SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. **90**

IZRADA UPRAVLJAČKIH PROGRAMA ZA OPERACIJSKI SUSTAV UCLINUX

Ivan Vukosav

Zagreb, lipanj 2010.

Sažetak

Unutar ovoga rada obrađena je tema izrade upravljačkih programa za operacijski sustav uCLinux. Na početku rada naglasak je stavljen na opisu instalacije potrebnih programskih alata i sklopovlja. Slijedi opis arhitekture, organizacije i mogućnosti operacijskog sustava Linux, s posebnim naglaskom na uCLinux koji se koristi za procesore bez MMU jedinice. U narednom poglavlju detaljno je opisan bootloader U-boot i njegova veza s operacijskim sustavom uCLinux. Pisanje upravljačkog programa za uCLinux na procesoru LPC2478 opisano je u sljedećem poglavlju. Kao ogledni primjer upravljačkog programa za razvojni sustav temeljen na ARM7TDMI jezgri procesora opisano je upravljanje CAN modulom.

Sadržaj

Popis oznaka i kratica

engl.	engleski
MMU	jedinica za upravljanje memorijom
uClinux	posebno prilagođena inačica operacijskog sustava Linux za procesore bez MMU
FER	Fakultet elektrotehnike i računarstva, Zagreb
GPL	najšire korištena licenca za slobodne programe
FSF	organizacija koja se brine za GPL licencu
NXP	poluvodička kompanija koja se bavi proizvodnjom elektroničkih komponenti osnovana od Philips kompanije 2006. godine
JTAG	ustaljen naziv za standard IEEE 1149.1 koji se danas široko koristi kao sučelje za ispravljanje pogrešaka prilikom izvođenja aplikacije na određenom procesoru
USB	skraćenica za *Universal Serial Bus* i predstavlja tehnološko rješenje spajanja vanjskih uređaja s računalom. Podaci se razmjenjuju serijski, a brzina razmjene podataka za inačicu USB 2.0 iznosi 480 Mbps.
RS232	RS-232 ili EIA RS-232C je standardni međusklop za serijski prijenos binarnih podataka između datotečne spojne opreme DTE (engl. *Data terminal equipment*) i datotečne komunikacijske opreme DCE (engl. *Data communication equipment*).
GCC	standardni C kompajler za operativne sustave koji su slični Unix-u.
API	skraćenica za engleski izraz *application programming interface* koji označava programsko sučelje za programiranje aplikacija.

Popis tablica

Popis slika

1. Uvod

Linux je ime za operacijski sustav koji je baziran na Linux jezgri (engl. *kernel*) čiji je izvorni autor finski računarac Linus Torvalds od koga potječe i naziv za jezgru. U ranim fazama razvoja Linux je bio dostupan jedino u izvornom kôdu (engl. *source code*) za pojedince koji su imali dovoljno znanja i iskustva da kompajliraju[1] i instaliraju takav operacijski sustav. Ranih devedesetih pojavila se želja za osnivanjem razvojnih udruženja preko interneta koja bi trebala pomoći pri razvoju prve distribucije Linux operacijskog sustava. Takve distribucije će sadržavati sve programske komponente potrebne za jednostavnu instalaciju i održavanje Linux operacijskog sustava, tj. moći će ga koristiti pojedinci koji nemaju specifična tehnička znanja. To je dovelo do sveprisutnosti Linux operacijskog sustava. Sveprisutnost se očituje u prilagodbi Linux-a raznim aplikacijama u rasponu od ugradbenih (engl. *embedded*) računalnih sustava pa do složenih poslužiteljskih (engl. *server*) računala. Linux se može koristiti i distribuirati pod posebnom licencom za slobodne programe – GPL licencom koju je izvorno osmislio Richard Stallman, a o kojoj se danas brine organizacija FSF (engl. *free software foundation*) pod njegovim vodstvom.

Unutar ovoga rada naglasak će biti usmjeren prema korištenju Linux operacijskog sustava za ugradbene računalne sustave, posebice pisanje upravljačkih programa (engl. *device driver*) za posebnu vrstu Linux operacijskog sustava koja je prilagođena za procesore bez MMU jedinice – uCLinux. Jezgra procesora na kojoj se izvodi uCLinux bazirana je na ARMv4T arhitekturi koju implementira procesor pod nazivom ARM7TDMI-S. Unutar širokog spektra proizvođača mikrokontrolera koji se zasnivaju na spomenutom procesoru odabran je NXP-ov LPC2478 mikrokontroler koji je ugrađen u Olimex-ov razvojni sustav pod nazivom LPC2478STK. Za spomenuti razvojni sustav napravljen je i detaljno opisan uCLinux upravljački program za CAN (engl. *controller area network*) modul.

[1] U radu će se koristiti posuđenica iz engleskog jezika za proces prevođenja izvornog kôda.

2. uCLinux

uCLinux (engl. *microcontroller linux*) je posebna verzija operacijskog sustava Linux koja je prilagođena za procesore bez sklopovske jedinice za upravljanje memorijom (MMU). Nastao je 1998. godine kada su dvojica programera (D. Jeff Dionne i Kenneth Albanowski) pokušavala instalirati jezgru Linux operacijskog sustava (verzija 2.0.33) na Motorolin 68k procesor obitelji *DragonBall*. Nakon što su javno objavili svoj rad potaknuli su stvaranje uCLinux organizacije koja će nastaviti istim putem prilagodbe jezgre Linux operacijskog sustava procesorima bez MMU jedinice. Godine 1999. podrška je dodana za Motorolinu *ColdFire* obitelj ugradbenih procesora, a narednih godina (do 2004. godine) podrška je proširena na najpoznatije 32b procesore - ARM. Kako se razvijala standardna jezgra operacijskog sustava Linux tako se usporedno razvijala jezgra uCLinux operacijskog sustava. Danas se ostvarila namjera brojnih programera da se unutar standardne Linux jezgre nalazi podrška i za procesore bez MMU jedinice. To je dovelo do toga da uCLinux danas više nije odvojena frakcija Linux-a. Također, organizacija pod istim imenom (http://www.uclinux.org) je napravila prvu distribuciju uCLinux-a koju konstantno razvijaju i dodaju podršku za razne druge arhitekture procesora.

uCLinux se koristi u raznim uređajima poput nekih DVD *player*-a, video kamera, mrežnih preusmjerivača (engl. *router*) i drugih. Najpoznatiji komercijalni primjer korištenja je Apple-ov iPod kojeg možemo vidjeti na sljedećoj slici. Naravno, postoji još mnogo platformi u kojima se nalazi uCLinux, ali mnoge od njih nisu toliko poznate.

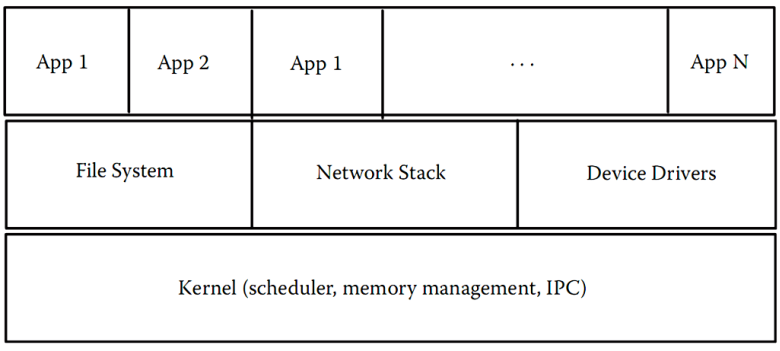
Slika 1. Apple iPod

2.1. Arhitektura

Standardni Linux operacijski sustav ima monolitičku arhitekturu, što znači da se cijeli operacijski sustav, osim korisničkih aplikacija, izvodi u jezgrinom prostoru (engl. *kernel space*) koristeći se pritom povlaštenim (*supervisor*) načinom rada procesora. Općenito, operacijski sustavi dolaze u 3 različita oblika arhitekture: arhitektura za rad u stvarnom vremenu, monolitička i mikrokernel arhitektura. Glavno svojstvo po kojem se vrši ovakva klasifikacija je način na koji operacijski sustav koristi sklopovlje za zaštitu i sigurnost operacijskog sustava u cjelini.

2.1.1. Arhitektura za rad u stvarnom vremenu

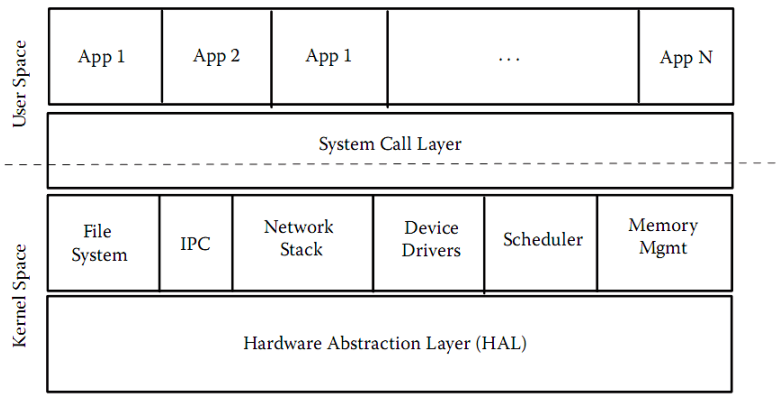
Ovakav tip arhitekture je zamišljen za procesore bez sklopovske jedinice za upravljanje memorijom (MMU). Kod operacijskih sustava koji implementiraju ovakvu arhitekturu cijeli adresni prostor je linearan i nema zaštite memorije između jezgre operacijskog sustava i aplikacija (vidi sliku 2). Dakle, jezgra takvog operacijskog sustava i njezini podsustavi dijele jednak adresni prostor s aplikacijama koje se nalaze u korisničkom prostoru (engl. *user space*). Prilagođeni su radu u stvarnom vremenu jer nema nepotrebnih sistemskih poziva i ostalih mehanizama koje bi ovakav operacijski sustav potakle na veće iskorištenje memorije. Rizično je dodavati nove aplikacije jer mogu doprinijeti nestabilnosti cijelog operacijskog sustava, tj. nakon što smo ih dodali potrebno je podrobno ispitati i testirati cijeli računalni sustav. Također, ne možemo dodavati aplikacije i jezgrene module dinamički dok se jezgra operacijskog sustava izvršava nego moramo statički povezati aplikacije i module prilikom kompajliranja jezgre. Primjer takvog operacijskog sustava je poznati RTOS. Ipak, ovakvi operacijski sustavi široko se primjenjuju kod jednostavnijih procesora za upravljanje različitim



Slika 2. Arhitektura za rad u stvarnom vremenu

2.1.2. Monolitička arhitektura

Cijena memorije je uvijek ograničavala količinu programa koji se može izvoditi na ugradbenom računalnom sustavu. Kako se snižavala cijena važnih komponenti ugradbenog računalnog sustava (posebice memorije) tako su računalni sustavi imali na sebi svi više složenijih programa koji su se mogli razdvojiti u dvije kategorije: sistemski programi i aplikacije. Budući da prethodna arhitektura nije više mogla zadovoljiti takve zahtjeve pojavila se potreba za novijim arhitekturama koje su zahtijevale da procesori imaju jednu posebnu sklopovsku jedinicu koja bi upravljala memorijom i provodila određivanje prava pristupa pojedinim dijelovima memorije od strane sistemskih programa (poput upravljačkih programa) i aplikacija. Monolitička arhitektura, prikazana na sljedećoj slici, se razlikuje od drugih arhitektura upravo po tome što definira apstraktno programsko sučelje koje omogućava upravljanje sklopovljem računalnog sustava, a također definira skup sistemskih poziva kako bi se implementirali mehanizmi poput upravljanja procesima, izbjegavanja konkurentnosti i upravljanja memorijom. Budući da je podržana od MMU jedinice moguće je jasno odvojiti jezgreni prostor od korisničkog prostora adresa. Kada se aplikacija izvršava u korisničkom prostoru onda ne može pristupiti sklopovlju ili izvršiti neke privilegirane instrukcije procesora. Koristeći posebne sistemske pozive aplikacije mogu ući u jezgrin način rada i obaviti neke privilegirane operacije. Korisničke aplikacije se izvršavaju u virtualnom prostoru adresa pa stoga ne mogu na bilo koji način ugroziti drugu aplikaciju ili jezgrenu memoriju. Budući da se svi ostali jezgreni moduli izvršavaju u jezgrinom načinu rada tada loše projektiran upravljački program ili bilo koji drugi jezgrin modul mogu ozbiljno ugroziti cijeli sustav.

