SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1253

Implementacija senzorskog sučelja upravljačkim programom na operacijskom sustavu Linux

Benjamin Horvat

Zagreb, lipanj 2016.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 11. ožujka 2016.

DIPLOMSKI ZADATAK br. 1253

Pristupnik: **Benjamin Horvat (0036452826)**
Studij: Računarstvo
Profil: Računalno inženjerstvo

Zadatak: **Implementacija senzorskog sučelja upravljačkim programom na operacijskom sustavu Linux**

Opis zadatka:

Istražiti preporučene metode implementacije upravljačkih programa. Napraviti pregled i primjere postojećih implementacija upravljačkih programa na platformi RaspberryPi ugradbenog računalnog sustava. Implementirati očitanje vrijednosti senzora iz operacijskog sustava Linux . Razviti sklopovsko pokusno okruženje potrebno za realizaciju očitanja senzora. Koristiti temperaturni senzor s mogućnošću nadogradnje sustava na više tipova senzora. Provesti testiranje i specifikaciju performansi realiziranog sustava, uz poseban osvrt na analizu potrebnog vremena izvođenja za očitanje podatka sa senzora. Razviti pokaznu okolinu za demonstraciju sustava te dokumentaciju s uputama za instalaciju, pokretanje te testiranje razvijenog sustava.

Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 1. srpnja 2016.

Mentor:
Prof. dr. sc. Davor Petrinović

Djelovođa:
Prof. dr. sc. Danko Basch

Predsjednik odbora za diplomski rad profila:
Prof. dr. sc. Mario Žagar

Diplomski rad je izrađen u okviru suradnje Sveučilišta u Zagrebu, Fakulteta elektrotehnike i računarstva i Ericssona Nikole Tesle d.d. na istraživačko-razvojnom projektu "Poboljšanje karakteristika rada LTE radio pristupnih uređaja" (Improvements for LTE radio access equipment - ILTERA).

Zahvaljujem svom mentoru prof. dr. sc. Davoru Petrinoviću na ugodnoj suradnji i pruženoj pomoći kada je to bilo potrebno i mag. ing. Saši Tepiću koji je uvijek strpljivo pazio da rad napreduje i bude završen na vrijeme.

SADRŽAJ

1. Uvod

U ovom diplomskom radu obrađuje se tema implementacije senzorskog sučelja upravljačkim programom na operacijskom sustavu Linux.

Ako smo programeri čija je pozadina pisanje aplikacija koje se izvršavaju u korisničkom prostoru, bit će nam pomalo zbunjujuća neobična struktura i način rada upravljačkih programa, odnosno jezgrenih modula. Proučavanjem izvornog koda gotove implementacije nekog modula, prvo pitanje koje se postavlja je: "Gdje je nestala *main()* funkcija?". Upravo ove nedoumice su objašnjene u 2. poglavlju koje govori o tome što je to Linux jezgra i zašto su upravljački programi jako dobra početna točka za onog koji želi više naučiti o jezgri. Detaljno se razrađuju vrste upravljačkih programa, njihov način rada, struktura, te u kojem su odnosu s Linux jezgrom. Također, objašnjava se razlika između korisničkog i jezgrenog prostora, na koji način se moduli učitavaju u jezgru, kako se obrađuju prekidi i što je to usporedno izvođenje u jezgri.

Nadalje, u 3. poglavlju se obrađuju temperaturni senzori. Podijeljeni su na analogne i digitalne, a detaljnije su obrađeni digitalni senzori čija su senzorska sučelja implementirana u ovom diplomskom radu — obrađuje se njihova struktura, performanse i komunikacijski protokoli.

U 4. poglavlju prikazano je fizičko ostvarenje sustava, te je detaljno objašnjena implementacija izvornih kodova upravljačkih programa, nakon čega slijedi 5. poglavlje — Upute za korištenje.

Trebalo bi također napomenuti da su u dodatku A priloženi oni izvorni kodovi implementacije, koji nisu preuzeti iz drugih izvora.

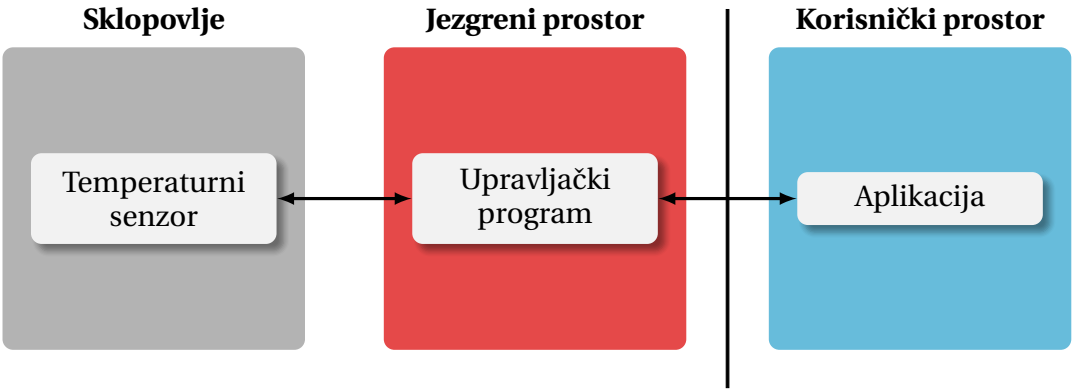
2. Upravljački programi

Linux jezgra je velik i kompleksan skup koda, no netko tko se želi baviti jezgrom treba početnu točku ulaza bez da bude preplavljen njenom kompleksnošću. Upravljački programi su često ta početna točka. Oni imaju posebnu ulogu u Linux jezgri. Zaduženi su za komunikaciju s određenim sklopovljem i poput "crne kutije" su, na čije se dobro definirano interno programsko sučelje sklopovlje odaziva, a pritom potpuno prikrivaju detalje o načinu rada sklopovlja. Korisničke se aktivnosti izvode prema skupu standardiziranih poziva koji su nezavisni od specifičnog upravljačkog programa. Mapiranje tih poziva na operacije određenog sklopa, koje se onda izvršavaju na stvarnom sklopovlju, uloga je upravljačkog programa. To programsko sučelje je takvo da se upravljački programi mogu pisati odvojeno od ostatka jezgre, te biti pokrenuti za vrijeme izvođenja operacijskog sustava, kada ih trebamo. Ta modularnost čini Linux upravljačke programe jednostavnima za pisanje [4].

2.1. Upravljački programi i Linux jezgra

Uloga upravljačkog programa je pružanje *mehanizma*, a ne *politike*. Ta razlika između mehanizma i politike je jedna od najboljih ideja Unix dizajna. Većina se programskih problema može podijeliti na dva dijela: "Koje će mogućnosti biti dostupne" (mehanizam) i "kako će se te mogućnosti moći koristiti" (politika). Ako ta dva problema rješavaju različiti dijelovi programa, ili pak dva u potpunosti različita programa, puno je lakše razvijati takav softverski paket i prilagoditi ga pojedinim potrebama.

Kada programer piše upravljački program, trebao bi posebnu pozornost pridati ovom temeljnom konceptu: pisati jezgreni kod kako bi pristupio sklopovlju, bez da nameće posebnu politiku korisniku, budući da različiti korisnici imaju različite potrebe. Upravljački program bi trebao omogućiti pristup sklopovlju, ostavljajući sva pitanja o načinu korištenja tog sklopovlja aplikaciji. Upravljački program je tada fleksibilan ako pruža pristup mogućnostima sklopovlja bez postavljanja ograniče-



Slika 2.1: Podjela na jezgreni i korisnički prostor

nja. Ponekad se, međutim, neke političke odluke moraju donijeti.

Na upravljački program možemo gledati i iz druge perspektive: on je softverski sloj koji se nalazi između aplikacije i sklopovlja (slika 2.1). Ta povlaštena uloga upravljačkog programa daje programeru slobodu da odluči na koji način će taj *device* ili sklop prikazati sustavu: različiti upravljački programi pružaju različite mogućnosti, čak i za isti device. Različita bi razmatranja o dizajnu upravljačkog programa trebala biti u ravnoteži.

U Unix sustavu, nekoliko istovremenih procesa obavlja različite zadatke. Svaki proces zahtjeva resurse sustava, bila to računalna snaga, memorija, mrežna povezanost, ili neki drugi resurs. Jezgra je veliki komad izvršnog koda koji je zadužen za rukovanje svim takvim zahtjevima. Iako podjela među različitim poslovima jezgre nije uvijek jasna, uloga jezgre se može podijeliti na sljedeće dijelove [4]:

- Upravljanje procesima
- Upravljanje memorijom
- Datotečni sustavi
- Upravljanje devicevima
- Mreža

2.2. Dinamičko učitavanje modula

Jedna od dobrih značajki Linuxa je mogućnost proširenja jezgrenih svojstava za vrijeme izvođenja sustava. Svaki dio koda koji se može učitati u jezgru za vrijeme izvođenja sustava se zove *modul*. Linux jezgra nudi podršku za nekoliko različitih tipova (ili klasa) modula, uključujući, ali ne ograničavajući se na, upravljačke programe.

Svaki se modul sastoji od objektne datoteke (npr. *"modul.ko"* – koja nije povezana u potpunu izvršnu datoteku) koja se može dinamički učitati u pokrenutu jezgru sa naredbom *insmod* a zaustaviti i izbaciti iz jezgre sa naredbom *rmmod*. Slika 2.2 prikazuje različite klase modula zaduženih za određene zadatke — modul pripada određenoj klasi prema funkcionalnostima koje nudi. Popis modula na slici 2.2 pokriva najvažnije klase, ali on nije potpun jer se sve više i više funkcionalnosti Linuxa modulariziraju [4].

2.3. Vrste upravljačkih programa

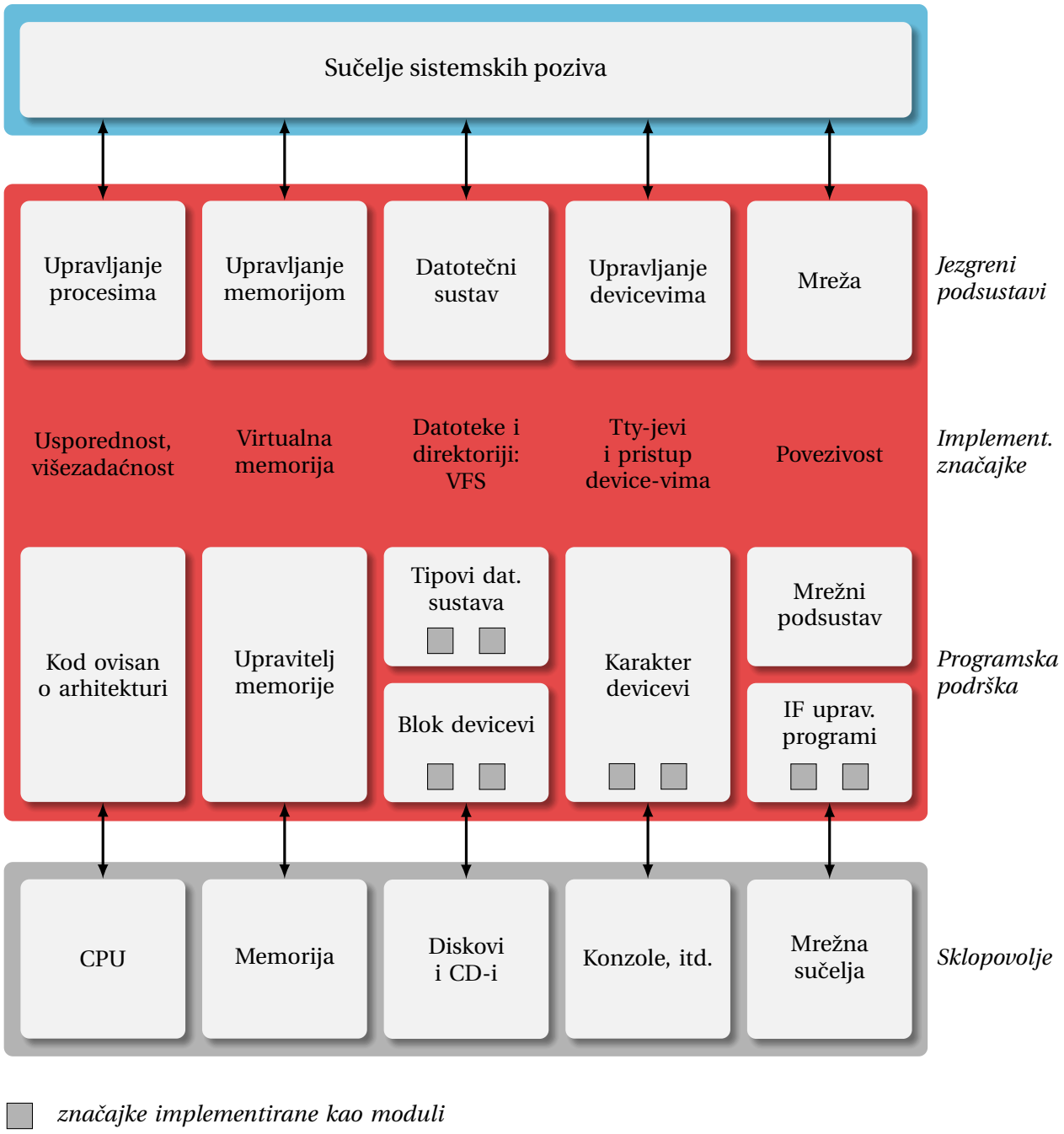
U Linuxu se devicevi dijele na tri osnovna tipa. Svaki modul najčešće implementira jedan od tih tipova, i time se klasificira kao *karakter modul*, *blok modul* ili *mrežni modul*. Ta podjela modula na različite tipove ili klase nije stroga; programer može odlučiti hoće li izgrađivati velike module implementirajući različite upravljačke programe u jednom komadu koda. Dobri programeri, ipak, najčešće za svaku novu funkcionalnost koju implementiraju izgrađuju novi modul, jer je dekompozicija ključni element skalabilnosti i proširivosti [4].

