se nahajajo 4 skupine letvic. Zgornja je podaljšek do I^2C vodila, naslednja spodaj do vodila RS232, spodnja, večja pa si prislužuje SPI vodilo.
- Poleg podpisa avtorja naloge (OM) se nahaja še pritrditvena luknja, ki služi stabilnosti. Natanko pod luknjo se nahaja večji kondenzator na Raspberry Pi, na katerega se razširitvena ploščica nasloni.



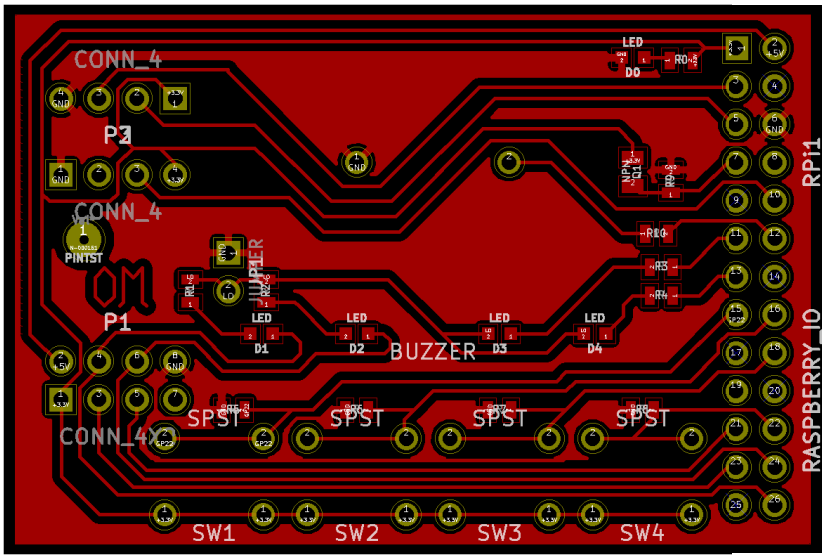
Slika A.1: Shema dodatne ploščice.

A.3 Izbor upora za NPN fototranzistor

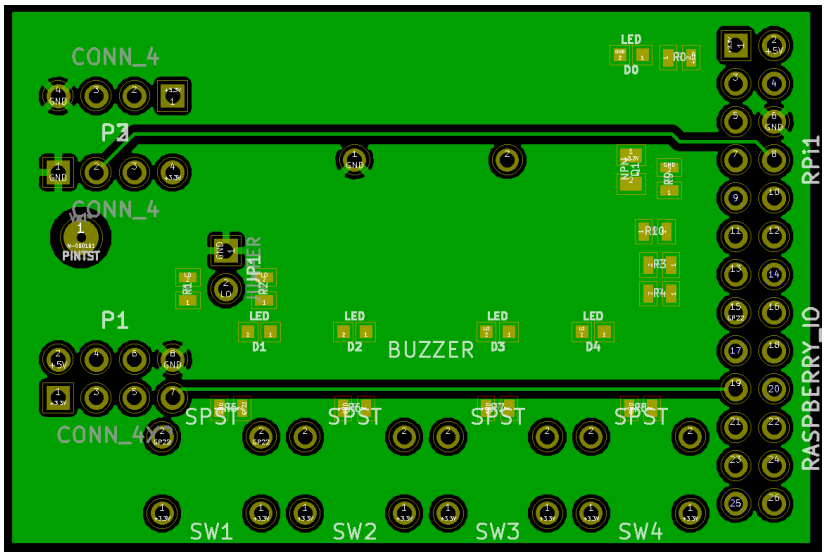
Poglejmo si, kako deluje Foto tranzistor. Uporabljen je tranzistor podjetja Vishay, model TEMT6200FX01 [9]. Ta je močno občutljiv na svetlobo iz svetlobnega spektra, na osvetlitev pa se odziva z vsiljevanjem toka. Ker je njegovo delovanje analogno po naravi, je za priključitev na digitalni vhod potrebna pravilna vezava in izbira dodatnih komponent. Gledano shemo na sliki A.1, NPN fototranzistor čez svoji sponki požene tok, ki je odvisen od napetosti na sponkah in od osvetlitve. Graf toka je prikazan na sliki A.4. Vzemimo za primer osvetlitev intenzitete 3000 luksov. Ta pri priključenih 3.3V povzroči tok velikosti 0,6mA. Ta steče preko upora R9 na maso. Pri prehodu toka čez upor se pojavi padec napetosti čez upor. Denimo, da je upor velikosti $30k\Omega$. Potem je padec napetosti