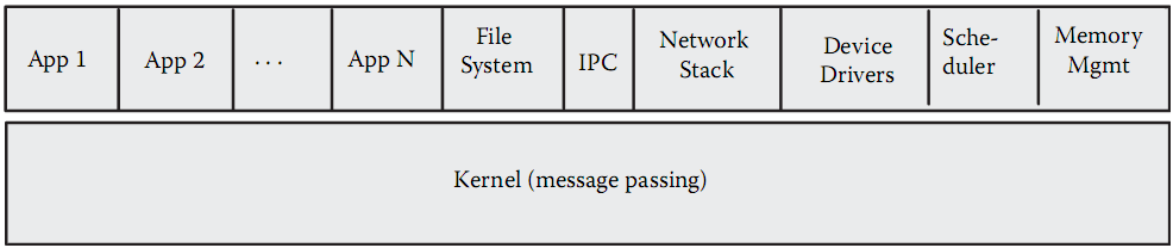


Slika 3. Monolitička arhitektura

Operacijski sustav uCLinux se zasniva na prijelaznoj arhitekturi između monolitičke i arhitekture za rad u stvarnom vremenu. Budući da je nastao iz operacijskog sustava Linux koji ima monolitičku arhitekturu naslijedio je takav način programiranja, a zbog nedostatka MMU jedinice svrstavamo ga u grupu operacijskih sustava za rad u stvarnom vremenu.

2.1.3. Mikrokernel arhitektura

Ova arhitektura je dugo vremena smatrana najboljim izborom s obzirom na principe izgradnje operacijskih sustava, ali praksa je pokazala da ipak postoje problemi koji su uzrokovali da ovakva arhitektura bude značajno manje zastupljena na tržištu. Arhitektura je prikazana na sljedećoj slici na kojoj možemo vidjeti da su datotečni sustav, upravljački programi i mrežno sučelje izvan jezgre operacijskog sustava, kao i aplikacije. Svaki podsustav ima svoje vlastito adresno područje pa je ključna uloga jezgre da osigura sigurnu i brzu komunikaciju podsustava (engl. *message passing*). Postoje brojne rasprave na temu optimalne arhitekture za ugradbene računalne sustave gdje se na jednoj strani vage nalazi složenija sklopovska podrška, a na drugoj složenija programska podrška. Ova arhitektura operacijskog sustava se zasniva na složenijoj programskog podršci. U detalje takvih rasprava nećemo ulaziti unutar ovog rada.

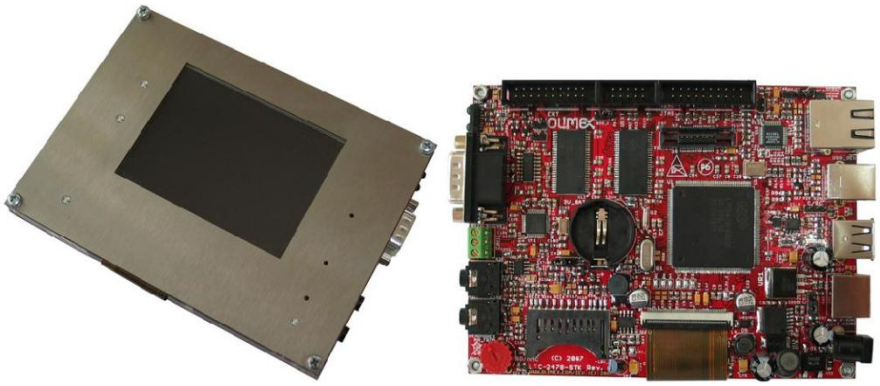


Slika 4. Mikrokernel arhitektura

3. Sklopovska podrška

3.1. Razvojni sustav

Razvojni sustav LPC2478STK koji je razvila kompanija Olimex korišten je prilikom izrade ovoga rada. Na slici 1 prikazane su obje strane dotičnog razvojnog sustava. Razvojni sustav se napaja izvorom napajanja +(9-12)VDC, a sadrži i bateriju (CR2032, Li, 3V) koja napaja jedino 2KB unutrašnje SRAM memorije i sat stvarnog vremena (RTC – engl. real time clock). Potrošnja razvojnog sustava ovisi o privedenom napajanju, ali uz +10VDC potrošnja je oko 250mA.



Slika 5. Izgled korištenog razvojnog sustava s pogledom na obje strane

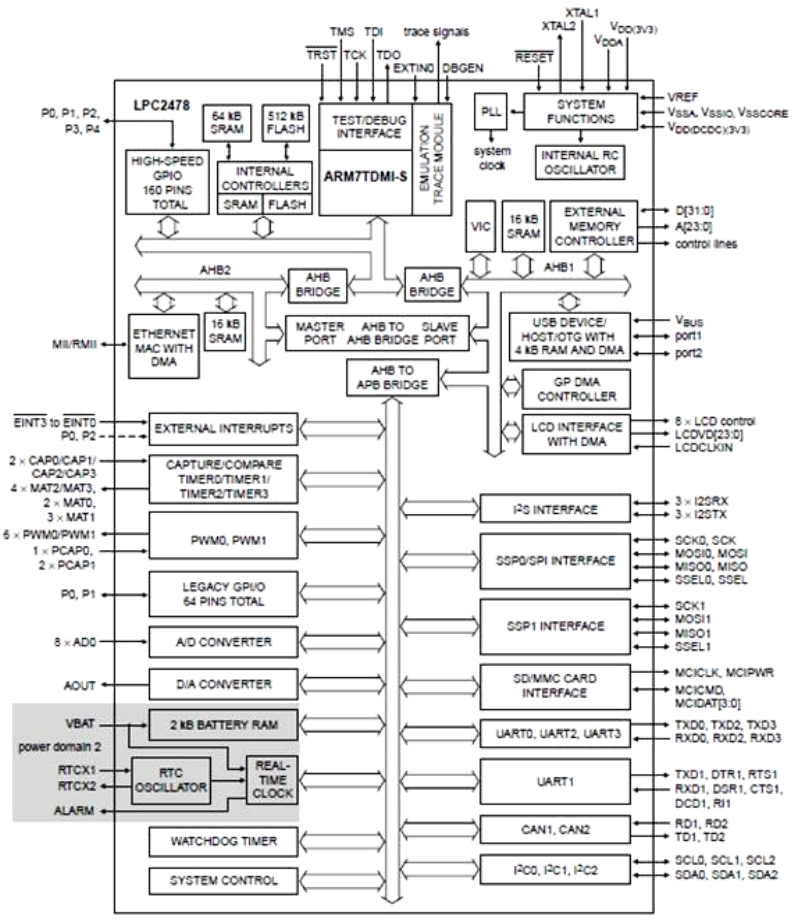
Spomenuti razvojni sustav koristi mikrokontroler proizvođača NXP pod nazivom LPC2478. Ovaj mikrokontroler se zasniva na ARM7TDMI-S procesoru s brzinom takta do 72 MHz. Procesor implementira ARMv4T arhitekturu i zbog svoje popularnosti izabran je kao glavni kriterij prilikom odabira ovog razvojnog sustava. ARM7TDMI-S procesor dio je familije ARM 32-bitnih procesora zasnovanih na RISC[2] (engl. *reduced instruction set computer*) principima smanjenog skupa naredbi. Jedan od glavnih principa je naredbeni *pipeline* koji je prikazan na sljedećoj slici, a sastoji se od tri operacije koje se mogu izvoditi istovremeno: dohvat naredbe, dekodiranje naredbe i izvođenje naredbe.

[2] Budući da su naredbe koje izvode takvi procesori jednostavne uvelike se dobiva na brzini izvođenja naredbi i jednostavnosti izvedbe sklopovlja

Slika 6. *Pipeline* struktura korištenog procesora

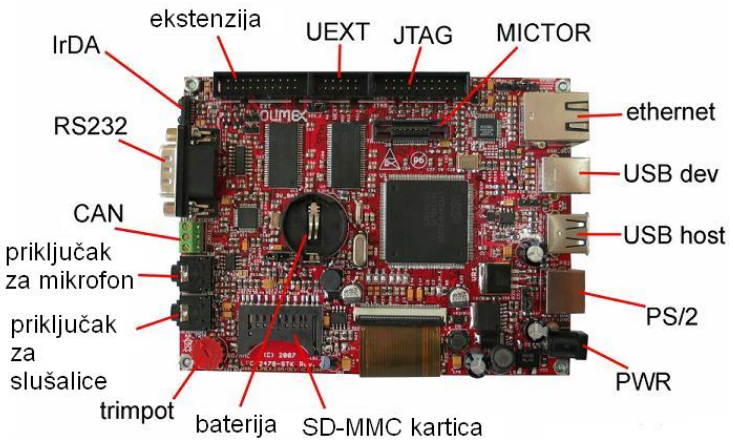
ARM7TDMI-S procesor zasniva se na Von Neumann-ovoj arhitekturi s jednom 32-bitnom sabirnicom kojom prolaze naredbe i podaci. Slovo „T“ unutar naziva označava da ovaj procesor podržava tzv. *thumb* skup naredbi širine 16-bita koji omogućava manju gustoću pakiranja kôda unutar memorije nasuprot 32-bitnom naredbenom skupu koji ima podržan veći skup naredbi od 16-bitnog, ali više zauzima memorijskog prostora. Slovo „D“ označava da spomenuti procesor pruža podršku za traženje pogrešaka (engl. *debugging*) prilikom izvođenja aplikacije na samom procesoru (najčešće preko JTAG sučelja). Slovo „M“ označava da procesor sadrži množilo izvedeno sklopovljem, a slovo „I“ označava da procesor sadrži posebnu sklopovsku jedinicu koja omogućava postavljenje *breakpoint*-a i *watchpoint*-a prilikom ispitivanja ispravnosti aplikacije na samom procesoru. Blok dijagram mikrokontrolera prikazan je na sljedećoj slici na kojoj možemo vidjeti količinu SRAM i FLASH memorije te podržane periferne module od kojih ističemo CAN modul za koji je implementiran upravljački program.