2.3.1. Karakter upravljački program

Karakter (char) deviceu se može pristupiti kao toku bajtova (kao datoteka); char upravljački program je zadužen za implementaciju takve vrste ponašanja. Takav upravljački program najčešće implementira barem *open*, *close*, *read* i *write* sustavske pozive. Tekst konzola (*/dev/console*) i serijski portovi (*/dev/ttyS0* i ostali) su primjeri char devicea, jer su dobro predstavljeni apstrakcijom toka. Char devicevima se pristupa pomoću čvorova datotečnog sustava (eng. *filesystem nodes*) kao što su */dev/tty1* i */dev/lp0*. Jedina bitna razlika između char devicea i regularne datoteke, je u tome što se u regularnoj datoteci možete pomicati naprijed i nazad, dok su većina char deviceva samo kanali podataka, kojima se može pristupiti samo slijedno. Ipak postoje char devicevi koji izgledaju kao područja podataka, po kojima se može kretati naprijed i nazad [4].

2.3.2. Blok upravljački program

Kao char devicevima, i block devicevima se pristupa preko čvorova datotečnog sustava u */dev* direktoriju. Block device je device (npr. disk) koji može biti poslužitelj datotečnog sustava. U većini Unix sustava, block device može samo rukovati



Slika 2.2: Podijeljeni pregled jezgre (prema [4])

U/I operacijama prijenosa jednog ili više cijelih blokova, koji su obično 512 bajtova (ili viša potencija broja dva) dužine. Linux, umjesto toga, omogućuje aplikaciji čitanje i pisanje block deviceva kao char devicea — dozvoljava prijenos bilo kojeg broja bajtova u datom trenutku. Kao rezultat toga, block i char devicevi se razlikuju samo po internom upravljanju jezgre podacima, a time i u programskom sučelju jezgre/upravljačkog programa. Kao char device, svakom block deviceu se pristupa preko čvora datotečnog sustava, a njihova razlika je vidljiva korisniku. Block devicevi imaju potpuno drugačije sučelje prema jezgri nego char devicevi [4].

2.3.3. Mrežni upravljački program

Svaka mrežna transakcija je ostvarena kroz sučelje — device, koji je u mogućnosti izmjenjivati podatke s ostalim poslužiteljima. Najčešće je *sučelje* elektronički sklop, ali može biti i čisti programski (softverski) device, kao što je *loopback* sučelje. Mrežno sučelje je zaduženo za slanje i primanje paketa podataka, upravljanih preko mrežnog podsustava jezgre, ne znajući kako su pojedinačne transakcije mapirane na stvarne pakete koji se prenose. Mnoge su mrežne veze (posebno one koje koriste TCP) protočno-usmjerene, a mrežni su devicevi, najčešće, projektirani oko slanja i primanja paketa. Mrežni upravljački program ne zna ništa o pojedinačnim vezama; samo rukuje paketima. Pošto mrežni device nije protočno-usmjeren, mrežno se sučelje teško preslikava (mapira) na čvor u datotečnom sustavu, kao što je */dev/tty*. Unix način pružanja pristupa sučeljima je dodjeljivanjem jedinstvenog naziva (kao što je `eth0`), ali taj naziv nema nikakve veze sa datotečnim sustavom. Komunikacija između jezgre i mrežnog upravljačkog programa je potpuno drugačija od one kod karakter i blok upravljačkih programa. Umjesto *read*-a i *write*-a, jezgra poziva funkcije koje su vezane uz prijenos paketa [4].

2.4. Razlika između jezgrenih modula i aplikacija

Dok većina malih aplikacija i aplikacija srednjih veličina obavlja neki zadatak od početka do kraja, svaki se jezgreni modul samo registrira kako bi poslužio buduće zahtjeve, a njegova se inicijalizacijska funkcija odmah prekida. Drugim riječima, zadatak je inicijalizacijske funkcije modula da se pripremi na kasnije pozivanje modulskih funkcija; to je kao da modul govori: "Tu sam, i to je ono što mogu činiti". Izlazna se funkcija modula poziva neposredno prije zaustavljanja samog modula. Trebao bi reći jezgri: "Nisam više ovdje, nemoj me pitati da činim bilo što drugo".

Ovakav je pristup programiranju sličan programiranju koje se temelji na upravljanju uvjetovanom događajima (eng. *event-driven*), ali dok sve aplikacije nisu upravljane događajima, svaki jezgreni modul jest. Druga velika razlika između aplikacija upravljanih događajima i jezgrenog koda je u izlaznoj funkciji; dok aplikacija koja završava može biti lijena u oslobađanju resursa ili izbjegava čišćenje uopće, izlazna funkcija modula mora pažljivo povratiti sve što je init funkcija pokrenula, ili će ti resursi biti zauzeti sve do ponovnog pokretanja sustava [4].

2.4.1. Jezgreni i korisnički prostor

Moduli se izvode u jezgrenom prostoru, dok se aplikacije izvode u korisničkom prostoru. Taj je koncept u srži teorije operacijskih sustava.

Uloga je operacijskog sustava, u praksi, pružati programima dosljedan pregled sklopovlja računala. Osim toga, operacijski sustav mora računati na samostalan rad programa i pružati zaštitu od neovlaštenog pristupa resursima. Taj je netrivijalan zadatak moguće ostvariti samo ako CPU nameće zaštitu sustavskog softvera iz aplikacija.

Svaki moderni procesor je u stanju provesti takvo ponašanje. Izabran je pristup provođenja različitih operacijskih razina u samom CPU-u. Svaka razina ima drugačiju ulogu, i neke su operacije zabranjene na nižim razinama; programski kod se može prebaciti s jedne razine na drugu samo kroz ograničen broj vrata. Unix sustavi su osmišljeni kako bi iskoristili ovu hardversku značajku, koristeći dvije takve razine. Svi postojeći procesori imaju barem dvije razine zaštite, a neki, poput x86 porodice, imaju više razina; kada postoji nekoliko razina, najviša i najniža razina se koriste. Pod Unixom, jezgra se izvršava u najvišoj razini (tzv. nadzorni mod rada – eng. *supervisor mode*), u kojem je sve dopušteno, dok se aplikacije izvode u najnižoj razini (tzv. korisnički mod – eng. *user mode*), gdje procesor regulira direktan pristup sklopovlju i neovlašteni pristup memoriji.

Takve načine izvršavanja nazivamo jezgreni i korisnički prostor. Ovi pojmovi ne obuhvaćaju samo različite razine povlastica koje su svojstvene za ta dva načina, već i činjenicu da svaki način može imati svoju vlastitu memorijsku presliku — svoj adresni prostor.

Unix prenosi izvršavanje iz korisničkog u jezgreni prostor kad kod aplikacija pošalje sustavski poziv ili je obustavljena sklopovskim prekidom. Jezgreni kod, kod izvršenja sustavskog poziva, radi u kontekstu procesa — djeluje u ime pozivajućeg procesa te je u mogućnosti pristupiti podacima u procesnom adresnom prostoru.

Kod koji obrađuje prekide, s druge strane, je asinkroni s obzirom na procese i ne odnosi se na bilo koji određeni proces.

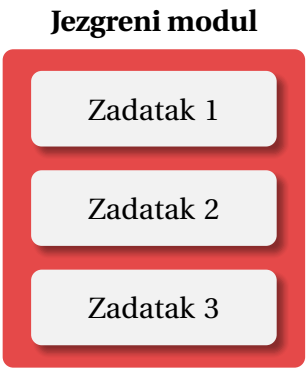
Uloga modula je proširenje funkcionalnosti jezgre; modularni kod se izvršava u jezgrenom prostoru. Obično upravljački program obavlja oba ranije opisana zadatka: neke se funkcije u modulu izvršavaju kao dio sustavskih poziva, a neke su zadužene za rukovanje prekidima [4].

2.4.2. Usporedno izvođenje u jezgri

Jedan od načina po kojem se jezgreno programiranje uvelike razlikuje od aplikacijskog programiranja je pitanje usporednosti (eng. *concurrency*). Većina aplikacija, s iznimkom višedretvenih aplikacija, se obično izvode slijedno, s početka prema kraju, bez da treba brinuti o tome bi li nešto drugo moglo promijeniti njihovu okolinu. Jezgreni kod se ne izvršava u tako jednostavnim uvjetima, pa čak i najjednostavniji jezgreni moduli moraju biti pisani s idejom da se više stvari može izvršavati odjednom (slika 2.3).

Postoji nekoliko izvora usporednosti u jezgrenom programiranju. Prirodno se u Linux sustavu izvodi više procesa, gdje postoji mogućnost da više njih želi koristiti upravljački program istovremeno. Većina deviceva može izazvati prekid procesora; obrađivači prekida se izvode asinkrono i mogu biti pozvani u vrijeme kada i upravljački program želi obaviti nešto.

Kao rezultat toga, jezgreni kod Linuxa, kao i kod upravljačkih programa, mora biti višenastupni (eng. *reentrant*) — mora biti sposoban za izvođenje u više od jednog konteksta istovremeno. Podatkovne strukture moraju biti pažljivo osmišljene kako bi držale višestruko izvođenje dretvi zasebnima, i kod se treba brinuti za takav pristup zajedničkim podacima koji sprječava uništavanje podataka [4].



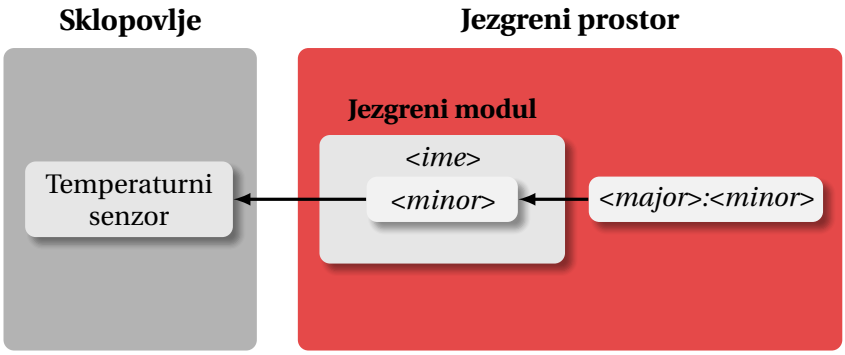
Slika 2.3: Više zadataka napreduje u isto vrijeme

2.5. Karakter upravljački programi

2.5.1. *Major* i *minor* brojevi

Linux jezgra predstavlja char i block deviceve kao parove brojeva <*major*>:<*minor*>. Neki su major brojevi rezervirani za određene upravljačke programe. Ostali se major brojevi dinamički dodjeljuju upravljačkim programima pri pokretanju operacijskog sustava. Npr. major broj 94 je uvijek major broj za DASD deviceve (eng. *Direct Access Storage Device*). Čak i više upravljačkih programa može dijeliti isti major broj. U */proc/devices* možemo vidjeti kako se major brojevi dodjeljuju u pokrenutoj instanci Linuxa (slika 2.4).