$$\delta U = 0.6mA \cdot 30k\Omega = 18V \ .$$

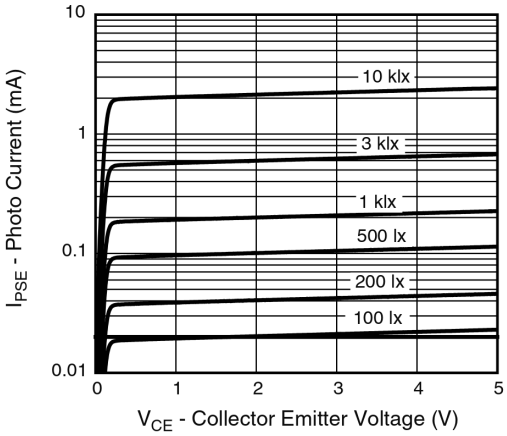
Seveda $18V$ ni realen odgovor, saj toliko napetosti niti nimamo na razpolago. Takoj, ko upor prevzame del napetosti, se napetost na tranzistorju zmanjša in s tem tudi tok. Po preteku zelo kratkega časovnega obdobja, se tok ustali v stabilni točki, ko je vsota obeh napetosti $3.3V$. Poskusimo določiti to stabilno točko: pri toku $0.1mA$ je napetost čez upor okoli $3V$, napetost na tranzistorju pa samo še kako desetinko volta. In ker je



Slika A.2: Zgornji del vezja.



Slika A.3: Spodnji del vezja.



Slika A.4: Graf vsiljenega toka preko kontaktov v odvisnosti od priključene napetosti ter osvetlitve NPN fototranzistorja [9].

vsota teh dveh napetosti čez palec enaka $3.3V$, vzamemo to kot dovolj dobro določitev stabilne točke. Pin je tako priključen na vmesno napetost, tako da je napetost na vhodu približno $3V$ in program nam pri branju vrne logično enico.

Naslednje vprašanje je, kaj se zgodi, ko na isti upor sveti samo še 100 luksov. Ponovno izračunajmo padec napetosti na uporu:

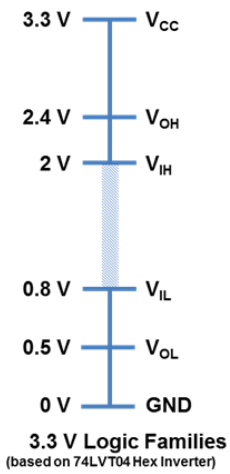
$$\delta U = 0.02mA \cdot 30k\Omega = 0.6V \ .$$

Tokrat napetost na uporu ne pade veliko. Ker je graf okoli točke $3V$ skorajda konstanten, se tok čez tranzistor spremenli za zanemarljivo spremembo in vmesna napetost ostane okoli $0.5V$. Glede na specifikacijo procesorja BCM2835 na Raspberry Pi (slika A.5) je $0.5V$ znotraj meja logične ničle.

Na ploščici ima R9 upornost zelo blizu $30k\Omega$, kar nam prinese lomno točko na okoli $50lx$. Po priporočilih Nemškega standarda DIN EN 12464-1 [11] mora biti osvetlitev na delovnem mestu od 500 do 1500 luksov. To pomeni, da bi v učilnici moral senzor svetlobe vračati logično enico, ko ne bo pokrit. Ko pa ga bomo zakrili pred svetlobo, pa bo senzor vračal logično 0.

A.4 Predlogi za izboljšave vezja

Porajajo se številne izboljšave, ki bi jih lahko realizirali v prihodnosti:



Slika A.5: TTL napetosti procesorja na piti [10].

Napajanje zvočnika je izpeljano direktno iz GPIO pina. Za daljšo življensko dobo vhodno/izhodnih pinov je priporočeno, da se nanje ne izvaja večjega pritiska. Zato bi bilo bolje med zvočnikom in pinom priključiti dodaten tranzistor, ki bi moč zvočniku dodeljeval kar direktno iz napajanja.

Komunikacijska vodila lahko povežemo z dodatnim A/D pretvornikom. Ta pretvornik je denimo povezan na temperaturni senzor z analognim izhodom. Tako lahko učenec brez priključevanja dodatnih naprav vadi delo z SPI ali I^2C vodilom.

Podporni kondenzator, ki naj bi skrbel za podporo na drugi strani V/I ploščice, je nekoliko zamaknjen glede na podporno luknjo v vezju. Ta naj bi bila točno nad kondenzatorjem. Manjši popravek bi napako hitro odpravil.

B Ostale priloge

B.1 Določanje tonov za pesmico

V želji po tem, da bi bile napisane funkcije čimbolj podobne primerljivim funkcijam v knjižnici *wiringPi*, smo funkcijo setPwmFreq spisali na tak način, da vzame celoštevilski argument. Seveda sosednji toni na resničnih glasbilih ne zasedajo samo tonov v obsegu frekvenc celih števil, ampak zavzemajo vse možne vmesne tone.