U detaljniji opis arhitekture spomenutog procesora i mikrokontrolera nećemo ulaziti jer to nije cilj ovoga rada pa se preporuča da čitatelj detaljnije pročita o arhitekturi procesora u literaturi [3] i o dotičnom mikrokontroleru u literaturi [4] radi boljeg razumijevanja sadržaja koji će uslijediti.



Slika 7. Blok dijagram korištenog mikrokontrolera

Budući da unutar mikrokontrolera nema dovoljno memorije za izvođenje jezgre operacijskog sustava uCLinux razvojni sustav sadrži dodatnih 64 MB SDRAM memorije. Među ostalim komponentama razvojnog sustava spomenimo USB sučelje, LCD TFT ekran, RS232 sučelje, dvije tipke, audio ulaz i izlaz, CAN primopredajnik i priključak itd. Na sljedećoj slici možemo vidjeti raspored svih priključaka na tiskanoj pločici razvojnog sustava.



Slika 8. Raspored priključaka korištenog razvojnog sustava

Razvojni sustav dolazi s ugrađenim programom pod nazivom U-boot koji omogućava podizanje operacijskog sustava uCLinux koji je u tom slučaju spremljen u vanjsku memoriju (prilikom projektiranja je korištena USB FLASH memorija) zbog već navedenih ograničenja veličine unutrašnje FLASH memorije mikrokontrolera LPC2478. U daljnjem tekstu za program U-boot ćemo koristiti naziv bootloader U-boot jer ovakvi programi vrše specifičnu funkciju punjenja memorije komprimiranom jezgrom operacijskih sustava.[3] Također, na pratećem CD-u mogu se naći razni alati među kojima je najvažniji *arm-linux-gcc* kompajler koji se koristi za kompajliranje jezgre operacijskog sustava uCLinux i bootloader-a U-boot. Za više informacija o korištenom razvojnom sustavu pogledajte literaturu [5] u kojoj se nalazi popis svih kratkospojnika i priključaka.

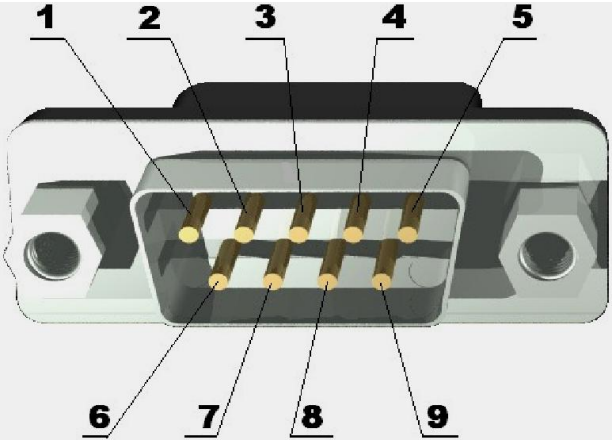
3.2. USB-CAN pretvornik

Kako bi provjerili ispravnost napisanog uCLinux upravljačkog programa za CAN modul opisanog razvojnog sustava korišten je uređaj pod nazivom USB-CAN pretvornik. Na sljedećoj slici prikazan je izgled navedenog uređaja proizvođača SYS TEC ELECTRONIC.

Slika 9. USB-CAN pretvornik

[3] U hrvatskom jeziku nema ustaljenog prijevoda za englesku riječ bootloader, ali se često koristi izraz „program punilac“

Funkcija tog uređaja se očituje u primanju poruka koje šalje CAN primopredajnik korištenog razvojnog sustava preko CAN sabirnice i slanje poruka primopredajniku razvojnog sustava. To nam pomaže u provjeri ispravnosti napisane aplikacije i upravljačkog programa pod operacijskim sustavom uCLinux. Uređaj sadrži DB-9 priključak za spajanje na CAN sabirnicu. Razmještaj pinova na spomenutom priključku je u skladu s CiA (engl. CAN in automation) specifikacijom i prikazan je u sljedećoj tablici. Također, možemo uočiti da nema priključka za napajanje jer se napajanje dovodi preko drugog priključka s kojim je USB-CAN pretvornik povezan s osobnim računalom.



Slika 10. Priključak DB-9

Tablica 1. Razmještaj pinova na priključku DB-9

1	N/C
2	CAN_L
3	GND
4	N/C
5	CAN *shiled*
6	GND
7	CAN_H
8	N/C
9	N/C

U sljedećoj tablici možemo vidjeti razmještaj pinova CAN priključka na razvojnom sustavu. U skladu s navedenim potrebno je napraviti kabel koji na propisan način spaja odgovarajuće pinove.

Slika 11. CAN priključak

Tablica 2. Razmještaj pinova na CAN priključku

1	CAN_H
2	CAN_L
3	GND

S druge strane USB-CAN pretvornika spajamo se na USB priključak osobnog računala kabelom prikazanim na donjoj slici. Na osobnom računalu se izvodi aplikacija koja unutar grafičkog sučelja korisniku prikazuje komunikaciju koja se odvija na CAN sabirnici.

Slika 12. Kabel koji povezuje USB-CAN pretvornik s računalom

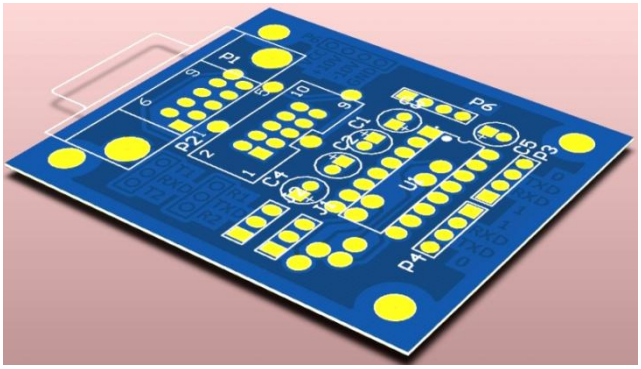
Da bi se USB-CAN pretvornik mogao koristiti na opisani način potrebno je instalirati upravljačke programe za operacijski sustav koji se izvodi na osobnom računalu. Na službenim stranicama proizvođača [6] možemo preuzeti upravljački program za Windows i Linux operacijske sustave. Na žalost, za Linux operacijske sustave nisu ponuđene gotove aplikacije koje bi upravljale USB-CAN pretvornikom preko odgovarajućih upravljačkih programa pa bi korisnik trebao napisati vlastitu aplikaciju koristeći ponuđeni API (engl. application programming interface). Iz nužne potrebe da na jednom računalu bude instaliran Linux operacijski sustav autor se odlučio koristiti drugo računalo na kojem je instalirano Windows okruženje za koje postoji gotova aplikacija koja prikazuje komunikaciju koja se odvija na CAN sabirnici. Princip korištenja aplikacije bit će prikazan u narednim poglavljima.

3.3. Serijska veza s osobnim računalom

Serijska veza s računalom na kojem je instaliran operacijski sustav Linux je potrebna kako bi mogli upravljati uCLinux operacijskim sustavom, pokretati razne aplikacije koje su napisane u njemu i još jednom provjeriti ispravnost upravljačkog programa i napisane aplikacije za CAN modul u slučaju kada USB-CAN pretvornik šalje poruku prema CAN primopredajniku razvojnog sustava. Dakle, serijska veza je potreba kako bi se ostvarila funkcija terminala koja omogućava interakciju korisnika s operacijskim sustavom (u ovom slučaju uCLinux-om). S obzirom na sve veću upotrebu prijenosnih računala koja većinom nemaju serijski priključak serijska veza je ostvarena poznatim adapterom koji ostvaruje serijsku vezu preko USB priključka koji je danas široko rasprostranjen. Adapter možemo pronaći pod nazivom HL-340 i prikazan je na sljedećoj slici.

Slika 13. HL-340 adapter

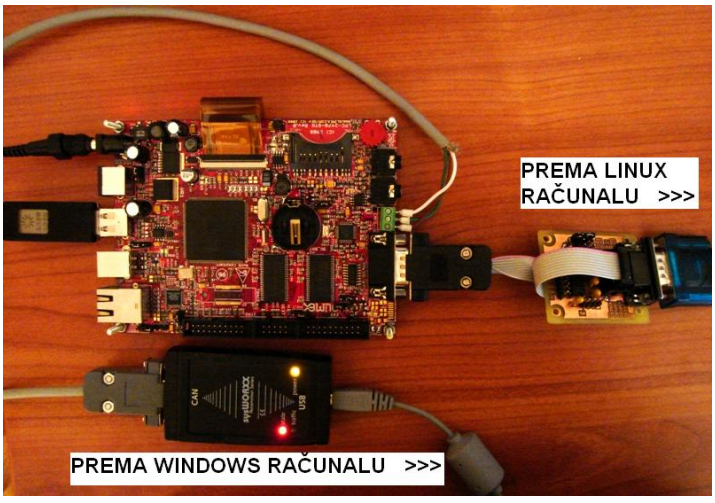
Također, korišten je pretvornik koji omogućava pretvorbu iz muškog DB-9 priključka u ženski i obrnuto. Autor pretvornika je Hrvoje Mihaldinec, bacc. elektroničkog i računalnog inženjerstva. Na sljedećoj slici možemo vidjeti 3D prikaz tiskane pločice takvog pretvornika. Način korištenja je jednostavan i detaljno je objašnjen unutar diplomskog rada samog autora. Za više informacija molimo pogledajte literaturu [2].



Slika 14. Pretvornik vrste DB-9 konektora

3.4. Način spajanja

Na sljedećoj slici možemo vidjeti kako je izgledalo radno mjesto prilikom projektiranja na opisanom razvojnom sustavu. Dakle, USB-CAN pretvornik se jednim krajem spaja na CAN priključak razvojnog sustava, a drugim krajem na računalo gdje je instaliran Windows operacijski sustav i odgovarajuća aplikacija za nadgledanje komunikacije na CAN sabirnici. Na slici vidimo USB FLASH memoriju koju koristi U-boot bootloader prilikom podizanja jezgre operacijskog sustava i datotečnog sustava[4]. Također, vidimo priključak za napajanje i opisanu serijsku vezu prema računalu na kojem je instaliran Linux operacijski sustav koji služi kao terminal za razvojni sustav.