Upravljački program koristi minor broj <*minor*> za razlikovanje pojedinih fizičkih i logičkih deviceva. Npr. DASD upravljački program dodjeljuje četiri minor brojeva svakom DASD-u: jedan broj DASD-u kao cijelini i ostala tri za tri particije. Upravljački programi dodjeljuju device imena svojim devicevima, prema određenoj shemi imenovanja. Svako device ime je povezano s minor brojem (slika 2.5).



Slika 2.5: Minor brojevi i imena deviceva

Aplikacije iz korisničkog prostora pristupaju char i block devicevima kroz device čvorove (eng. *device nodes*) koji se također spominju kao posebne datoteke ili device datoteke (eng. *special files* ili *device files*). One se dogovorno nalaze u */dev* direktoriju. Kada se stvori device datoteka, povezuje se s major i minor brojem (slika 2.6) [3].

2.5.2. Alociranje i oslobađanje *device* brojeva

Jedna od prvih stvari koju upravljački program treba učiniti pri postavljanju char devicea je zatražiti jedan ili više device brojeva s kojima će raditi. Potrebna funkcija za to je *register_chrdev_region*, koja je deklarirana u <*linux/fs.h*>:

```
pi@raspberrypi ~ $ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
100 char_dev
116 alsa
128 ptm
136 pts
153 spi
162 raw
180 usb
189 usb_device
204 ttyAMA
249 hidraw
250 vchiq
251 vc-cma
252 bsg
253 vc-mem
254 rtc

Block devices:
  1 ramdisk
259 blkext
  7 loop
  8 sd
 65 sd
...
```

Slika 2.4: Ispis */proc/devices* datoteke na Raspbianu 3.10.25+

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

Kao i kod većine jezgrenih funkcija, povratna vrijednost će biti 0 ako je alokacija bila uspješna, a ako je došlo do pogreške biti će vraćen negativni error kod.

Neki su major device brojevi već statički dodijeljeni najčešćim devicevima. Listu tih deviceva možemo pronaći u dokumentaciji (*Documentation/devices.txt*) izvornog koda Linux jezgre.

Dobro je koristiti *register_chrdev_region* samo ako unaprijed znamo device broj kojeg želimo. Često, međutim, nećemo znati koje će major brojeve naš device zapravo koristiti. Ako je to tako, jezgra će nam rado dodjeliti neki nasumični major broj koji je slobodan, koristeći funkciju:

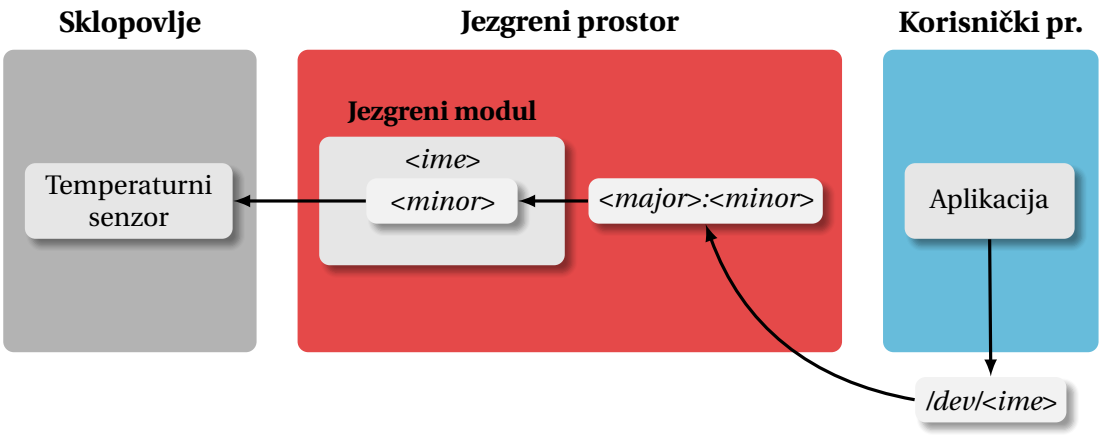
```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
                        unsigned int count, char *name);
```

Neovisno o tome koju funkciju koristimo za alokaciju device brojeva, oslobađamo ih funkcijom:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Funkcija *unregister_chrdev_region* je uobičajeno smještena i poziva se iz "cleanup" funkcije modula.

Gore navedene funkcije alociraju device brojeve u svrhu korištenja istih od strane upravljačkog programa, ali one ne daju jezgri do znanja kako će se ti brojevi koristiti. Prije nego li aplikacija iz korisničkog prostora može pristupiti jednome od tih device brojeva, upravljački ih program treba povezati sa svojim internim funkcijama koje implementiraju operacije tog devicea [4].



Slika 2.6: Minor brojevi i imena deviceva

2.6. Obrada prekida

Linux obrađuje prekide na isti način na koji obrađuje signale u korisničkom prostoru. U najvećem dijelu, upravljački program treba samo registrirati prekidnu funkciju za svoj device, a prekide ispravno obraditi kada stignu.

U jezgri se nalazi registar prekidnih linija, sličan registru U/I portova. Od modula se očekuje da zatraži prekidni kanal (ili IRQ od eng. *interrupt request*) prije uporabe, i da ga oslobodi kada završi. U mnogim situacijama, od modula se također očekuje da bude u mogućnosti dijeliti prekidne linije s drugim upravljačkim programima. Sljedeće funkcije, deklarirane u <*linux/interrupt.h*>, implementiraju registracijsko sučelje prekida:

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);
```

```
void free_irq(unsigned int irq, void *dev_id);
```

Nije neuobičajeno da funkcija vrati -EBUSY signal, time signalizirajući da već drugi upravljački program koristi traženu prekidnu liniju [4].

3. Temperaturni senzori

Postoji više vrsta senzora za razne električne i elektroničke primjene. Neki od njih su senzori tlaka, temperature, položaja, sile, blizine, prisutnosti, protoka, zvuka, brzine, termički senzori, kemijski, optički, magnetski itd. Možemo ih podijeliti i na analogne i digitalne.

Senzor se može definirati kao uređaj koji detektira fizičke, električne ili neke druge promjene, te proizvodi električni ili optički izlazni signal kao potvrdu te promjene [2].

Razmotrit ćemo razliku između analognih i digitalnih senzora, te analognih i digitalnih temperaturnih senzora. Također ćemo proučiti specifičnosti digitalnih temperaturnih senzora DS18B20, DHT11 i DHT22 čije smo upravljačke programe implementirali u ovom diplomskom radu.

3.1. Analogni temperaturni senzori

Analognim senzorima se smatraju oni senzori koji proizvode kontinuirani analogni izlazni signal. Taj je kontinuirani izlazni signal proporcionalan mjerenoj veličini. Postoji više vrsta analognih senzora: akcelerometri, senzori tlaka, senzori svjetla, zvučni senzori, temperaturni senzori, itd. [2].

Široko su rasprostranjeni i digitalni i analogni temperaturni senzori. Tipični analogni temperaturni senzor je termorezistor (eng. *thermistor* — slika 3.1). Postoje različite vrste termorezistora koji se koriste za različite primjene. Termorezistor je toplinski osjetljivi otpornik koji služi za mjerenje promjena u temperaturi. Ako temperatura raste, tada raste i električni otpor termorezistora. Isto, ako temperatura pada, i otpor pada.

Termorezistor jer najčešće korišteni analogni temperaturni senzor, zbog svoje niske cijene, ali nije toliko precizan kao digitalni temperaturni senzori.

Slika 3.1: Termorezistor SG307 Ametherm

3.2. Digitalni temperaturni senzori

Električni senzori ili elektrokemijski senzori u kojima se pretvorba i prijenos podataka dešavaju digitalno, se zovu digitalni senzori. Digitalni senzori zamjenjuju analogne senzore jer su sposobni prevladati nedostatke analognih senzora.

Digitalni se senzori sastoje od tri komponente: senzora, elektičnog voda i odašiljača. U digitalnim se senzorima mjereni signal izravno pretvara u digitalni izlazni signal unutar samog senzora, i taj se digitalni signal odašilje preko žice digitalno.

Postoje različite vrste digitalnih senzora koji prevladavaju nedostatke analognih [2].

3.2.1. DS18B20

Digitalni temperaturni senzor DS18B20 (slika 3.2) omogućuje 9-bitna do 12-bitna mjerenja temperature Celzijusa, te posjeduje alarmnu funkciju kojoj se može korisnički programirati gornja i donja granica okidanja.

DS18B20 komunicira preko *1-Wire* sabirnice koja prema definiciji zahtjeva samo jednu podatkovnu liniju (i masu) za komunikaciju sa centralnim mikroprocesorom. Osim toga, DS18B20 se može napajati direktno preko podatkovne linije ("parazitno napajanje" — eng. *parasite power*), time eliminirajući potrebu za vanjskim izvorom napajanja.

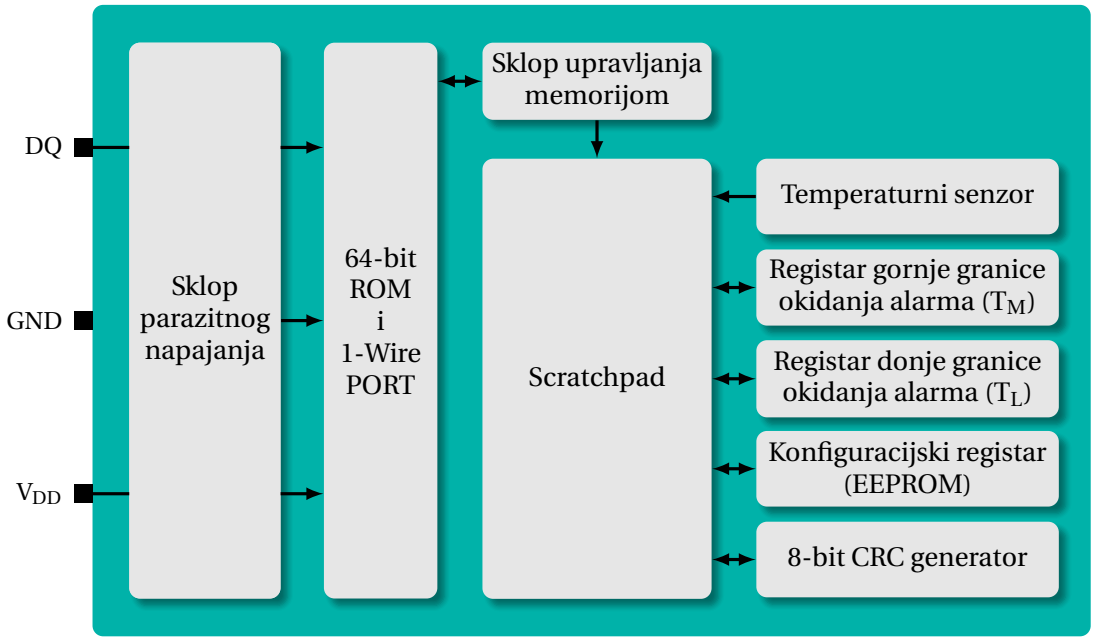
Svaki DS18B20 ima jedinstveni 64-bitni serijski broj, što omogućuje funkcioniranje više DS18B20 senzora na istoj 1-Wire sabirnici. Stoga je jednostavno koristiti jedan mikroprocesor za upravljanje mnogih DS18B20 senzora raspoređenih na jako velikom području [6].

Glavne značajke senzora:

- 1-Wire sučelju je potreban samo jedan pin za komunikaciju
- Smanjeno brojanje komponenata pomoću integriranog temperaturnog senzora i EEPROM-a
  - Temperaturno mjerno područje od -55–125 °C (-67–257 °F)
  - ±0.5 °C točnost od -10–85 °C
  - Programabilna rezolucija od 9-bitova do 12-bitova
  - Nema potrebe za vanjskim komponentama
- Parazitski način napajanja zahtjeva samo 2 pina za rad (DQ i GND)
- Pojednostavljuje primjene distribuiranog mjerenja temperature s *Multidrop* mogućnošću
  - Svaki uređaj ima jedinstveni 64-bitni serijski broj pohranjen u ROM-u
- Fleksibilne postavke alarma koje korisnik može definirati, s alarmnom naredbom traženja senzora (eng. *Alarm Search Command*) čije se temperature nalaze izvan programiranih granica
- Dostupni u 8-pinskim SO, µSOP i 3-pinskim TO-92 kućištima

Slika 3.2: Temperaturni senzor Dallas DS18B20

Blok dijagram za DS18B20 se nalazi na slici 3.3 — sastoji se od sklopa za parazitno napajanje, 64-ROM-a i 1-Wire PORT-a, sklopa za upravljanje memorijom, scratchpad memorije, temperaturnog senzora, registara za postavljanje gornje i donje granice okidanja alarma, konfiguracijskog registra i 8-bitnog CRC generatora. Na dijagramu možemo vidjeti da senzor ima 3 pina: DQ – podatkovni pin, GND – masa i V_{DD} – napajanje.



Slika 3.3: DS18B20 blok dijagram

Transakcijski slijed za pristup senzoru DS18B20 je sljedeći:

- Inicijalizacija
- ROM naredba (nakon čega slijede potrebne podatkovne razmjene)
- DS18B20 funkcijska naredba (nakon čega slijede potrebne podatkovne razmjene)

Vrlo je važno slijediti ovaj redoslijed svaki puta kada pristupamo senzoru, jer DS18B20 neće reagirati ako će bilo koji od koraka u slijedu nedostajati ili biti neispravan. Iznimke ovom pravilu su naredbe *Search ROM* [F0h] i *Alarm Search* [ECh]. Nakon izdavanja bilo koje od ovih ROM naredbi, *master* se mora vratiti na korak 1. u slijedu [7].

Inicijalizacija

Inicijalizacijski slijed se sastoji od *Reset* signala kojeg odašilje *master* sabirnice, nakon čega slijedi *Presence* signal kojeg odašilje *slave*. *Presence* signal daje *masteru* do znanja da su *slave* uređaji prisutni na sabirnici i da su spremni za rad.

ROM naredbe

Nakon što je *master* sabirnice detektirao *Presence* signal, može izdati ROM naredbu. Te naredbe rade s jedinstvenim 64-bitnim ROM kodovima svakog *slave* uređaja, te

omogućuju *masteru* da izdvoji određeni uređaj ako su prisutni mnogi na sabirnici. Te naredbe također omogućuju *masteru* da odredi koliko je uređaja i kojeg tipa prisutno na sabirnici, te je li neki uređaj u alarmnom stanju. Postoji pet ROM naredbi, i svaka je duga 8-bita. *Master* uređaj mora izdati prikladnu ROM naredbu prije nego izdaje DS18B20 funkcijsku naredbu. Te su naredbe: *Search Rom* [F0h], *Read Rom* [33h], *Match Rom* [55h], *Skip Rom* [CCh] i *Alarm Search* [ECh].

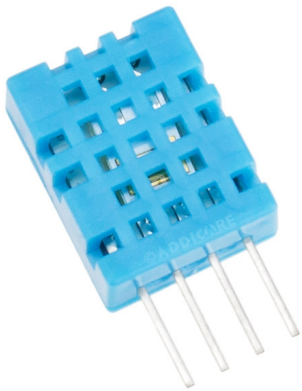
Funkcijske naredbe

Nakon što je *master* izdao ROM naredbu kako bi adresirao DS18B20 s kojim želi komunicirati, *master* može izdati jednu od DS18B20 funkcijskih naredbi. Te mu naredbe omogućuju pisanje u, i čitanje iz *scratchpad* memorije, pokretanje temperaturne pretvorbe i određivanje načina napajanja [7]. Te su naredbe: *Convert T* [44h], *Write Scratchpad* [4Eh], *Read Scratchpad* [BEh], *Copy Scratchpad* [48h], *Recall* E^2 [4Eh], *Read Power Supply* [B4h].

Neki su od signala, implementiranog upravljačkog programa senzora DS18B20, za vrijeme njegovog izvođenja, prikazani na slikama 4.5 i 4.6 u odjeljku 4.1.

3.2.2. DHT11