Glasbeni ton je določen s tem, koliko poltonov je oddaljen od osnovnega tona A pri $400Hz$. Če vemo, da sta sosednja tona določena s pomočjo geometrijskega zaporedja, lahko ton, oddaljen n poltonov določimo s pravilom

$$f_{ton} = f_0 \cdot \zeta^n \ .$$

Če ste se kdaj spraševali, zakaj oktava zveni tako, kot da bi bil to en sam ton in razloga še ne veste, je odgovor tak: frekvenca oktave nekega tona je za faktor dva višja ali nižja od tega tona. Če pogledamo sliko klavirja, nam število poltonov v oktavi da dovolj informacij, da izračunamo faktor ζ:

$$
\begin{aligned}
f_{oktava} &= f \cdot \zeta^{12} \,, \\
f_{oktava} &= f \cdot 2 \,, \\
2 &= \zeta^{12} \,, \\
\zeta &= \sqrt[12]{2} \approx 1.059463 \,.
\end{aligned}
$$

Kuža pazi vsebuje 3 tone, kjer sta sosednja med seboj ločena za dva poltona. Želel bi si zaigrati pesem nakje znotraj prvih dveh lestvic C-dura, kar pomeni približno znotraj intervala med 250 in 1000 Hz. Izberemo denimo za frekvenco prvega tona frekvenco 250 Hz. Posledično za naslednji frekvenci dobimo naslednje:

$$280.615512077343Hz \qquad 314.980262473718Hz$$

Odmaknjenost od celoštevilskih vrednosti je kar velika. Efektivna vrednost (angl. *root mean square, RMS* [12]) odmika je v tem primeru 0.385. Preko metode najmanjših kvadratov (katera išče najmanjšo rms vrednost odmika, koda B.1) lahko določimo, da z najmanjšim efektivnim odmikom 0.033 lahko za program izberemo naslednje tri frekvence:

$$531Hz \qquad 596Hz \qquad 669Hz$$

Koda B.1: Določanje frekvenc v Octave

```
exp = 2^(1/12);
poss = 250:794; # 794=1000/2^(4/12)
tones = [ poss.*(exp^0) ;
          poss.*(exp^2) ;
          poss.*(exp^4)];

err = sqrt(sum((tones-round(tones)).^2));
[min,idx] = min(err);
err(idx)
tones(:, idx)
```

B.2 Razhroščevanje ARM zbirniških programov

Za razhroščevanje lahko uporabimo GNU Debugger. Pogoj za to je, da program prevedemo z zastavico $-g$. Po zagonu gdb izvedemo inicializacijo z ukazi iz odseka kode B.2.

Koda B.2: GDB instructions

```
1 display/4i $pc
2 display $r0
3 display $r1
4 display $r2
5 display $r3
6 ...
7 break main
8 run
```

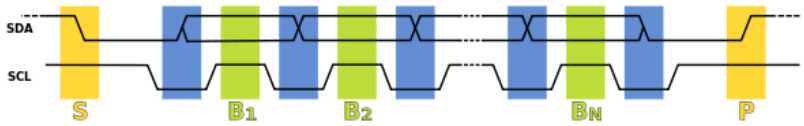
Razložimo ukaze:

- **display $rx** vrne stanja registrov in njihove vrednosti za vsak korak;
- **display/i $pc** vrne trenutni ukaz, ki se izvaja;
- **where** vrne več podatkov, med drugim tudi vrstico v izvorni kodi, kjer se trenutno nahajamo;
- **step** - korak.

Po prejšnjem zaporedju s kombinacijo ukazov *step* in *where* spremljamo naš napredek. Ob vsakem koraku namreč gdb v standardni izhod izpiše informacije, ki smo jih nastavili v inicializaciji. Vse to bi bilo z dovolj volje mogoče avtomatizirati in implementirati na spletnem strežniku iz poglavja 3.

B.3 I^2C protokol

I^2C je komunikacijski protokol, ki omogoča komunikacijo z do 127 napravami. Prenos se opravlja preko dveh signalov, SDA in SCL. SCL predstavlja urno sinhronizacijo med napravama, SDA pa predstavlja podatke. Podatki na SDA so veljavni ob pozitivni spremembi SCL (slika B.1). Podatki se prenašajo v 8-bitnih paketih, prenos pa nadzira



Slika B.1: I^{2}C časovni diagram na uporabljenih signalih (vir [13]).

glavna naprava, ki lahko zahteva prenos podatkov v eno ali drugo smer od ene izmed suženjskih naprav. V prvem prenesenem bajtu je vsebovana informacija o naslovljeni napravi (bit [7 : 1]) ter smer prenosa podatkov (bit 0). V drugem prenesenem bajtu je naslov registra znotraj naslovljene naprave. Odvisno od določene smeri prenosa lahko v ta register pišemo ali iz njega beremo. Tretji in četrti preneseni bajt predstavljata dejanski prenos podatkov. Več informacij o protokolu najdemo v njegovi specifikaciji [14].

B.4 Funkcija delay

V večini inicializacij in main funkcij uporabljamo delay funkcije. tudi v nekaterih nalogah iz poglavja 4 se pojavlja klic te funkcije. V kodi B.3 prilagam primer implementacije. Enota argumenta časovnega zamika je tisočinka sekunde.

Koda B.3: delay()

```
1 void delay (unsigned int howLong)
2  {
3    struct timespec sleeper, dummy ;
4
5    sleeper.tv_sec = (time_t)(howLong / 1000) ;
6    sleeper.tv_nsec = (long)(howLong % 1000) * 1000000 ;
7
8    nanosleep (&sleeper, &dummy) ;
9  }
```