Slika 15. Radno mjesto prilikom projektiranja na razvojnom sustavu

[4] Više o jezgri i datotečnom sustavu pročitajte u narednim poglavljima

4. Programska podrška

4.1. Razvojna okolina i alati

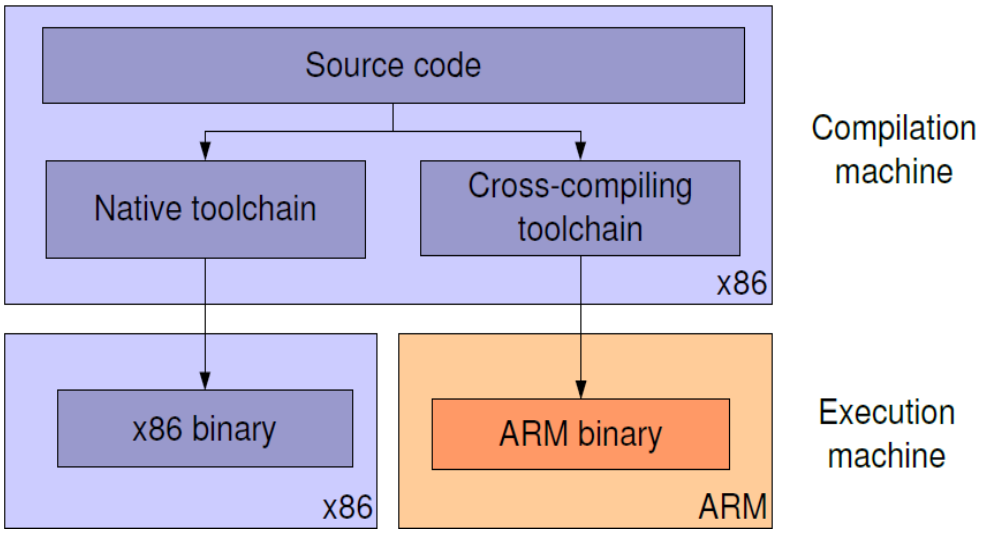
Alati za razvoj programske podrške kod ugradbenih računalnih sustava razlikuju se od ostalih alata koji se često koriste u programskom inženjerstvu po svojstvima upotrebe razvojne okoline. Kod razvoja aplikacija za ugradbene računalne sustave pronalazimo princip razvoja na više platformi – razvijamo aplikaciju pomoću razvojne okoline na osobnom računalu opće namjene, a ta aplikacija se prevodi za upotrebu na drugoj platformi, tj. ugradbenom računalnom sustavu, specifične namjene. Upravo zbog tog svojstva unutar takvih alata nalazimo kompajler kojeg nazivamo *cross*-kompajler, tj. alat koji prevodi napisanu aplikaciju na različitoj platformi u odnosu na onu za koju se prevodi (kompajlira). Među ostalim alatima nalazimo povezivač[5] (engl. *linker*), asembler, c/c++ kompajler, biblioteke i zaglavlja. Mogu se naći i dodatne biblioteke koje omogućavaju usluge kompresije, alati za pomoć pri traženju pogrešaka[6] (engl. *debugging*) i provjeru memorije. Dakle, kod ugradbenih računalnih sustava nije moguće imati jednaku platformu za kompajliranje aplikacije i za izvođenje aplikacije (slika 1).

U ovom poglavlju objasniti ćemo kako izgraditi razvojnu okolinu s pripadnim alatima na računalu opće namjene za razvoj aplikacija na ARM platformi koristeći GNU (engl. *GNU is not Unix*) kolekciju alata koji su dostupni za osobnu i komercijalnu upotrebu potpuno besplatno zahvaljujući Richardu Stallmanu - tvorcu GNU ideje, osnivaču organizacije FSF (engl. *free software foundation*), aktivisti slobodnog programa te osnivaču najpoznatije licence za slobodne programe GNU GPL koja je osnova za korištenje Linux jezgre i za većinu programa koji se koriste u svijetu slobodnih programa. GNU je trebao biti Unix-u sličan operacijski sustav koji se potpuno sastoji od slobodnih programa, a ne sadrži Unix-ov kôd[7]. Međutim, službena jezgra sustava, GNU Hurd, nije dovršena te je stoga GNU projekt postao najvažnija komponenta operativnog sustava GNU/Linux.

[5] U daljnjem tekstu koristiti će se engleski naziv *linker*

[6] U daljnjem tekstu koristiti će se engleski naziv *debugging*

[7] Odatle potječe naziv za GNU projekt – „GNU nije Unix“



Slika 16. Prikaz različitih platformi kompajliranja i izvođenja aplikacije

4.2. GNUARM

GNUARM je skup GNU prevoditelja otvorenog kôda za ARM mikrokontrolere. Ovaj skup razvojnih alata sastoji se od GNU alata za upravljanje binarnim datotekama kao što su GNU asembler i *linker*, GCC kompajlera, Newlib C biblioteke i GDB Insight grafičkog sučelja za *debugging* aplikacija. Kompajliranje i instalacija GNU razvojnih alata je složen i naporan posao koji zahtjeva dobro razumijevanje povezanosti navedenih programskih paketa i njihovih uloga u tom lancu. Postoje tri načina izgradnje GNU razvojnih alata: pribaviti izvorni kôd preko interneta i izgraditi cijeli sustav od nule uz programsko rješavanje međusobnih zavisnosti[8] pojedinih alata, pribaviti izvorni kôd preko interneta uz već razriješenu međuovisnost alata ili pribaviti prevedene binarne verzije od druge osobe ili kompanije. Potonje tipično uključuje alate spremne za upotrebu, grafičko sučelje razvojne okoline i razne dodatke na postojeću razvojnu okolinu koji omogućavaju lakši razvoj aplikacije. Također, uvelike može sačuvati vrijeme potrebno za razvoj konačnog proizvoda – aplikacije za ugradbeni računalni sustav, ali obično nisu slobodni. S druge strane, ako želimo izgraditi sustav direktno iz izvornog kôda ili barem da nastojimo razumjeti što sve obuhvaća takav proces onda ćemo ostvariti svoju nezavisnost i biti ćemo u mogućnosti da sami napravimo svoju razvojnu okolinu – potpuno besplatno.

[8] Zavisnost alata uzrokuje najviše pogrešaka prilikom prevođenja

U ovom dokumentu opisati ćemo instalaciju GNUARM razvojnih alata direktno kompajliranjem otvorenog kôda u kojem su razriješene međuovisnosti pojedinih alate, te kako instalirati takve alate. Prije nego što počnemo s prvim koracima potrebno je instalirati neku Linux distribuciju na našem osobnom računalu. Za početnike nije preporučljivo koristiti Cygwin okruženje za Windows operativne sustave jer je GNU projekt osmišljen za *Unix-like* operativne sustave poput Linux-a, stoga bi bilo sigurnije raditi u takvim okruženjima pa je preporuka da se instalira Ubuntu 9.10 distribucija koja je korištena unutar ovog rada.

Na početku ćemo definirati termine koje treba razlikovati prilikom izgradnje i korištenja spomenutih razvojnih alata:

- *build* – predstavlja platformu na kojoj kompajliramo razvojne alate
- *host* – predstavlja glavno računalo na kojoj se izvodi razvojna okolina
- *target* – predstavlja platformu za koju će se izgrađivati aplikacija

Budući da unutar korištene Linux distribucije nisu instalirani određeni programski paketi koje koriste pojedini GNU alati morati ćemo to napraviti prije izgradnje samog razvojnog sustava, inače ćemo se čuditi brojnim greškama koje će nastati prilikom instalacije. U sljedećoj tablici prikazani su neophodni programski paketi potrebni prije početka instalacije GNU alata.

Tablica 3. Neophodni paketi prije instalacije GNU alata

libgmp3-dev	http://packages.ubuntu.com/dapper/libgmp3dev
libmpfr-dev	http://packages.ubuntu.com/hardy/libmpfrdev
texinfo[9]	http://packages.ubuntu.com/hardy/texinfo
patch	http://packages.ubuntu.com/hardy/patch
libncurses5-dev	http://packages.ubuntu.com/dapper/libncurses5dev
libx11-dev	http://packages.ubuntu.com/dapper/libx11dev
termcap	ftp://ftp.gnu.org/gnu/termcap/

Kako bi bili u mogućnosti instalirali prethodne pakete potrebna su administratorska prava na instaliranoj Linux platformi. Početnik se može naći u problemu da ne zna administratorsku zaporku, stoga je poželjno promijeniti administratorsku zaporku sljedećim nizom naredbi koje treba unijeti unutar terminala:

```
sudo passwd root

> korisnička zaporka

> željena root-zaporka
```

Nakon promjene *root*-zaporke možemo se prijaviti kao administrator i izvršiti instalaciju traženih paketa sljedećim nizom naredbi:

```
su

> root-zaporka

apt-get install libgmp3-dev libmpfr-dev

apt-get install patch

apt-get install libncurses5-dev libx11-dev

apt-get install texinfo
```

[9] Službeni dokumentacijski format GNU projekta

Izgradnja razvojnih alata uključuje sljedeće korake:

1. Dekompresija pribavljenih paketa GNU alata
2. Konfiguracija za razvoj na ARM platformi
3. Kompajliranje pribavljenih GNU paketa
4. Instalacija paketa

Navedeni koraci su ostvareni u *shell* skripti koja je navedena u dodatku A ovoga rada[10]. Najvažniji parametar konfiguracije prilikom izgradnje razvojnih alata je TARGET-NAME koji određuje vrstu kompajlera i ekstenziju datoteke koju kompajler generira.

U nastavku ćemo ukratko objasniti neke od osnovnih mogućnosti:

arm-linux – ovo je najčešći odabir kod izgradnje ugradbenih aplikacija jer podržava ELF[11] format za standard ARMLinux.

arm-linuxaout – ova mogućnost generira zastarjeli 'a.out' format za standard ARMLinux koji se sve manje koristi i zamjenjuje ga ELF format binarnih datoteka.

arm-aout, *arm-coff*, *arm-elf*[12], *arm-thumb* – ove mogućnosti generiraju samostalne ELF binarne datoteke, tj. one koje nisu vezane za neki operativni sustav.

Ako razvojne alate želimo razviti za posebnu ARM arhitekturu onda možemo umjesto *arm* prefiksa staviti odgovarajući prefiks arhitekture za koju želimo razviti alate:

armv2, *armv3l*, *armv3b*, *armv4l*, *armv4b*, *armv5l*, *armv5b*.

Unutar nekog odabranog direktorija potrebno je napraviti tri mape – *install*, *src*, *build*. Unutar istog direktorija presnimiti datoteku s nastavkom ***.sh** čiji je sadržaj skripta koju treba pokrenuti kako bi se instalirali razvojni alati GNUARM.