DHT11 (slika 3.4) je jednostavan i vrlo jeftin digitalni senzor za mjerenje temperature i vlage. Koristi kapacitivni senzor za mjerenje vlage i termorezistor za mjerenje temperature, a na podatkovnom pinu izbacuje izmjerenu informaciju kao digitalni signal (nije potreban ulazni analogni signal). Poprilično je jednostavan za korištenje, ali zahtjeva pažljivo računanje vremena za dohvaćanje podataka. Jedina prava loša strana ovog senzora je da se mjerenje može dohvatiti svake sekunde, pa su i informacije stare do jednu sekundu.



Slika 3.4: Temperaturni senzor DHT11

Uspoređujući ga s DHT22, ovaj je senzor manje precizan i radi u manjem mjernom području temperature i vlage, ali je manji i jeftiniji [1].

Iako koristi samo jedan podatkovni pin za slanje podataka, ne koristi Dallasov 1-Wire protokol već svoju vlastitu izvedbu protokola. Za razliku od DS18B20, nema mogućnost funkcioniranja više senzora na istoj podatkovnoj liniji, već svaki senzor mora imati svoj zasebni podatkovni pin.

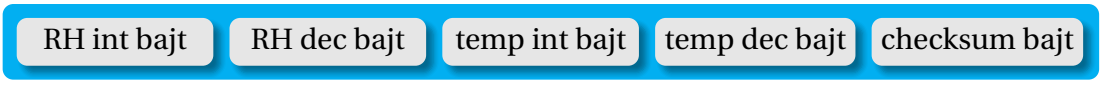
Glavne značajke senzora:

- Jeftin
- 3–5 V napajanje i U/I
- Maksimalna struja od 2.5 mA pretvorbe (dohvaćanja podataka)
- Dobar za čitanje 20–80% vlažnosti zraka s preciznošću od 5%
- Dobar za čitanje temperature od 0–50 °C s preciznošću od ±2 °C
- Uzorkovanje do 1 Hz (jednom svake sekunde)
- Veličina kućišta 15.5 mm x 12 mm x 5.5 mm
- Sadrži 4 pina

DHT11 komunikacijski vremenski dijagrami

Sljedeći vremenski dijagrami opisuju protokol za prijenos podataka između CPU-a i senzora DHT11. CPU inicira prijenos podataka slanjem *Start* signala. CPU prvo postavlja podatkovnu liniju na nisko na najmanje 18 ms, a zatim ju postavlja visoko za sljedećih 20–40 µs prije nego ju oslobađa. Zatim senzor reagira na CPU *Start* signal povlačenjem linije nisko za 80 µs nakon čega slijedi logička jedinica koja isto traje 80 µs. Nakon što je CPU primio *Response* signal sa senzora, trebao bi biti spreman za primanje podataka sa senzora (slika 3.5).

Senzor zatim šalje 40 bitova (5 bajta) podataka kontinuirano na podatkovnoj liniji. Treba imati na umu da za vrijeme prijenosa bajtova, senzor šalje najznačajniji bit prvi. 40-bitovni podatak sa senzora ima sljedeću strukturu:



Slika 3.6: Struktura 40-bitovnog podatka sa senzora DHT11

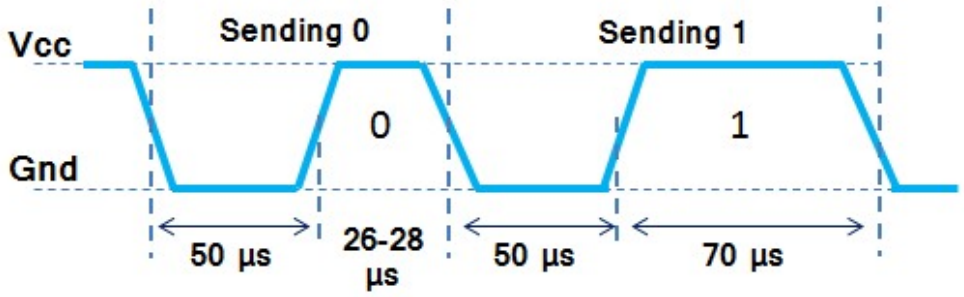
Za senzor DHT11 su decimalni bajtovi mjerenja temperature i vlage uvijek nula. Prema tome, prvi i treći bajt primljenog podatka zapravo daju brojčane vrijednosti

izmjerene relativne vlage (RH%) i temperature (°C). Zadnji bajt je *checksum* bajt koji se koristi kako bi bili sigurni da se prijenos podataka dogodio bez greške. Ako su svih pet bajtova uspješno preneseni, onda *checksum* bajt mora biti jadnak zadnjim osam bitova zbroja prva četiri bajta. Npr.:

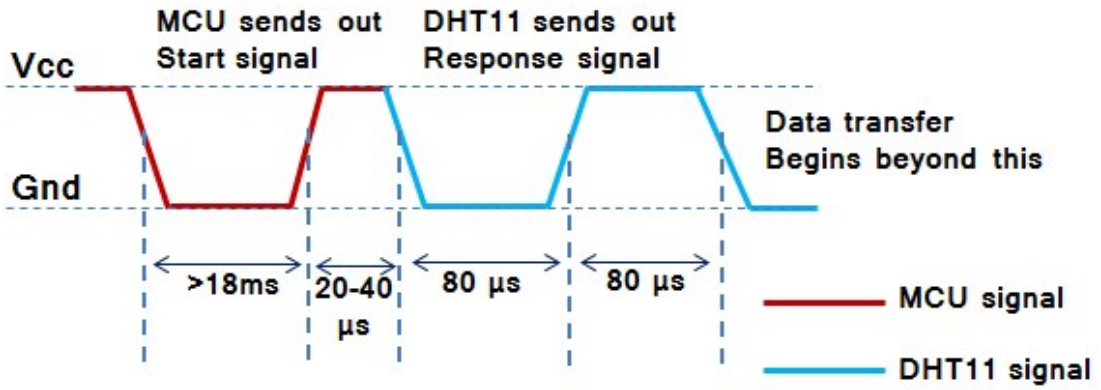
checksum bajt = zadnjih osam bitova zbroja prva četiri bajta

Slika 3.7: *Checksum* senzora DHT11

Bitno je znati na koji način se signalizira prijenos "0" ili "1". Kako bi poslali jedan bit informacije, senzor prvo postavlja liniju nisko na 50 µs. Zatim podiže liniju visoko na 26–28 µs ako treba poslati "0", ili na 70 µs ako treba poslati "1". Tako je širina pozitivnog impulsa ona koja nosi informaciju o "0" ili "1" (slika 3.8). Na kraju zadnjeg poslanog bita, senzor povlači podatkovnu liniju nisko na 50 µs i oslobađa ju.

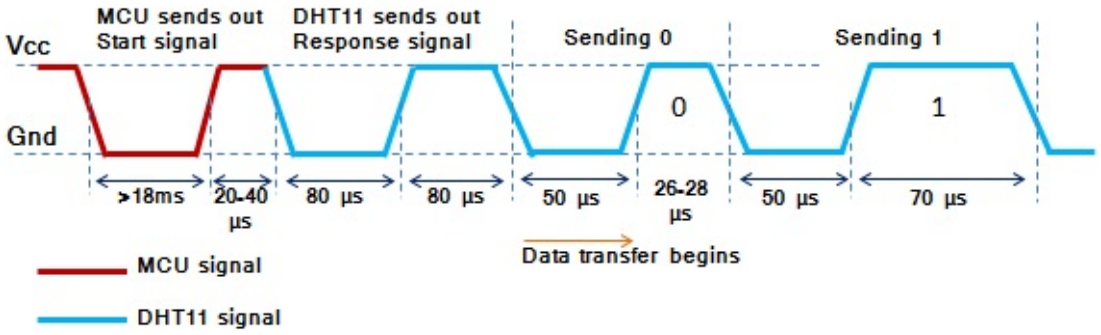


Slika 3.8: Vremenska razlika za slanje nula i jedinica [12]



Slika 3.5: *Start* i *Response* signali [12]

Na slici 3.9 je prikazan cijeli proces komunikacije između CPU-a i senzora, od traženja senzora do primanja podataka sa senzora.

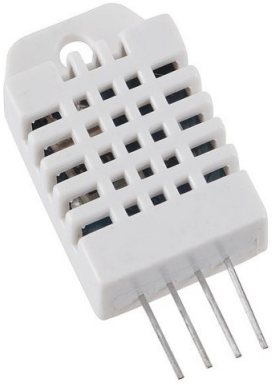


Slika 3.9: Start, Response and Data signals in sequence [12]

Senzoru DHT11 potreban je eksterni *pull-up* otpornik koji treba biti spojen između napajanja i podatkovne linije, tako da je u stanju mirovanja podatkovna linija uvijek povučena visoko. Nakon završetka prijenosa podataka i oslobađanja podatkovne linije, DHT11 odlazi u mod niske potrošnje energije sve dok novi *Start* signal ne dođe sa CPU-a.

3.2.3. DHT22

Ovaj senzor (slika 3.10) je isti kao DHT11, tj. koristi isti protokol za komunikaciju i ima jednak broj pinova, ali ima drugačije performanse. Precizniji je i ima šire mjerno područje temperature/vlage, te je veći i skuplji, ali se informacije mjerenja mogu dohvaćati svake dvije sekunde. Još je bitno za napomenuti da DHT22 koristi sva četiri bajta informacije (i decimalne vrijednosti), za razliku od DHT11 koji koristi samo dva bajta, a decimalne vrijednosti su mu uvijek nula.



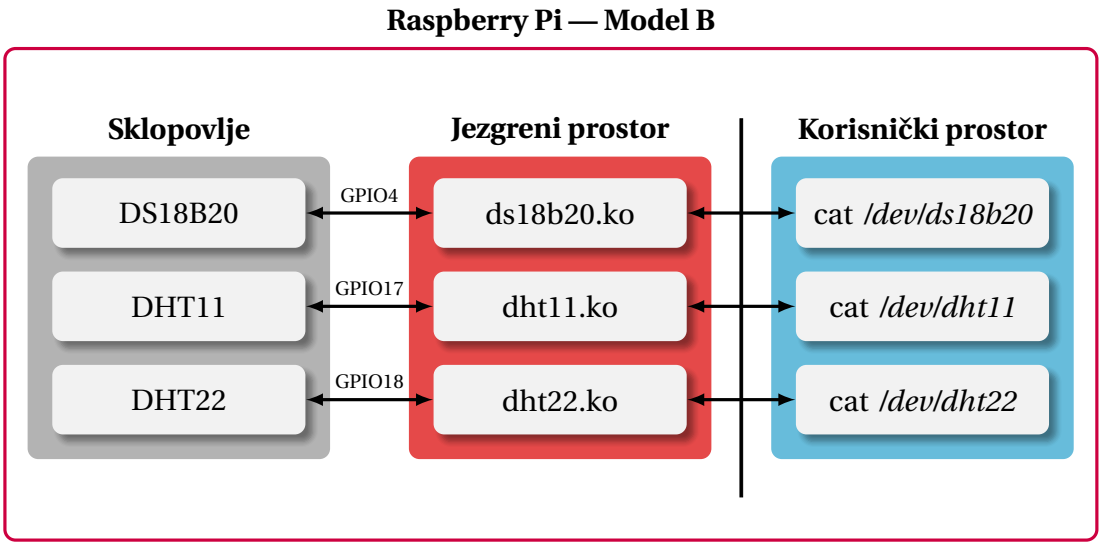
Slika 3.10: Temperaturni senzor DHT22

Glavne značajke senzora:

- Jeftin
- 3–5 V napajanje i U/I
- Maksimalna struja od 2.5 mA pretvorbe (dohvaćanja podataka)
- Dobar za čitanje 0–100% vlažnosti zraka s preciznošću od 5%
- Dobar za čitanje temperature od -40–80 °C s preciznošću od ±0.5 °C
- Uzorkovanje do 0.5 Hz (jednom svake dvije sekunde)
- Veličina kučišta 27 mm x 59 mm x 13.5 mm
- Sadrži 4 pina
- Težina 2.4 g

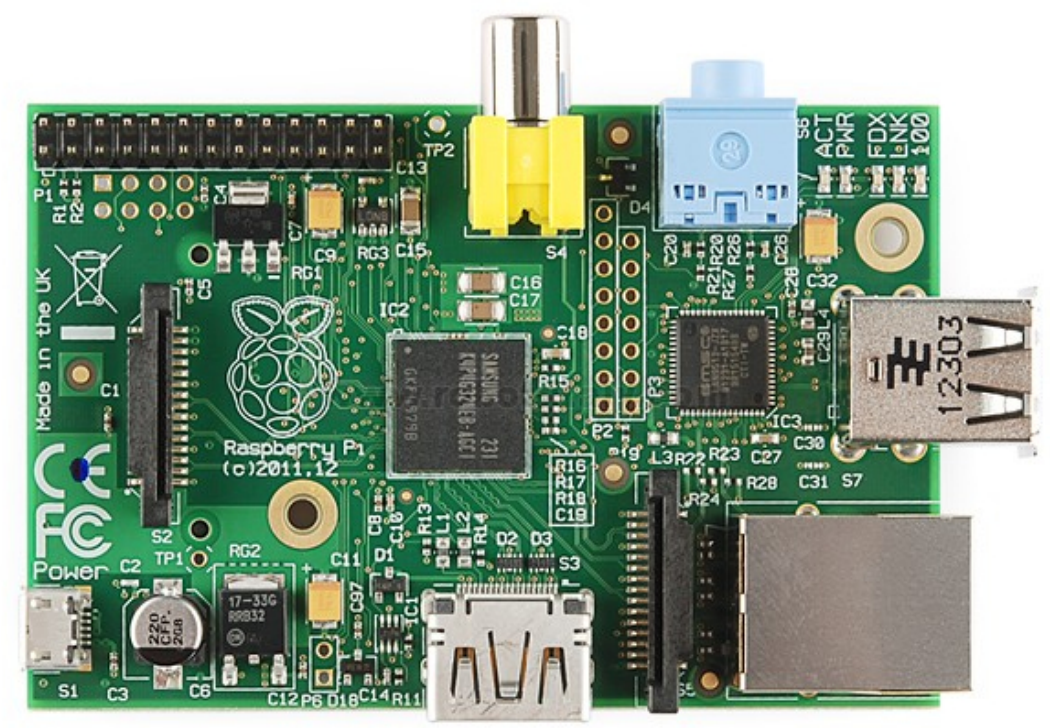
4. Implementacija

Na slici 4.1 je prikazan sustav koji se sastoji od Raspberry Pi-a Modela B (slika 4.2), na kojemu je pokrenut operacijski sustav Raspbian s Linux jezgrom verzije 3.10.25+. Senzori su spojeni na Raspberry Pi preko GPIO headera (slika 4.3). Senzor DS18B20 je spojen preko pina GPIO4 (koristi vanjsko napajanje i 4.7 kΩ *pullup* otpornik), senzor DHT11 preko pina GPIO17, a senzor DHT22 preko pina GPIO18. Svi senzori su spojeni na zajedničko napajanje od 5 V preko fizičkog pina 2.

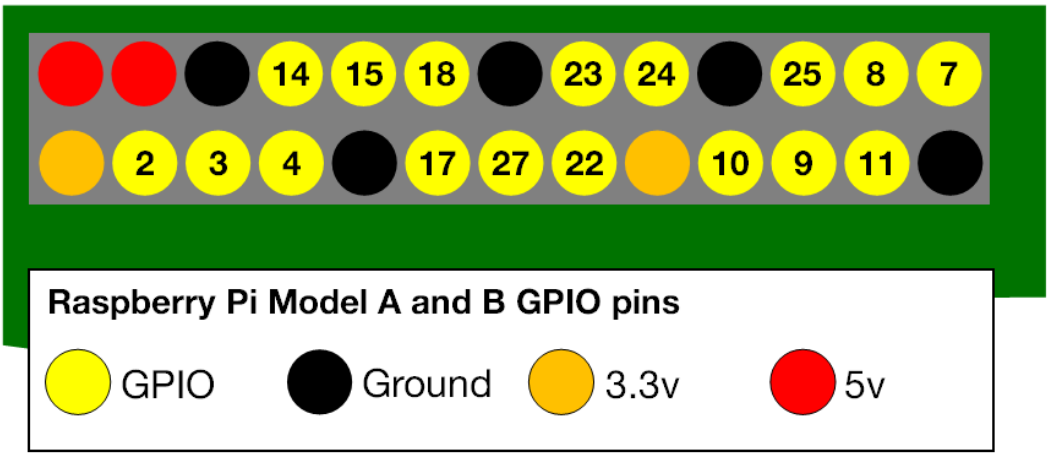


Slika 4.1: Prikaz implementacije cijelog sustava

Za svaki je senzor implementiran zaseban jezgreni modul koji je zadužen za komunikaciju sa svojim senzorom. Pokretanjem modula i stvaranjem potrebnih device datoteka (objašnjeno u poglavlju — 5. Upute za korištenje) možemo pristupiti podacima koje mjere senzori. Npr. pokretanjem naredbe "cat */dev/dht11*" iz korisničkog prostora šalje se zahtjev jezgrenom modulu *dht11.ko* za dohvaćanjem informacije o izmjerenoj temperaturi/vlazi sa senzora. Jezgreni modul koji je u jezgrenom prostoru prihvaća taj zahtjev i komunicira izravno sa senzorom. Nakon što je jezgreni modul pročitao podatke sa senzora, šalje ih nazad u korisnički prostor, gdje se informacije ispisuju na zaslon u naredbenom retku (*standard I/O*).



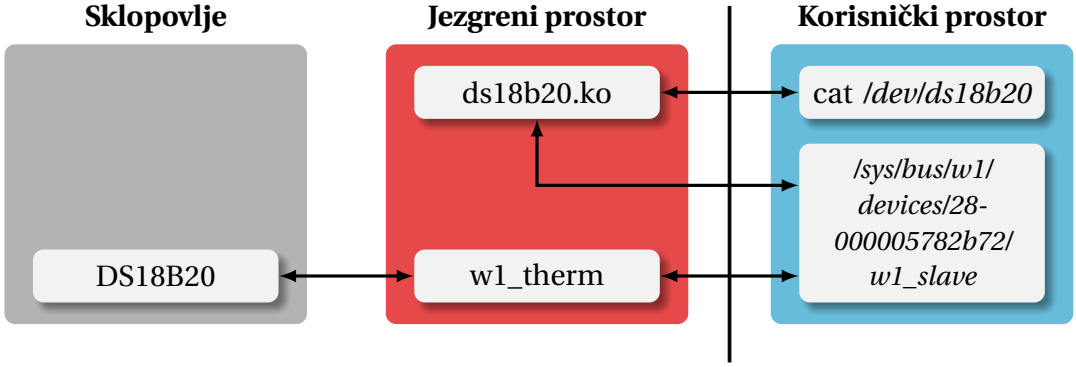
Slika 4.2: Raspberry Pi Model B



Slika 4.3: Raspberry Pi Model B raspored *header* pinova [5]

4.1. DS18B20

Detaljan prikaz implementacije jezgrenog modula *ds18b20.ko*, koji je implementiran kao među modul, je prikazan na slici 4.4. Izvršavanjem "cat */dev/ds18b20*" naredbe jezgreni modul čita podatke iz datoteke */sys/bus/w1/devices/28-000005782b72/w1_slave*, koja se nalazi u korisničkom prostoru. Zapravo je *w1_therm* onaj jezgreni modul koji komunicira sa senzorom, a već je implementiran u Linux jezgri koju koristimo. Neuobičajeno je da jezgreni modul radi na takav način, a u nastavku ovog poglavlja je objašnjeno zašto je to tako, i kako se ostvaruje.



Slika 4.4: Prikaz implementacije sustava senzora DS18B20

Kao i svaki modul, i ovaj modul sadrži *init* i *exit* funkcije. Funkcija *init* se poziva prilikom pokretanja modula naredbom *insmod*, a funkcija *exit* prilikom zaustavljanja modula naredbom *rmmod*:

```
static int __init ds18b20_in(void);
static int __exit ds18b20_out(void);

module_init(ds18b20_in);
module_exit(ds18b20_out);
```

Unutar funkcije *init* se poziva funkcija *alloc_chrdev_region* kojom modul zahtijeva dodjelu nasumičnog major broja s kojim će raditi. Također se poziva funkcija *cdev_init* koja inicijalizira cdev strukturu što ju čini spremnom za dodavanje u sustav s *cdev_add*:

```
alloc_chrdev_region(&dev_num, 0, 1, DEVICE_NAME);
cdev_init(&my_cdev, &my_fops);
cdev_add(&my_cdev, dev_num, 1);
```

Struktura *cdev* je jezgrena interna struktura koja predstavlja char deviceve, i ona je jedan od elemenata *inode* strukture koju jezgra interno koristi za predstavljanje datoteka:

```
struct cdev my_cdev;
```

Unutar *exit* funkcije se pozivaju funkcije *cdev_del* i *unregister_chrdev_region* kojima se oslobađa *cdev* struktura i device brojevi u sustavu:

```
cdev_del(&my_cdev);
unregister_chrdev_region(dev_num, 1);
```

Naredbom "cat */dev/ds18b20*" se pokreće funkcija *my_read* iz jezgrenog modula *ds18b20.ko*:

```
ssize_t my_read(struct file *filp, char __user *buff, size_t count,
                loff_t *f_pos);
```

Ona je registrirana u *file_operations* strukturi kao funkcija koja je zadužena za rukovanje operacijom čitanja:

```
struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .read = my_read
};
```

Iz funkcije *my_read* se poziva funkcija *read_file* koja dohvaća podatke iz */sys/bus/w1/devices/28-000005782b72/w1_slave* i dohvaćene podatke parsira:

```
static int read_file(char *filename);
```

Pomoću te funkcije jezgreni modul, koji se nalazi u jezgrenom prostoru, čita datoteku iz korisničkog prostora. Takav način rada jezgrenih modula nije uobičajen, i zapravo bi ga trebalo izbjegavati, jer time modul nameće svoju politiku (objašnjeno u odjeljku 2.1).

Jezgra očekuje da pokazivač, koji se prosljeđuje *filp_open()* funkcijskom pozivu, dolazi iz korisničkog prostora. Zato vrši provjeru nad pokazivačem, kako bi potvrdio da se nalazi unutar ispravnog adresnog prostora u nastojanju da ga pretvori u jezgreni pokazivač, koji ostatak jezgre može koristiti. Dakle, kada pokušavamo funkciji proslijediti jezgreni pokazivač, javlja nam grešku -EFAULT.

Kako bi riješili ovu neusklađenost adresnih prostora, koristimo funkcije *get_fs()* i *set_fs()*. Ove funkcije mijenjaju adresne granice procesa na one koje pozivatelj želi. U slučaju funkcije *filp_open()*, želimo reći jezgri da su pokazivači unutar jezgrenog adresnog prostora sigurni, pa onda taj odsječak koda izgleda ovako:

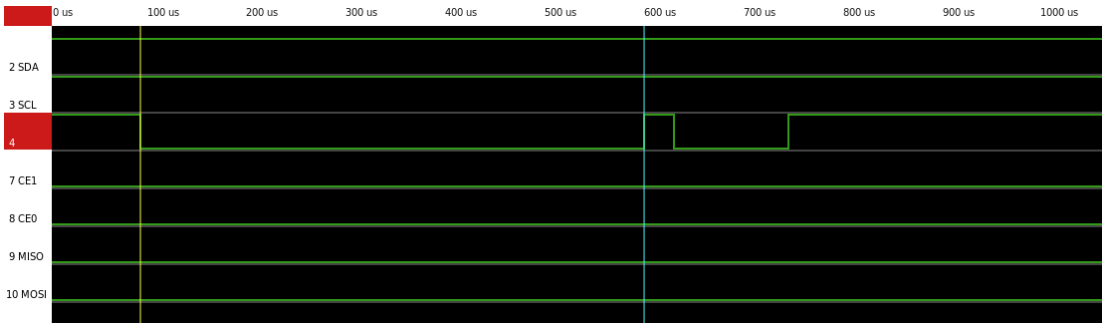
```
fp = filp_open(filename, O_RDONLY, 0);

old_fs = get_fs();
set_fs(KERNEL_DS);
...
vfs_read(fp, buffer, sizeof(buffer), &pos);
...
set_fs(old_fs);
```

Jedina dva važeća argumenta za *set_fs()* su KERNEL_DS (eng. *kernel data segment*) i USER_DS (eng. *user data segment*).

Kako bi utvrdili koje su trenutne adresne granice, prije nego ih mijenjamo, trebamo zvati funkciju *get_fs()*. Zatim, kada jezgreni modul završi sa zloupotrebljavanjem jezgrenog API-a, može obnoviti ispravne adresne granice [9].

Inicijalizacija

Na slici 4.5 vidimo kako izgledaju inicijalizacijski *Reset* i *Presence* signali našeg implementiranog sustava koji radi. *Reset* signal traje minimalno 480 µs nakon čega treba čekati 15–60 µs, a zatim slijedi *Presence* signal koji traje 60–240 µs, što je vidljivo i na vremenskom dijagramu [1].



Slika 4.5: DS18B20 inicijalizacija — *Reset* (master) i *Presence* (slave) signali

ROM naredba — *Skip Rom* [CCh]

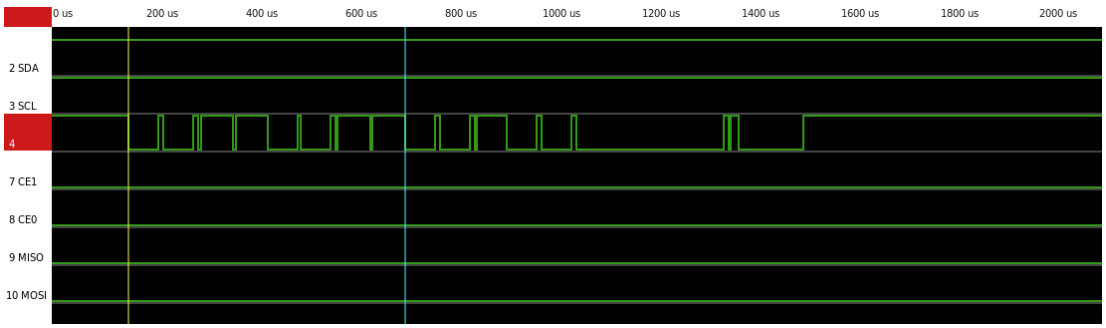
Nakon inicijalizacije *master* izdaje *Skip Rom* ROM naredbu, koju vidimo na prvoj polovici vremenskog dijagrama slike 4.6. *Master* koristi ovu naredbu za adresira-

[1] Signali su izmjereni, na realnom sustavu koji radi, pomoću aplikacije *Piscope* [8]. Ona mjeri i prikazuje stanja GPIO pinova Raspberry Pi-a

nje svih uređaja na sabirnici, bez slanja bilo kakvih podataka o ROM kodu. Npr. *master* može izdati zahtjev da svi DS18B20 senzori na sabirnici istovremeno izvode temperaturnu pretvorbu izdavanjem naredbe *Skip Rom* nakon koje slijedi *Convert T* naredba.

Funkcijska naredba — *Convert T* [44h]

Nakon ROM naredbe slijedi funkcijska naredba *Convert T*, koju vidimo na drugoj polovici slike 4.6. Ta naredba započinje jednu temperaturnu pretvorbu. Nakon pretvorbe, izmjereni podatak o temperaturi je pohranjen u 2-bajtnom temperaturnom registru u *scratchpad* memoriji, a DS18B20 se vraća u način rada niske potrošnje energije (eng. *idle state*). Ako uređaj koristi parazitno napajanje, unutar maksimalno 10 µs nakon što je ta naredba izdana *master* mora upaliti jaki *pullup* na *1-Wire* sabirnici, za vrijeme trajanja pretvorbe. Ako se DS18B20 napaja s vanjskim napajanjem, *master* može izdati vremenske slotove čitanja nakon naredbe *Convert T*, a DS18B20 će odgovoriti slanjem 0 ako traje temperaturna pretvorba, a 1 ako je pretvorba završila. U parazitnom se načinu napajanja ova tehnika obavještavanja ne može koristiti, jer je sabirnica postavljena na visoko pomoću *pullupa* za vrijeme pretvorbe [7].



Slika 4.6: DS18B20 *Skip ROM* [CCh] + *Convert T* [44h] naredbe

4.2. DHT11

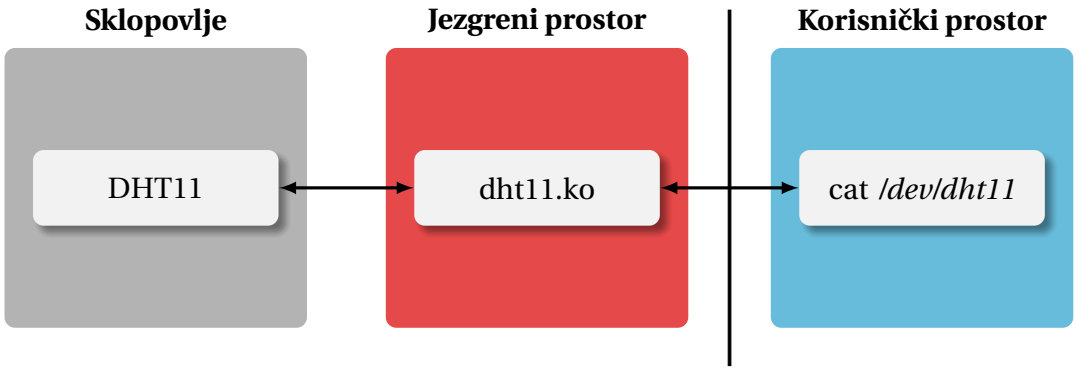
Prije nego prijeđemo na implementaciju samog jezgrenog modula, bilo bi dobro napomenuti da je ovaj sustav DHT11 senzora ostvaren bez vanjskog otpornika na podatkovnoj liniji prema napajanju, jer GPIO pinovi na Raspberry Pi-u već imaju interni *pullup*.

Čitanje senzora DHT11 je ostvareno pomoću *bit banging* metode. To je metoda za serijsku komunikaciju u kojoj softver izravno postavlja i uzorkuje stanja pinova, a

ne sklopovlje, pa se može implementirati po vrlo niskoj cijeni. Softver je odgovoran za sve parametre signala: vrijeme, sinkronizaciju, itd.

Bit banging metoda unutar jezgrenog modula *dht11.ko* je ostvarena pomoću prekida (eng. *interrupts*). Nakon što jezgreni modul pošalje *Start* signal i postavi svoj GPIO pin kao ulazni, započinje prijenos podataka sa senzora (prvo *Response* signal, a zatim podaci). Svaka sa promjena na pinu, bila ona padajućeg ili rastućeg brida, prepoznaje kao prekid. Pojavom prekida pokreće se prekidna rutina (eng. *interrupt handler*) unutar koje se provjerava je li kombinacija niske i visoke razine signala "1" ili "0", ovisno o dužini trajanja visoke razine (slika 3.8). Pročitani broj bitova se broji i sprema u char polje. Nakon što je pročitano 5 bajtova, ispituje se ispravnost primljenih podataka, te se podaci iz jezgrenog prostora šalju u korisnički prostor.

Implementacija ovog jezgrenog modula se bazira na implementaciji upravljačkog programa iz [10], uz male promjene autora diplomskog rada.



Slika 4.7: Prikaz implementacije sustava senzora DHT11

Kao i svaki modul, i ovaj modul sadrži *init* i *exit* funkcije. Funkcija *init* se poziva prilikom pokretanja modula naredbom *insmod*, a funkcija *exit* prilikom zaustavljanja modula naredbom *rmmod*:

```
static int __init dht11_init_module(void);
static void __exit dht11_exit_module(void);

module_init(dht11_init_module);
module_exit(dht11_exit_module);
```

Unutar *init* funkcije registriramo device sa željenim *major* device brojem, i pozivamo *init_port()* funkciju kojom inicijaliziramo GPIO memorijski prostor:

```
register_chrdev(driverno, DHT11_DRIVER_NAME, &fops);
init_port();
```

Exit funkcijom oslobađamo zauzeti memorijski prostor i oslobađamo *major* device broj:

```
iounmap(gpio);
unregister_chrdev(driverno, DHT11_DRIVER_NAME);
```

Unutar jezgrenog modula su ostvarene operacije *read, open* i *release*:

```
static struct file_operations fops = {
        .read = device_read,
        .open = read_dht11,
        .release = close_dht11
};
```

Kada želimo čitati device datoteku naredbom "cat */dev/dht11*" pokreće se *open* operacija čiji je rukovatelj funkcija *read_dht11()*. Ona je zapravo incijalizacijska funkcija koja se pokreće prilikom otvaranja datoteke:

```
static int read_dht11(struct inode *inode, struct file *file);
```

Unutar nje se postavljaju sve varijable i polja podataka na nulu i šalje se *Start* signal, te se GPIO pin postavlja kao ulazni, kako bi mogao primati podatke sa DHT11:

```
GPIO_DIR_OUTPUT(gpio_pin);  //Set pin to output
GPIO_CLEAR_PIN(gpio_pin);   //Set low
mdelay(20);                 //DHT11 needs min 18mS to signal a startup
GPIO_SET_PIN(gpio_pin);     //Take pin high
udelay(40);                 //Stay high for a bit before swapping to read mode
GPIO_DIR_INPUT(gpio_pin);   //Change to read
```

Također započinje mjerenje vremena od početka čitanja podataka i postavlja se prekidna rutina sljedećim funkcijama:

```
do_gettimeofday(&lasttv);   //Start timer to time pulse length
setup_interrupts();
```

Ako su primljeni podaci ispravni (*checksum* provjera) čitanje završava i podaci se formatirano spremaju u char polje iz kojeg će se podaci slati u korisnički prostor

kada se to od devicea zatraži, a ako su podaci neispravni pokreće se ponovni pokušaj čitanja podataka.

Ako je nakon izvršavanja naredbe "cat */dev/dht11*" *open* operacija bila uspješna, pokreće se *read* operacija čiji je rukovatelj *device_read()*:

```
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
                           loff_t * offset);
```

Ona je zadužena za slanje podataka iz jezgrenog u korisnički prostor funkcijom *put_user()*:

```
put_user(*(msg_Ptr++), buffer++);
```

Unutar *setup_interrupts()* se izdaje zahtjev za prekidnom linijom i registrira se funkcija prekidne rutine pomoću:

```
request_irq(INTERRUPT_GPIO0, (irq_handler_t) irq_handler, 0,
            DHT11_DRIVER_NAME, (void*) gpio);
```

Također se uključuje detektiranje prekida i briše se prekidna zastavica sljedećim naredbama:

```
GPIO_INT_RISING(gpio_pin, 1);   //GPREN0 GPIO Pin Rising Edge Detect Enable
GPIO_INT_FALLING(gpio_pin, 1);  //GPFEN0 GPIO Pin Falling Edge Detect Enable
GPIO_INT_CLEAR(gpio_pin);       //clear interrupt flag
```

Svaka promjena na pinu izaziva prekid i poziva prekidnu rutinu. Funkcija prekidne rutine mjeri trajanje svake razine signala na pinu, i na temelju toga odlučuje je li primljeni podatak "1" ili "0". Podatke sprema u polje i broji broj pročitanih bitova odnosno bajtova:

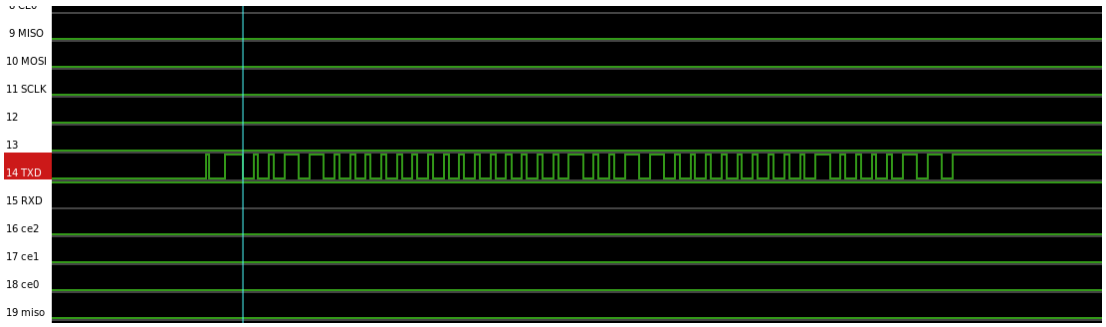
```
static irqreturn_t irq_handler(int i, void *dev_id, struct pt_regs *regs);
```

Kada proces zatvori device datoteku poziva se funkcija *close_dht11()*. Ona poziva funkciju *clear_interrupts()* kojom se isključuje detektiranje prekida i oslobađa prekidna linija:

```
static int close_dht11(struct inode *inode, struct file *file);
clear_interrupts();
```

```
GPIO_INT_RISING(gpio_pin, 0);   //GPREN0 GPIO Pin Rising Edge Detect Disable
GPIO_INT_FALLING(gpio_pin, 0);  //GPFEN0 GPIO Pin Falling Edge Detect Disable

free_irq(INTERRUPT_GPIO0, (void *) gpio);
```

Na slici 4.8 je prikazan GPIO signal senzora DHT11 koji se sastoji od *Response* signala i primljenih podataka sa senzora.



Slika 4.8: *Bit banging* DHT11 GPIO signala

4.3. DHT22

Implementacija upravljačkog programa DHT22 je identična implementaciji DHT11, jer senzori koriste potpuno isti komunikacijski protokol, osim što senzor DHT22 još prikazuje i decimalne vrijednosti izmjerenih podataka jer ima veću preciznost.

4.4. Dinamičko alociranje *major* brojeva

Kao autor upravljačkog programa, imamo dva izbora: izabrati broj koji se čini slobodnim, ili dinamički alocirati major broj. Samostalno biranje broja će raditi tako dugo dok smo mi jedini korisnik svojeg upravljačkog programa; jednom kada je naš upravljački program šire rasprostranjen, nasumično biranje major broja dovesti će do konflikata i problema.

Nedostatak dinamičkog dodjeljivanja je u tome što ne možemo stvoriti device datoteke unaprijed, jer će dodijeljeni major broj varirati. Za normalno korištenje upravljačkog programa, to neće biti problem, jer jednom kada je broj dodijeljen, možemo ga čitati iz */proc/devices*.

Za učitavanje upravljačkog programa pozivanje naredbe *insmod* se može zamijeniti jednostavnom skriptom koja, nakon što se pozove *insmod*, čita */proc/devices* kako bi stvorila device datoteku (Kod 1).

Skripta za učitavanje modula kojem je bio dodijeljen dinamički broj može biti napisana pomoću alata *awk* za dohvaćanje informacija iz */proc/devices* kako bi stvorila datoteke u */dev* [4].

```
#!/bin/bash
module="ds18b20"
device="ds18b20"
mode="664"
# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./$module.ko $* || exit 1
# remove stale nodes
rm -f /dev/${device}
major=$(awk -v mod="$module" '$2==mod{print $1}' /proc/devices)
mknod /dev/${device} c $major 0
# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}
chmod $mode  /dev/${device}
```

Kod 1: Stvaranje device datoteke za senzor DS18B20

4.5. Ispis rezultata

```
#!/bin/bash
#cat all sensors
RED='\033[0;31m'
WHITE='\033[1;37m'
LIGHT_BLUE='\033[1;34m'
NC='\033[0m' # No Color

printf "${RED}ds18b20:${NC}\n"
cat /dev/ds18b20
printf "${LIGHT_BLUE}dht11:${NC}\n"
cat /dev/dht11
printf "${WHITE}dht22:${NC}\n"
cat /dev/dht22
```

Kod 2: Bash skripta (*cat.sh*) za ispis rezultata

Skripta za ispis rezultata (Kod 2) tri puta uzastopno pokreće naredbu “cat” kojom se zapravo čitaju podaci iz device datoteka, od kojih je svaka povezana sa svojim

upravljačkim programom. U skripti su definirane boje koje olakšavaju čitanje ispisanih podataka sa naredbenog retka. Senzor DS18B20 je označen crvenom bojom, senzor DHT11 svijetlo plavom, a DHT22 bijelom bojom.

5. Upute za korištenje

5.1. Prevođenje modula

Pokretanjem skripte *make_all.sh* prevodimo sve module odjednom, tj. pokreće se naredba *make* u svakom od direktorija u kojem se nalazi izvorni kod modula. Svaki senzor ima svoj Makefile, npr. za DS18B20:

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := ds18b20.o

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
	$(MAKE) -C $(KDIR) M=$$PWD

clean:
	$(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

Kod 3: Makefile DS18B20 jezgrenog modula

Za prevođenje modula nam je potreban izvorni kod jezgre kojeg možemo jednostavno dohvatiti skriptom *rpi-source*, koja se nalazi na adresi [11].

5.2. Pokretanje, resetiranje i zaustavljanje modula

Pokretanjem skripte *start_all.sh* pokrećemo sve module koji su potrebni za komunikaciju sa senzorima. Za DS18B20 se pokreću moduli *w1-gpio*, *w1_therm* koji su već u jezgri i potrebni su modulu *ds18b20*. Za senzore DHT11 i DHT22 se pokreću moduli *dht11* i *dht22*.

Sve module možemo resetirati naredbom *restart_all.sh*, a u potpunosti zaustaviti module i izbrisati sve device datoteke naredbom *stop_all.sh*.

Ulaskom u pojedini direktorij senzora možemo zasebno pokrenuti, resetirati i zaustaviti modul naredbama *start.sh*, *restart.sh* i *stop.sh*.

Naredbom *clean_all.sh* se izvršava *make clean* za sve senzore.

5.3. Ispis rezultata

Pokretanjem skripte *cat.sh* ispisat će se podaci dohvaćeni iz svih senzora (slika 5.1). DS18B20 prikazuje samo temperaturu jer nema senzor vlage, a DHT11 i DHT22 ispisuju stanje temperature i vlage.

```
pi@raspberrypi ~/diplomski $ ./cat.sh
ds18b20:
Temperature: 25.687C
Humidity: N/A
Result: OK
dht11:
Temperature: 26C
Humidity: 45
Result: OK
dht22:
Temperature: 26.3C
Humidity: 62.4
Result: OK
```

Slika 5.1: Ispis rezultata svih senzora

Svi senzori ispisuju je li dohvaćen podatak ispravan, a provjera je implementirana interno u svakom od upravljačkih programa. Ako je podatak ispravan ispisuje se “Result: OK”, a u suprotnom “Result: BAD”.

6. Zaključak

Nakon obrađene teme možemo potvrditi da se jezgreni moduli doista veoma razlikuju od korisničkih aplikacija. Jezgreni moduli se izvode u jezgrenom prostoru, a aplikacije se izvode u korisničkom prostoru. Dok većina korisničkih aplikacija obavlja neki zadatak slijedno, od početka do kraja, svaki se jezgreni modul samo registrira kako bi posluživao buduće zahtjeve. Jezgreni modul mora biti u mogućnosti izvršavati više zadaća odjednom i biti višenastupni (*eng. reentrant*). Uz sve operacije koje pruža, obavezno sadrži *init* i *exit* funkcije koje se pokreću prilikom učitavanja modula u sustav, odnosno zaustavljanja modula. Ovakav je pristup programiranju sličan programiranju koje se temelji na upravljanju uvjetovanom događajima. Prilikom pisanja upravljačkih programa potrebno je pridržavati se strogih pravila, kako bi oni bili što stabilniji i sigurniji. Upravljački program ne smije nametati svoju politiku korisniku, već samo pružati željene mehanizme. Mogućnost dinamičkog učitavanja modula u jezgru, za vrijeme njezinog izvođenja, doista pomaže u pisanju i testiranju jezgrenih modula.

POPIS SLIKA

POPIS IZVORNIH KODOVA

LITERATURA

[1] Adafruit. *DHT11 basic temperature-humidity sensor + extras*. URL https://www.adafruit.com/product/386. (datum pristupa: 20.6.2016.).

[2] Tarun Agarwal. *Types of Analog and Digital Sensors with Applications*. URL https://www.elprocus.com/types-analog-digital-sensors. (datum pristupa: 20.6.2016.).

[3] IBM Knowledge Center. *Device names, device nodes, and major/minor numbers*. URL https://www.ibm.com/support/knowledgecenter/linuxonibm/com.ibm.linux.z.lgdd/lgdd_c_udev.html. (datum pristupa: 16.6.2016.).

[4] Jonathan Corbet, Alessandro Rubini, i Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2005.

[5] Raspberry Pi Foundation. *GPIO: Raspberry Pi Models A and B*. URL https://www.raspberrypi.org/documentation/usage/gpio. (datum pristupa: 23.6.2016.).

[6] Maxim Integrated. *Programmable Resolution 1-Wire Digital Thermometer*. URL https://www.maximintegrated.com/en/products/analog/sensors-and-sensor-interface/DS18B20.html. (datum pristupa: 20.6.2016.).

[7] Maxim Integrated. *Programmable Resolution 1-Wire Digital Thermometer*. Maxim Integrated Products, Inc., 2015. URL https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf.

[8] joan2937. *piscope*. URL http://abyz.co.uk/rpi/pigpio/piscope.html. (datum pristupa: 29.6.2016.).

[9] Greg Kroah-Hartman. *Driving Me Nuts - Things You Never Should Do in the Kernel*. URL http://www.linuxjournal.com/article/8110. (datum pristupa: 23.6.2016.).

[10] Nigel Morton. *RaspberryPi DHT11 temperature and humidity sensor driver*. URL http://www.tortosaforum.com/raspberrypi/dht11driver.htm. (datum pristupa: 28.6.2016.).

[11] notro. *Raspberry Pi kernel source installer*. URL https://github.com/notro/rpi-source. (datum pristupa: 16.6.2016.).

[12] R-B. *Measurement of temperature and relative humidity using DHT11 sensor and PIC microcontroller*. URL http://embedded-lab.com/blog/measurement-of-temperature-and-relative-humidity-using-dht11-se\\nsor-and-pic-microcontroller. (datum pristupa: 21.6.2016.).

Dodatak A

Izvorni kodovi

A.1. ds18b20/ds18b20.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/types.h>     // for dev_t typedef
#include <linux/kdev_t.h>    // for format_dev_t
#include <linux/fs.h>        // for alloc_chrdev_region()
#include <linux/cdev.h>      // for "struct cdev"
#include <asm/uaccess.h>     // for user/kernel space copy routine
#include <linux/string.h>

static dev_t dev_num;        // (major,minor) value
static char buffer[100];     // buffer for read_file() function
static char temp[100];
int ret;
#define DEVICE_NAME "ds18b20"

static int read_file(char *filename)
{
    struct file *fp;
    loff_t pos;
    mm_segment_t old_fs;

    fp = filp_open(filename, O_RDONLY, 0);
    if(IS_ERR(fp)){
        return -1;
    }

    memset(buffer,0,sizeof(buffer));
    memset(temp,0,sizeof(temp));

    old_fs = get_fs();
    set_fs(KERNEL_DS);
    pos = 69; //read form position 69 to end (only temperature value)
    vfs_read(fp, buffer, sizeof(buffer), &pos);
    sprintf(temp, "Temperature: %c%c.%c%c%cC\nHumidity: N/A\nResult:OK\n",
            buffer[0], buffer[1], buffer[2], buffer[3], buffer[4]);
```

```
    filp_close(fp, NULL);
    set_fs(old_fs);
    return 0;
}

ssize_t
my_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    if (buffer[*f_pos] == '\0'){
        printk(KERN_INFO "%s: End of string, returning zero.\n", DEVICE_NAME);
        return 0;
    }

    ret = read_file("/sys/bus/w1/devices/28-000005782b72/w1_slave");
    if (ret < 0){
        printk(KERN_WARNING
               "%s: unable to read DS18B20, no file in /sys/bus/w1/devices?\n",
               DEVICE_NAME);
        return ret;
    }

    ret = copy_to_user(buf, temp, strlen(temp));
    if (ret > 0){
        return -EFAULT;
    }

    *f_pos += strlen(temp);
    ret = strlen(temp);
    return ret;
}

struct cdev my_cdev;

struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .read = my_read
};

static int __init ds18b20_in(void)
{
    printk(KERN_INFO "%s: module %s being loaded.\n", DEVICE_NAME,
           DEVICE_NAME);

    ret = alloc_chrdev_region(&dev_num, 0, 1, DEVICE_NAME);
    if (ret < 0){
        printk(KERN_WARNING
               "%s: couldn't allocate major and minor number(s).\n",
               DEVICE_NAME);
        return ret;
    }

    printk(KERN_INFO "%s: Allocated [major:minor] number: [%s]\n",
           DEVICE_NAME, format_dev_t(buffer, dev_num));

    cdev_init(&my_cdev, &my_fops);
```

```
    my_cdev.owner = THIS_MODULE;

    ret = cdev_add(&my_cdev, dev_num, 1);
    if (ret < 0){
        printk(KERN_WARNING "%s: unable to add cdev to kernel\n", DEVICE_NAME);
        return ret;
    }
    return 0;
}

static void __exit ds18b20_out(void)
{
    printk(KERN_INFO "%s: module ds18b20 being unloaded.\n", DEVICE_NAME);

    cdev_del(&my_cdev);
    unregister_chrdev_region(dev_num, 1);
}

module_init(ds18b20_in);
module_exit(ds18b20_out);

MODULE_AUTHOR("Benjamin Horvat");
MODULE_LICENSE("GPL");
}
```

A.2. ds18b20/Makefile

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := ds18b20.o

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
        $(MAKE) -C $(KDIR) M=$$PWD

clean:
        $(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

A.3. ds18b20/start.sh

```
#!/bin/bash
module="ds18b20"
device="ds18b20"
mode="664"
# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
```

```
/sbin/insmod ./$module.ko $* || exit 1
# remove stale nodes
rm -f /dev/${device}
major=$(awk -v mod="$module" '$2==mod{print $1}' /proc/devices)
mknod /dev/${device} c $major 0
# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}
chmod $mode  /dev/${device}
```

A.4. ds18b20/restart.sh

```
#!/bin/bash
module="ds18b20"
device="ds18b20"
mode="664"
# remove module with rmmod
/sbin/rmmod ./$module.ko
# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./$module.ko $* || exit 1
# remove stale nodes
rm -f /dev/${device}
major=$(awk -v mod="$module" '$2==mod{print $1}' /proc/devices)
mknod /dev/${device} c $major 0
# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}
chmod $mode  /dev/${device}
```

A.5. ds18b20/stop.sh

```
#!/bin/bash
module="ds18b20"
device="ds18b20"
# remove module with rmmod
# and use a pathname, as newer modutils don't look in . by default
/sbin/rmmod ./$module.ko
# remove stale nodes
rm -f /dev/${device}
```

A.6. DHT11 i DHT22

DHT11/DHT22 Makefile i bash skripte za pokretanje, resetiranje i zaustavljanje modula su samo modifikacije datoteka senzora DS18B20, a izvorni kod upravljačkog programa se može preuzeti sa [10].

A.7. make_all.sh

```
#!/bin/bash
#make
cd ds18b20
make
cd ..
cd dht11km
make
cd ..
cd dht22km
make
```

A.8. clean_all.sh

```
#!/bin/bash
#clean
cd ds18b20
make clean
cd ..
cd dht11km
make clean
cd ..
cd dht22km
make clean
```

A.9. start_all.sh

```
#!/bin/bash
#kernel module start script
./clean_all.sh
./make_all.sh
sudo modprobe w1-gpio
sudo modprobe w1_therm
cd ds18b20
sudo ./start.sh
cd ..
cd dht11km
```

```
sudo ./start.sh
cd ..
cd dht22km
sudo /sbin/insmod ./dht22km.ko gpio_pin=18 format=4
sudo rm -f /dev/dht22
sudo mknod /dev/dht22 c 81 0
```

A.10. restart_all.sh

```
#!/bin/bash
#kernel module restart script
./clean_all.sh
./make_all.sh
sudo rmmod w1_therm
sudo modprobe w1_therm
cd ds18b20
sudo ./restart.sh
cd ..
cd dht11km
sudo ./restart.sh
cd ..
cd dht22km
sudo rmmod dht22km
sudo insmod dht22km.ko gpio_pin=18 format=4
sudo rm -f /dev/dht22
sudo mknod /dev/dht22 c 81 0
```

A.11. stop_all.sh

```
#!/bin/bash
#kernel module stop script
cd ds18b20
sudo ./stop.sh
cd ..
sudo rmmod w1_therm
cd dht11km
sudo ./stop.sh
cd ..
cd dht22km
sudo rmmod dht22km
sudo rm -f /dev/dht22
```

A.12. cat.sh

```
#!/bin/bash
#cat all sensors
```

```
RED='\033[0;31m'
WHITE='\033[1;37m'
LIGHT_BLUE='\033[1;34m'
NC='\033[0m' # No Color

printf "${RED}ds18b20:${NC}\n"
cat /dev/ds18b20
printf "${LIGHT_BLUE}dht11:${NC}\n"
cat /dev/dht11
printf "${WHITE}dht22:${NC}\n"
cat /dev/dht22
```

Implementacija senzorskog sučelja upravljačkim programom na operacijskom sustavu Linux

Sažetak

Tema ovog diplomskog rada je implementacija senzorskog sučelja upravljačkim programom na operacijskom sustavu Linux. Započinje uvodom, nakon kojeg slijedi 2. poglavlje u kojem je objašnjeno što je Linux jezgra, a što su upravljački programi. Razrađuju se vrste upravljačkih programa, njihov način rada i struktura. Objašnjava se razlika između korisničkog i jezgrenog prostora, na koji način se moduli učitavaju u jezgru, kako se obrađuju prekidi i što je to usporedno izvođenje u jezgri. U 3. poglavlju se obrađuju temperaturni senzori — njihova struktura, performanse i komunikacijski protokoli. U 4. poglavlju prikazano je fizičko ostvarenje sustava, te je objašnjena implementacija izvornih kodova upravljačkih programa, nakon čega slijedi 5. poglavlje — Upute za korištenje.

Ključne riječi: Upravljački program, jezgreni modul, char device, temperaturni senzor, Linux, DS18B20, DHT11, DHT22, Raspberry Pi.

Sensor interface implementation on Linux operating system using device driver

Abstract

The subject of this thesis is sensor interface implementation on Linux operating system using device driver. It begins with an introduction, followed by the 2. chapter which explaines what a Linux kernel is, and what device drivers are. It is elaborated upon the type of drivers, their mode of operation and structure. The difference between the user and kernel space, how the modules are loaded into the kernel, how to handle interrupts and what is concurrency in the kernel is also explained in it. The 3. chapter discusses the temperature sensors — their structure, performance and communication protocols. The 4. chapter shows the physical realization of the system and the implementation of the drivers source code, followed by the 5. chapter — User Manual.

Keywords: Device driver, kernel module, char device, temperature sensor, Linux, DS18B20, DHT11, DHT22, Raspberry Pi.