10 Ukoliko čitatelj ne razumije *shell* naredbe upućujemo ga na literaturu [13]

11 Najpoznatiji format za binarne datoteke koje nastaju nakon prevođenja

12 Skripta instalira GNUARM alate s parametrom *arm-elf*

Preuzeti arhive navedenih GNUARM alata s navedenih URL-ova i spremiti ih u *src* mapu:

Tablica 4. Popis GNUARM alata potrebnih prilikom razvoja aplikacija

binutils-2.19	http://gnuarm.org/binutils-2.19.tar.bz2
gcc-4.3.2.tar	http://gnuarm.org/gcc-4.3.2.tar.bz2
newlib-1.16.0	http://gnuarm.org/newlib-1.16.0.tar.gz
insight-6.8-1	http://ftp.twaren.net/Unix/Sourceware/insight/releases/

Navedena skripta će dekomprimirati arhive i alati će biti instalirani unutar *install* mape. Recimo da se datoteka u kojoj se nalazi skripta zove *gnuarm-install.sh*, pozicioniramo se u glavni direktorij u kojem se nalazi skripta i unutar terminala unesemo sljedeću naredbu:

```
./gnu-arm-installer.sh > output.txt 2> error.txt
```

Oznakom '>' označava se preusmjeravanje standardnog izlaza u *output.txt* datoteku koju ne treba prethodno stvarati, a oznakom '2>' označavamo preusmjeravanje toka standardne pogreške u datoteku *error.txt*. Preusmjeravanje omogućava da nakon (ne)uspješne instalacije vidimo u čemu je mogao biti problem prilikom instalacije. Nakon 40-ak minuta trebali bi imati instalirane GNUARM alate unutar *install* mape. Kako bi provjerili da li je instalacija prošla onako kako smo očekivali trebamo se pozicionirati unutar */install/bin* mape i pogledati što sadrži:

```
/direktorij/install/bin# ls arm*
```

Nakon toga bi trebali dobiti sljedeći niz instaliranih alata:

arm-elf-addr2line	*arm-elf-gcc*	*arm-elf-insight*	*arm-elf-run*
arm-elf-ar	*arm-elf-gcc-4.3.2*	*arm-elf-ld*	*arm-elf-size*
arm-elf-as	*arm-elf-gccbug*	*arm-elf-nm*	*arm-elf-strings*
arm-elf-c++	*arm-elf-gcov*	*arm-elf-objcopy*	*arm-elf-strip*

arm-elf-c++filt	*arm-elf-gdb*	*arm-elf-objdump*
arm-elf-cpp	*arm-elf-gdbtui*	*arm-elf-ranlib*
arm-elf-g++	*arm-elf-gprof*	*arm-elf-readelf*

Kako bi omogućili da instalirane alate uopće možemo pozvati moramo taj direktorij dodati unutar PATH varijable. Unesimo sljedeći niz naredbi u terminal:

```
nano ~/.bashrc
```

Otvoriti će se tekst uređivač unutar terminala sa sadržajem datoteke *.bashrc*. Na kraju te datoteke potrebno je dodati sljedeći redak[13]:

```
export PATH=$PATH:/direktorij/install/bin
```

Izaći iz uređivača teksta komandom **Ctrl+X** i sačuvati promjene. Da bi promjene bile odmah vidljive potrebno je unutar terminala upisati:

```
source ~/.bashrc.
```

S druge strane, postoje već prevedene binarne verzije takvih alata od raznih kompanija ili osoba. Sa službene stranice GNUARM projekta možemo preuzeti različite verzije već prevedenih alata. Odaberimo neku verziju i spremimo ju u korisnički *home* direktorij i dekomprimiramo ju. Sve što je potrebno napraviti je dodati sljedeći redak unutar *~/.bashrc* datoteke:

```
export PATH=$PATH:/putanja/do/gnuarm-x.x.x/bin
```

Ukoliko takva prevedena binarna verzija nije prilagođena računalu na koji se instalira onda bi bilo poželjno preuzeti odgovarajuće izvorne kôdove preko GNUARM web stranice za koje su riješene međuovisnosti različitih alata i onda ponoviti postupak koji je opisan gore koristeći napisanu skriptu. Ako želimo prevesti i instalirati alate od nule onda obično unutar datoteke koja opisuje pogreške možemo vidjeti veliku količinu upozorenja koja potencijalno mogu predstavljati probleme pa je stoga neophodno ispraviti takva upozorenja koja su nastala prilikom instalacije GNUARM alata iz izvornog kôda. Navesti ćemo najvažnije vrste upozorenja: ignoriranje povratnih vrijednosti funkcija za koje to nije dozvoljeno, nedefinirani prototipovi i nekorištene funkcije. Ova upozorenja mogu

[13]Poželjno je da budemo prijavljeni kao obični korisnici

se riješiti modifikacijom izvornog kôda pri čemu je nužno dobro poznavanje povezanosti koda svih instaliranih alata. Također, unutar ovoga rada korišteni su GNU alati koji su preporučeni od kompanije Olimex.

Sve dodatne informacije o instalaciji navedenih alata čitatelj može pronaći na službenoj stranici [7]. Ukoliko ima problema i ne može naći rješenje službenim putem onda predlažemo *yahoo-gnuarm* grupu [8], gdje može saznati neophodne informacije o svemu što smo naveli, a i mnogo više. Također, na internetu može pronaći projekt koji nastoji prikazati koje se verzije gore spomenutih GNU alata međusobno slažu [9].

4.3. Gedit

Gedit je službeni editor teksta za GNOME Linux grafička okruženja koji je jedan od preporučenih za razvoj aplikacija. Jednostavan je i lako se koristi, a istovremeno predstavlja moćan razvojni alat koji nudi sljedeće funkcionalnosti:

- Podrška za internacionalni tekst (UTF-8)
- Podesivo sintaksno naglašavanje ključnih riječi za brojne programske jezike (C, C++, Java, HTML, XML, Python, Perl, itd.)
- Editiranje datoteka sa udaljenih lokacija
- Periodičko spremanje otvorenih datoteka
- Automatsko formatiranje kôda
- Prikazivanje broja linije kôda
- Predefinirani odlomci kôda i njihovo formatiranje[14]
- Upravitelj vanjskim alatima[15]
- Veliki izbor dodataka

Ovakav editor nudi veliki izbor dodataka koji iznimno olakšavaju programiranje. Od brojnih dodataka izdvojiti ćemo samo neke[16]:

- *Bracket Completion*: Automatsko dodavanje završne zagrade koja označava kraj bloka kôda
- *Change Case*: Promjena veliko-malo i malo-veliko slovo unutar označenog teksta
- *Code Comment*: Komentiranje ili odkomentiranje označenog kôda

[14] Code Snippets

[15] External Tools Manager (npr. Build, Open terminal here, Run command ...)

[16] Navedeni dodaci se mogu naći na [10]

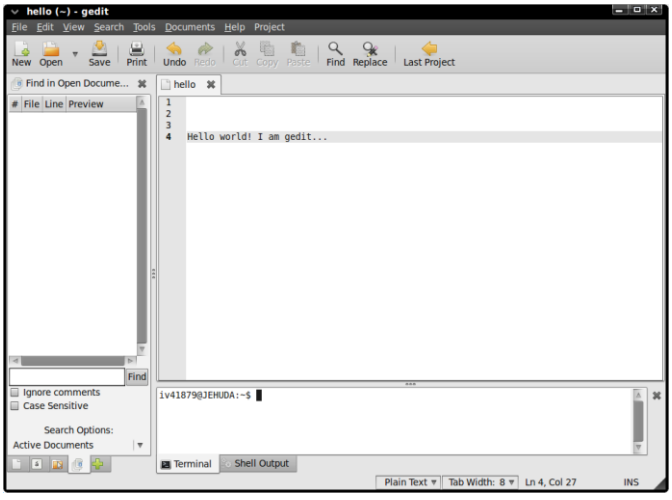
- *Color Picker*: Automatsko ubacivanje heksadecimalne vrijednosti odabrane boje unutar kôda
- *Embedded Terminal*: Ugrađeni terminal
- *External Tools*: Izvršavanje vanjskih naredbi i shell skripti
- *Find In Documents*: Pretraživanje svih otvorenih datoteka ili datoteka unutar aktivnog direktorija
- *Insert Date/Time*: Umetanje trenutnog vremena i datuma na mjesto kursora unutar teksta
- *Project Manager*: Omogućava grupiranje datoteka u jedan projekt

Jedan veći dio navedenih alata može se instalirati upisivanjem sljedeće naredbe unutar terminala:

```
sudo apt-get install gedit-plugins
```

Dodaci koji nisu unutar standardnog paketa Ubuntu distribucije dodaju se kopiranjem preuzetih datoteka sa GNOME web stranice [10] u *.gnome2/gedit/plugins/* ili */usr/lib/gedit-2/plugins/* direktorij, ovisno o tome da li želimo da instalacija vrijedi za sve korisničke račune instalirane na određenoj Linux distribuciji.

Ukoliko čitatelj ima veće zahtjeve onda preporučamo *Eclipse* razvojno okruženje. Više o instaliranju i podešavanju *Eclipse-a* možete pročitati u diplomskom radu bacc. Hrvoja Mihaldinca [2].



Slika 17. Editor teksta – gedit

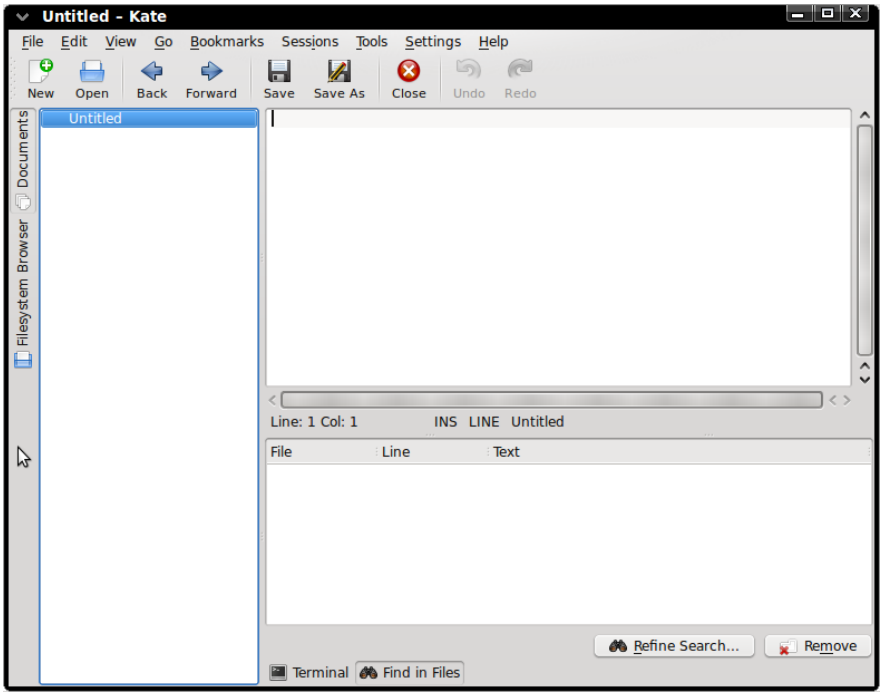
4.4. Kate

Kate je službeni editor teksta za KDE Linux grafička okruženja i također je preporučeni tekst editor za razvoj raznih aplikacija. Podržava gotovo identičnu funkcionalnost koja je navedena i za gedit tekst editor uz mogućnost nadogradnje koristeći dodatke (engl. *plugins*). Sintaksno naglašavanje ključnih riječi koje je korišteno u ovome radu je kopirano iz ovog tekst editora zahvaljujući mogućnosti eksportiranja kôda u HTML format[17] (*File->Export as HTML…*) prilikom čega se sačuva sintaksa jezika u kojem je kôd napisan i nakon kopiranja u neki *word* procesor (Word, OpenOffice, itd.).

Instalacija Kate editora se može provesti sljedećom naredbom unutar terminala instalirane Linux distribucije:

```
sudo apt-get install kate
```

Nakon instalacije i pokretanja ovog editora dobiti ćete grafički prikaz koji je sličan prikazu na sljedećoj slici.



Slika 18. Editor teksta - kate

[17] Službena verzija gedit-a nema podržanu tu funkciju

5. Bootloader

5.1. Uvod

Bootloader[18] je program koji se izvodi odmah nakon priključenja napajanja na ugradbeni računalni sustav. Odgovoran je za inicijalizaciju sklopovlja[19] nužnog za izvođenje bilo kakve aplikacije, inicijalizaciju i testiranje sklopovlja koje koristi prilikom svoga izvođenja, premještanje vlastitog kôda iz FLASH memorije u radnu RAM memoriju, inicijalizaciju sklopovlja koje ostvaruje korisničko sučelje prema PC-u, inicijalizaciju sklopovlja koje je odgovorno za preuzimanje[20] jezgre operacijskog i datotečnog sustava[21] u radnu RAM memoriju, pozivanje jezgre operacijskog sustava s odgovarajućim ulaznim argumentima i predaja kontrole toka izvođenja jezgri (slika 1).

Prilikom odabira bootloader-a vodimo se sljedećim kriterijima:

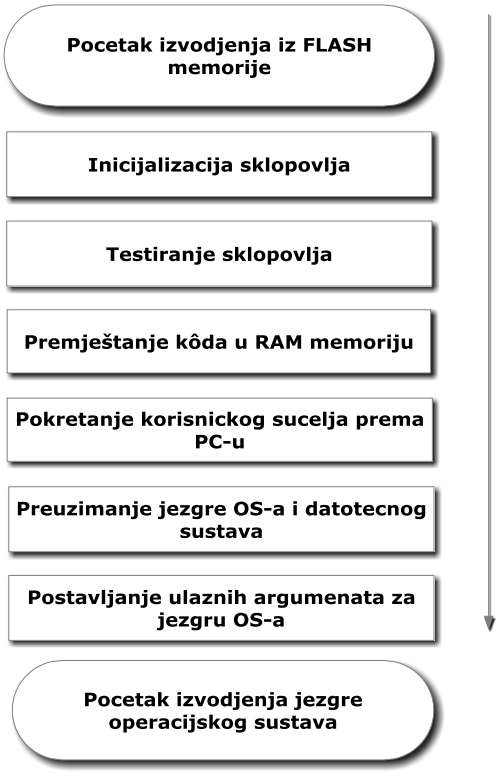
- Podržane platforme i procesorske arhitekture
- Podržani operacijski sustavi
- Mogućnost *debugging*-a
- Stabilnost, portabilnost i konfigurabilnost
- Korisničko sučelje
- Naredbeno sučelje
- Mogućnost komprimiranja u svrhu oslobođenja memorije
- Podržani protokoli preuzimanja jezgre OS-a i datotečnog sustava sa odgovarajućeg medija
- Mogućnost prenošenja argumenata prema jezgri OS-a
- itd.

[18] Program punilac

[19] Procesor, memorijski modul

[20] Flash, USB, Ethernet (TFTP), UART, IDE ...

[21] Cramfs, ext2, FAT ...



Slika 19. Tok izvođenja bootloader-a

Unutar ovog rada naglasak će biti na bootloader-ima koji su podržani od operacijskog sustava koji se bazira na linux jezgri. Od tih bootloader-a navesti ćemo one koji podržavaju specifičnu arhitekturu procesora kao i univerzalne bootloader-e. Poseban naglasak ćemo staviti na grupu univerzalnih bootloader-a gdje ćemo detaljnije opisati jednog od najpoznatijih bootloader-a pod nazivom U-boot (U=Universal).

5.2. U-boot

U-boot[22] je bootloader otvorenog kôda[23] kojeg razvija i održava Wolfgang Denk[24] kao nadogradnju na ranije PPCBoot i ARMBoot projekte. Koncept U-boot-a je zasnovan na Linux-ovoj jezgri[25] operacijskog sustava te se prilikom projektiranja nastoji razmišljati na principu *prenosivog sučelja operacijskog sustava*. Podržava mnoge procesorske arhitekture među kojima su ARM, AVR32, Blackfin, x86,

[22] Universal Bootloader

[23] http://www.denx.de/wiki/U-Boot/

[24] DENX-ov programski inženjer

[25] Nekada i cijeli komadi kôda

Motorola 68K, Xilinx Microblaze, MIPS, Alterra NIOS, NIOS2, PowerPC i ostali. Među podržanim operacijskim sustavima izdvajamo (uC)Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS i LynxOS. Omogućava visoku kofigurabilnost pomoću sistemskih varijabli koje mogu sačuvati svoje stanje, pruža korisničko sučelje preko širokog skupa naredaba, podržava preuzimanje komprimirane jezgre OS-a i datotečnog sustava preko ethernet-a (BOOTP/DHCP, TFTP i NFS), USB FLASH memorije, SD/MMC kartice, IDE/SCSI HDD i konzole, omogućava sažimanje i pakiranje koristeći gzip i bzip2 alate. Podržava širok spektar datotečnih sustava – Cramfs (Linux), ext2 (Linux), FAT (Microsoft) i JFFS2 (Linux) i jedna od vodećih prednosti pred ostalim bootloader-ima je sadržajna dokumentacija koja se može pronaći na službenim web stranicama. Unatoč brojnim prednostima U-boot ima svoje nedostatke koji se ogledaju u potrebi dubokog poznavanja arhitekture procesora i platforme na kojoj se nalazi; model U-boot-a se ne zasniva na čistom modelu upravljačkog programa (engl. *device driver model*) kao što to možemo susresti kod jezgri brojnih operacijskih sustava (potrebno više apstrakcije umjesto direktnog pristupa memoriji); ručna konfiguracija U-boot-a preko korisničkih konstanti unutar zaglavlja još je jedna negativna strana koja se nastoji izbjeći u novijim verzijama. Također, mnogi upravljački programi napisani za U-boot imaju gotovo jednak API (engl. *application programming interface*) pa se u novijim verzijama radi na izgradnji odgovarajućih framework-a, ali glavna mjera koja postavlja granicu u dodatnim mogućnostima je veličina kôda – to je još uvijek glavni zahtjev prilikom izgradnje U-boot-a jer se nastoji zadržati dovoljno kompaktan kôd. U nastavku navodimo pregled bootloader-a korištenih od operacijskih sustava zasnovanih na jezgri Linux.

Univerzalni bootloader-i:

- U-boot
- RedBoot
- Smart Firmware
- MicroMonitor

Bootloader-i za X86 arhitekturu:

- Lilo
- GRUB
- Etherboot
- LinuxBIOS

- ROLO
- SYSLINUX
- NILO

Bootloader-i za ARM arhitekturu:

- Blob
- IPAQ

Bootloader-i za PPC arhitekturu:

- PPCBoot
- Yaboot

Bootloader-i za MIPS arhitekturu:

- PMON

Bootloader-i za Sun arhitekturu:

- Sun3 Bootloader

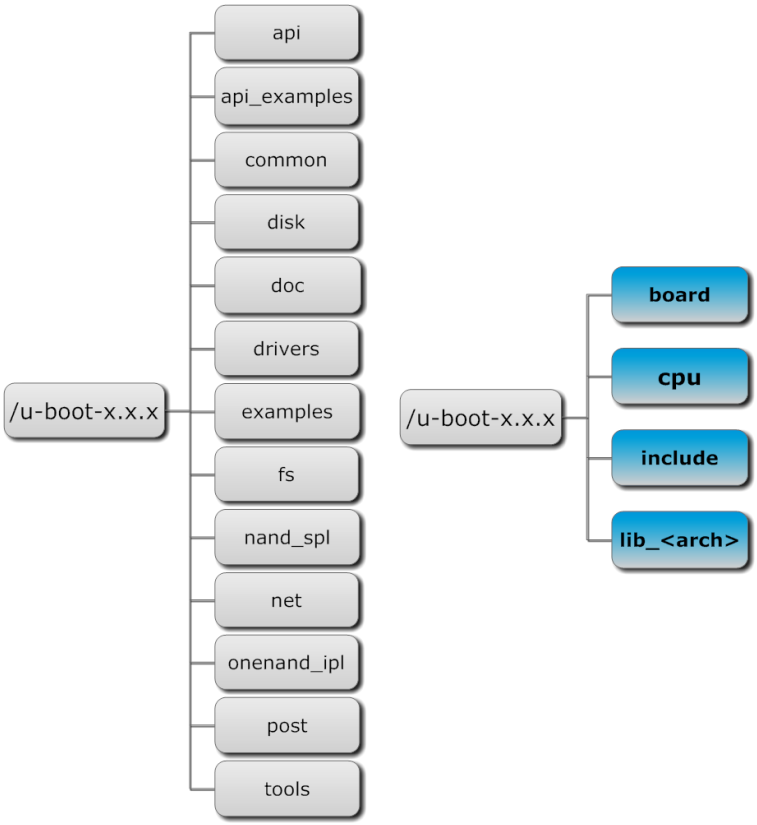
Bootloader-i za Super-H arhitekturu:

- Sh-ipl+g

5.2.1. Struktura direktorija U-boot-a

Nakon što smo preuzeli i dekomprimirali izvorni kôd U-boot bootloader-a sa službenih web stranica možemo istraživati strukturu direktorija koja je prikazana na slici 2. Postoje dvije vrste mapa unutar U-boot direktorija - mape ovisne o platformi za koju se U-boot konfigurira i one koje to nisu već spadaju u grupu općih ili zajedničkih mapa. U prvu grupu mapa spadaju: **board, cpu, include** i **lib_<arch>**[26] mape i na slici 2 su posebno izdvojene, a u drugu grupu spadaju sve ostale. Mape koje su ovisne o platformi moraju sadržavati kôd koji implementira sučelje prema U-boot-u u svrhu ostvarivanja odabrane funkcionalnosti, npr. ako platforma (razvojni sustav) ima podržanu komunikaciju preko serijskog protokola onda postoje točno određene funkcije koje moraju biti implementirane od strane korisnika kako bi ostatak U-boot kôda mogao omogućiti takvu funkcionalnost. Za čitatelje koji su upoznati s objektno orijentiranim programiranjem razumjeti će koncept implementacije sučelja koje pruža odgovarajući objekt.

[26] arch – označava arhitekturu za koju se U-boot konfigurira



Slika 20. Struktura direktorija

Na prikazanoj slici oznaka x.x.x označuje verziju korištenog U-boot-a. Verzija koja će biti korištena kako bi se detaljnije opisalo korištenje U-boot-a je 1.3.2. Razvojni sustav za koji se konfigurira U-boot je Olimex-ov LPC2478STK[27]. U sljedećoj tablici možemo vidjeti opis najvažnijih mapa navedenih na gornjoj slici gdje u opisu onih mapa koje su ovisne o platformi navodimo datoteke koji su od važnosti za navedeni Olimex-ov razvojni sustav.

[27] http://www.olimex.com/dev/index.html

Tablica 5. Opis najvažnijih direktorija U-boot-a

Naziv mape	*Opis*
board/<boardname>	Sadrži mape koje su specifične za svaku platformu. U njima se nalaze lokalno korištena zaglavlja (*.h) *.c datoteka koje sadrže razne inicijalizacijske funkcije. Te funkcije se pozivaju iz *lib_<arch>/board.c*, *cpu/arm720t/start.S*, *net/net.c*, *common/cmd_*.c*, *drivers/** i *disk/part.c* datoteka te uz ostale upravljače funkcije predstavljaju dio U-boot sučelja. Također, unutar tih *.c datoteka nalazimo lokalne prekidne funkcije, razne makroe i ostalo.
common	Tu se nalaze naredbe koje podržava U-boot (npr. cmd_boot.c, cmd_date.c, command.c ...), implementacija CRC16 zaštite (*crc16.c*), implementacija malloc funkcija (*dlmalloc.c*), funkcije zadužene za upravljanje sistemskom okolinom U-boot-a (*env_**), podrška za FLASH (*flash.c*) glavni program *main.c* u kojem sa nalazi jedna od važnijih funkcija U-boot-a *void main_loop (void)* itd.
cpu/arm720t	Specifični kôd za pojedini procesor koji sadrži inicijalizaciju procesora, stôga, prekida, globalnih prekidnih funkcija (*cpu.c*, *interrupts.c*), implementaciju serijskog protokola (*serial.c*), *startup* kôd za odgovarajuću arhitekturu (*start.S*). Upravo *startup* kôd predstavlja početnu točku iz koje se počinje izvoditi bootloader U-boot.
drivers	Razni upravljački programi (serial, usb, video, i2c, mtd ...)

include	Unutar ove mape nalazimo razna zaglavlja (*header* datoteke) u kojima su različiti prototipovi funkcija, deklaracije i konfiguracijske konstante. Jedan od najvažnijih zaglavlja je *common.h* koji predstavlja jezgru U-boot definicija, deklaracija i prototipova. Za odgovarajuću platformu stvaramo mapu *asm-arm/arch-lpc2468* u kojoj se nalaze globalna zaglavlja vezana upravo za dotični procesor (*lpc2468_registers.h*). Najvažnija datoteka koja predstavlja jezgru konfiguracije U-boot-a smještena je unutar *configs* podmape pod nazivom <*boardname*>*.h*. Unutar tog zaglavlja nalazi se veliki broj definiranih konstanti koje omogućavaju ili onemogućavaju neke funkcije U-boot-a. Prije pisanja toga zaglavlja preporuča se pročitati README datoteku unutar *root* direktorija U-boot-a.
lib_*	Sadrži specifične biblioteke procesora i osnovne funkcije iz kojih se inicijalizira U-boot koje se većinom nalaze unutar *board.c*. Jedna od osnovnih funkcija je *void start_armboot (void)* koja se poziva odmah nakon *startup* kôda *start.S*. Unutar te datoteke definirana je sekvenca inicijalizacije U-boot-a kao polje pokazivača na funkcije: *init_fnc_t *init_sequence[] = {cpu_init, board_init, ...}*
net	Sadrži datoteke kôda koji je vezan za ethernet komunikaciju (eth.c, net.ch, bootp.c ...). Obično se ta komunikacija ostvaruje kako bi se preuzela jezgra operacijskog sustava Linux i datotečni sustav sa poslužitelja.

doc	Kratka dokumentacija u kojoj je veliki broj README datoteka.
examples	Primjeri raznih aplikacija (hello_world.c ...) za U-boot.
fs	Direktorij koji sadrži kôd za datotečne sustave (FAT, JFFS2, Cramfs, FDOS...).
tools	Razni alati i datoteke (env, gdb, logos, mkimage.c).
post[28]	Sadrži kôd za testiranje (post.c, codec.c, cache.c...).

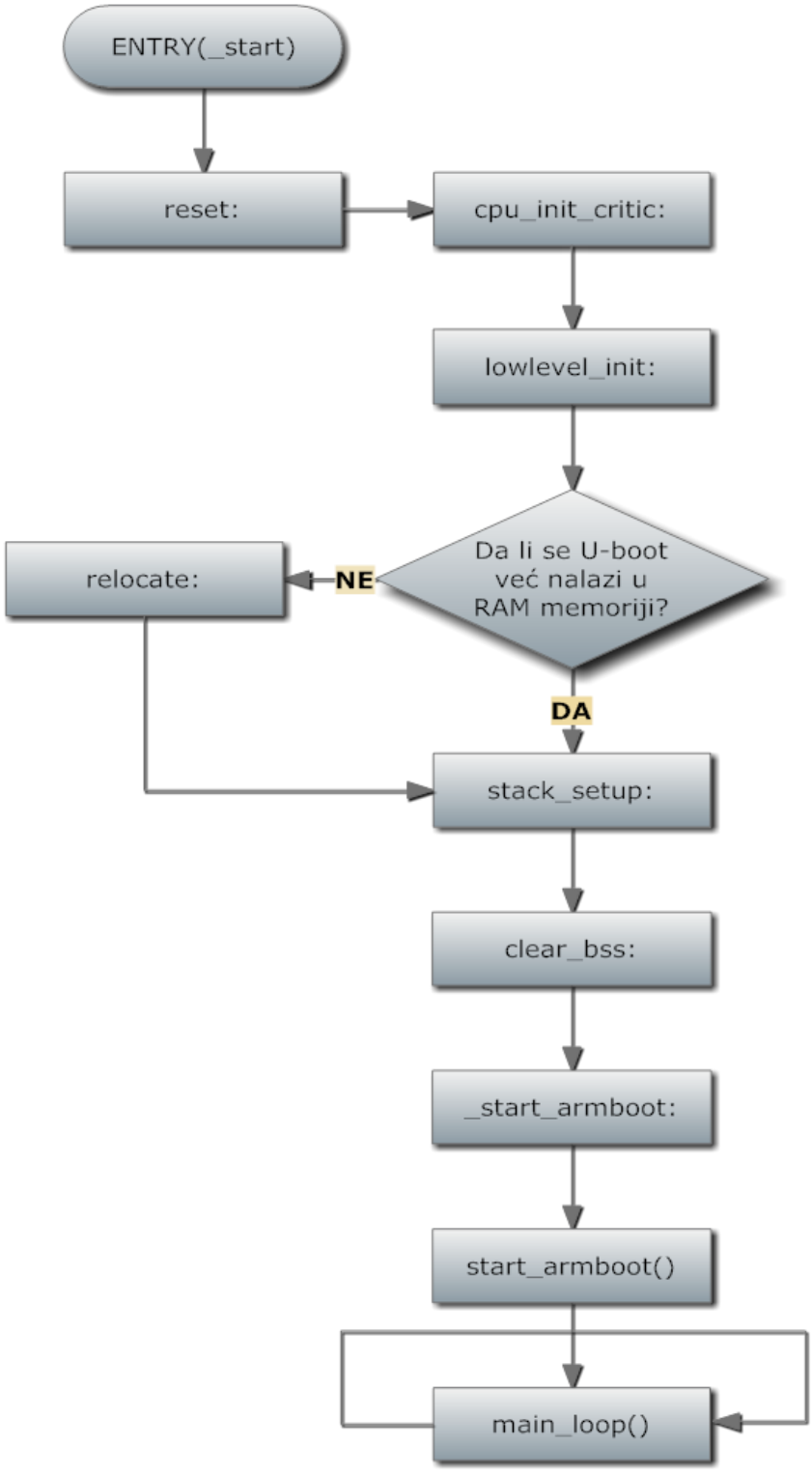
[28] engl. *Power-On Self Test*

5.2.2. Tok izvođenja

Tok izvođenja U-boot-a prikazan je na slici 21. Kako bi se odredilo gdje počinje izvođenje kôda pogledajmo unutar skriptne datoteke GNU *linker*-a (*board/lpc_2478_stk/u-boot.lds*) kako bi saznali koja je ulazna točka za U-boot. Dan je sljedeći isječak kôda iz navedene datoteke:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;

        . = ALIGN(4);
        .text      :
        {
          cpu/arm720t/start.o  (.text)
          *(.text)
        }

        . = ALIGN(4);
        .rodata : { *(.rodata) }

        . = ALIGN(4);
        .data : { *(.data) }

        . = ALIGN(4);
        .got : { *(.got) }

        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;

        . = ALIGN(4);
        __bss_start = .;
        .bss : { *(.bss) }
        _end = .;
}
```

Naredba ENTRY predstavlja jedan od nekoliko načina određivanja ulazne točke programa. U ovom slučaju radi se o labeli *_start* koja je definirana unutar *startup* datoteke *cpu/arm720t/start.S*. Dakle, najprije se počinje izvoditi *startup* kôd jezgre procesora gdje na mjestu *_start* labele odmah dolazi naredba za grananje na *reset* dio *startup* kôda. Jedini posao unutar *reset* kôda je ulazak u *supervisor* (SVC) način rada procesora u kojem je omogućen veći skup naredbi i operacija koje se inače ne mogu koristiti u korisničkom načinu rada (User).



Slika 21. Tok izvođenja do glavne petlje U-boot-a

Nakon *_reset* dijela pozivamo funkciju c*pu_init_critic* u kojoj samo pripremamo privremeni stog veličine 0x2000 na adresi 0x40008000 za sljedeću veoma važnu funkciju – *lowlevel_init* koja se nalazi unutar */board/lpc_2478_stk/lowlevel_init.c*. Funkcije unutar te datoteke su lokalne i koriste se unutar funkcije *void lowlevel_init(void)* koja jedina predstavlja U-boot sučelje prema *startup* kôdu iz kojeg se poziva. Unutar te funkcije inicijalizira se PLL, pokreću se sistemski vremenski sklopovi (engl. *timer*), postavlja se modul za ubrzavanje memorije, inicijaliziraju se kontroleri ulaza i izlaza (GPIO), kontroler prekidnih vektora (VIC), dva modula za serijsku komunikaciju (UART0 i UART1), sučelje prema vanjskoj memoriji (EMC) te se omogućuje LCD kontroler. Navedeno možemo pogledati na sljedećim isječcima kôda. Korisno je primijetiti da se U-boot podrška za novu platformu projektira pomoću blokova kôda ograđenih pretprocesorskim direktivama (#if, #elif, #else ...) kako se ne bi narušio kôd za već podržane platforme.

```
.globl _start
_start: b       reset   /* grananje na RESET */
        ldr     pc, _undefined_instruction
        ldr     pc, _software_interrupt
        ldr     pc, _prefetch_abort
        ldr     pc, _data_abort
        /* ... */

reset:
        /*
         * SVC32 nacin rada
         */
        mrs     r0,cpsr
        bic     r0,r0,#0x1f
        orr     r0,r0,#0x13
        msr     cpsr,r0

        bl      cpu_init_crit

        bl      lowlevel_init

cpu_init_crit:
        /* ... */
#elif defined(CONFIG_LPC2468)
        ldr     r0, =0x40008000
        mov     sp, r0
        sub sl, sp, #0x2000  /* SL = Stack Limit (R10) */
        /* ... */
```

Nakon inicijalizacije slijedi premještanje U-boot kôda iz FLASH memorije u RAM memoriju u svrhu bržeg izvođenja. Sljedeći odsječak kôda provjerava da li je U-boot već premješten u RAM memoriju. Labela _TEXT_BASE definirana je unutar *board/lpc_2478_stk/config.mk* i predstavlja adresu na koju će se premjestiti U-boot kôd. Obično se prilikom *debugging*-a U-boot odmah napuni u RAM memoriju pa provjera ima smisla. S druge strane, unutar *include/config/lpc_2478_stk.h* može se dodati *#define CONFIG_SKIP_RELOCATE_UBOOT* pa se *relocate* dio *startup* kôda neće niti prevesti prilikom poziva GNU alata. U slučaju da se U-boot želi odmah napuniti u RAM memoriju potrebno je modificirati početnu adresu punjenja u *linker* skripti *board/lpc_2478_stk/u-boot.lds* tako što ćemo umjesto 0x00000000 napisati odgovarajuću adresu unutar RAM memorije[29].

```
relocate:   /* r0 <- trenutna pozicija U-boot-a (FLASH ili RAM) */
        adr     r0, _start
        ldr     r1, _TEXT_BASE
        cmp     r0, r1
        beq     stack_setup     /* Ako se U-boot vec izvodi iz RAM
memorije idi odmah na postavljanje stoga */
```

Premještanje kôda je jednostavno kopiranje cijelog *armboot* dijela (od *_start* labele pa sve do bss[30] segmenta) od izvorišne adrese unutar FLASH memorije prema odredišnoj adresi unutar RAM memorije.

```
/* premjestanje U-boot-a u RAM memoriju */
        ldr     r2, _armboot_start /* r2 = _armboot_start = _start */
        ldr     r3, _bss_start     /* r3 = _bss_start */
        sub     r2, r3, r2         /* r2 <- velicina armboot-a */
        add     r2, r0, r2         /* r2 <- zavrsna adresa */
                /* r0 <- trenutna pozicija U-boot-a (FLASH) */
copy_loop:
        ldmia   r0!, {r3-r10}       /* r0 = izvorisna adresa koda */
        stmia   r1!, {r3-r10}      /* r1 = adresa odredista koda */
        cmp     r0, r2             /* ponavljaj sve do zavrsne adrese */
        ble     copy_loop
```

Čitatelj se možda zapitao gdje je definiran početak bss segmenta (*_bss_start*). Definiran je unutar *linker* skripte *board/lpv_2478_stk/u-boot.lds* nakon što *linker* napuni memoriju s prethodno definiranim segmentima kôda.

---

[29] Također, nakon što smo povezali odgovarajući *debuger* (mora podržavati GDB server, npr. Jlink i ostali), pokrenuli GDB server i grafičko sučelje GDB Insight onda prilikom *debugging*-a moramo modificirati vrijednost programskog brojila na odgovarajuću vrijednost početne RAM lokacije.

[30] Opis tog segmenta je u nastavku

Sljedeći korak je konfiguracija stôga kako je prikazano sljedećim odsječkom kôda. Kôd prikazuje jednostavno oduzimanje vrijednosti pokazivača (registar) r0 od vrijednosti početne adrese unutar RAM memorije prilikom rezervacije memorijskog prostora za različita područja koja se koriste unutar U-boot-a (malloc, globalne varijable, prekidi i abort-stôg). Konačna vrijednost pokazivača r0 se pridruži registru SP[31]. Gore opisano možemo prikazati u matematičkom obliku:

SP = _TEXT_BASE – glb_data – heap – irq_stack – fiq_stack – abrt_stack.

```
stack_setup:
        ldr     r0, _TEXT_BASE          /* gornjih 128 KB: premjesteni U-
boot */
        sub     r0, r0, #CFG_MALLOC_LEN /* malloc prostor */
        sub     r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo = board info */
#ifdef CONFIG_USE_IRQ
        sub     r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
        sub     sp, r0, #12             /* abort-stack = 3 rijeci */

clear_bss:
        ldr     r0, _bss_start          /* pocetak bss segmenta */
        ldr     r1, _bss_end            /* kraj bss segmenta */
        mov     r2, #0x00000000         /* upisivanje 0 na bss segment */

clbss_l:str     r2, [r0]                /* ocisti trenutnu lokaciju */
        add     r0, r0, #4              /* pomaknu se na sljedecu rijec */
        cmp     r0, r1                  /* da li je kraj bss segmenta? */
        ble     clbss_l

        ldr     pc, _start_armboot      /* lib_arm/board.c */

_start_armboot: .word start_armboot
```

Unutar prethodnog odsječka možemo primijetiti inicijalizaciju bss segmenta kôda kojeg koriste mnogi kompajleri i *linkeri* kao poseban dio podatkovnog segmenta unutar kojeg nalazimo statički alocirane varijable čija je inicijalna vrijednost jednaka 0.

[31] engl. *Stack Pointer*

Konačno, predzadnja linija kôda prikazuje bezuvjetni skok na labelu *_start_armboot* koja u sebi sadrži adresu funkcije *void start_armboot (void)* koja se nalazi unutar *lib_arm/board.c* datoteke, kako prikazuje zadnja linija kôda unutar prethodnog odsječka. To je još jedna od važnih funkcija prilikom pokretanja U-boot-a koja obavlja inicijalizaciju ostalog sklopovlja. Sljedeći odsječak kôda je posebice važan unutar spomenute datoteke:

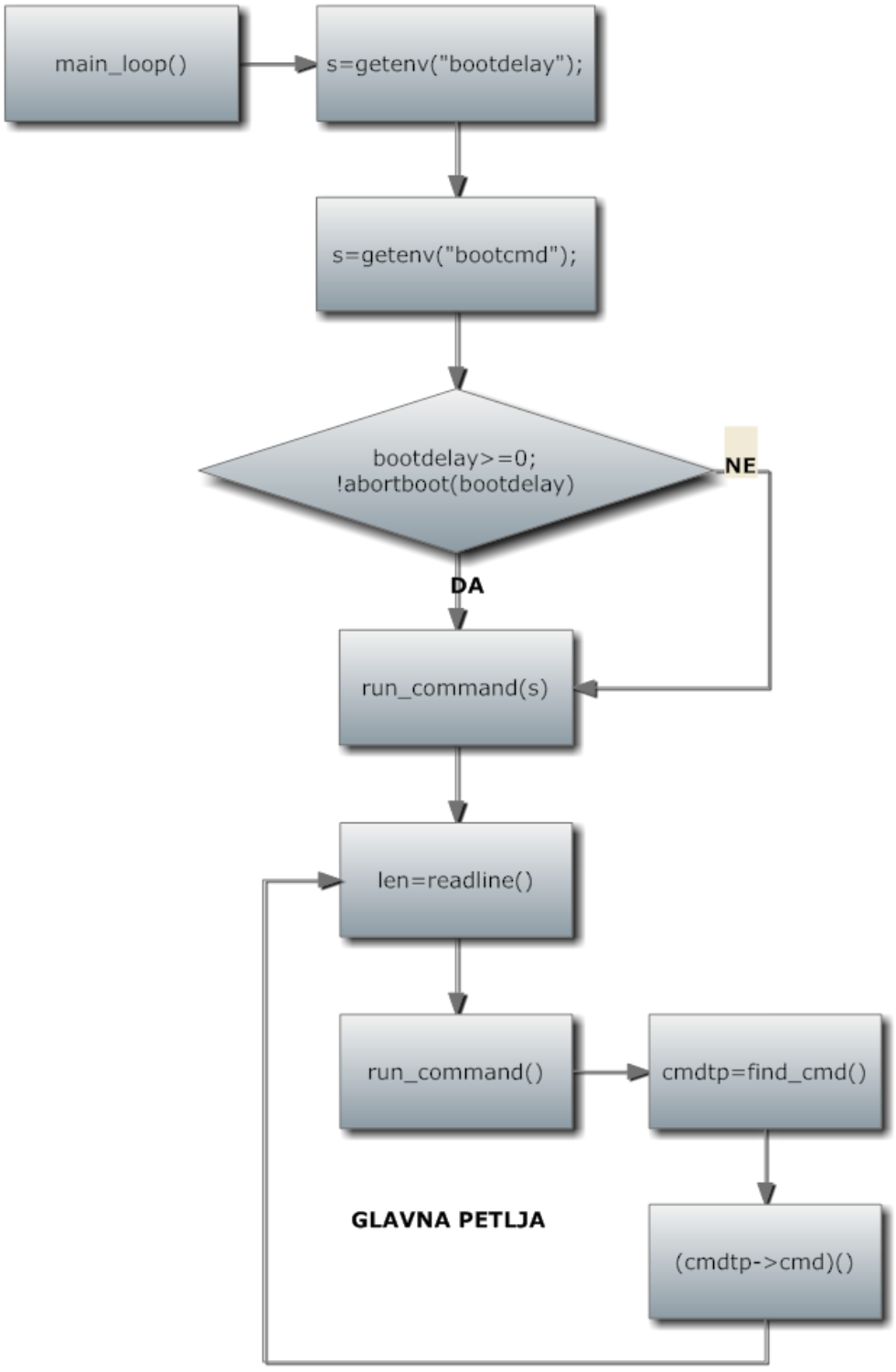
```
init_fnc_t *init_sequence[] = {
        cpu_init,               /* cpu/arm720t/cpu.o */
        board_init,             /* board/lpc_2478_stk/lpc_2478_stk.c */
        interrupt_init,         /* cpu/arm720t/interrupts.c */
        env_init,               /* common/env_flash.c */
        serial_init,            /* cpu/arm720t/serial.c */
        console_init_f,         /* common/console.c */
#if defined(CONFIG_DISPLAY_CPUINFO)
        print_cpuinfo,          /* board/lpc_2478_stk/lpc_2478_stk.c */
#endif
        dram_init,              /* board/lpc_2478_stk/lpc_2478_stk.c */
        NULL,
};

void start_armboot (void)
{
        init_fnc_t **init_fnc_ptr;

        /* ... */

        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr)
{
                if ((*init_fnc_ptr)() != 0) {
                        hang ();
                }
        }

        /* ... */
}
```

Izvan dotične funkcije definirano je polje naziva funkcija *init_sequence[]* koje se pozivaju unutar *for()* petlje funkcije *void start_armboot (void)*. Pokraj naziva funkcija koje se trebaju pozvati navedeno je gdje su definirane pa se čitatelj upućuje da svakako pogleda te funkcije. Umjesto gore navedenih inicijalizacijskih funkcija postoji funkcija koja konfigurira FLASH (*flash_init*), inicijalizira sistemsku okolinu U-boot-a (*env_relocate*), postavlja IP i MAC adresu iz varijabli sistemske okoline (*ipaddr, ethaddr*) koje se definiraju unutar *include/config/lpc_2478_stk.h* datoteke kao CONFIG_EXTRA_ENV_SETTINGS konstante, omogućava prekide (*enable_interrupts*), inicijalizira ethernet (*eth_initialize*) itd. Na dnu funkcije *void start_armboot (void)* nalazimo beskonačnu petlju poziva prema funkciji void

main_loop (void) koja se nalazi unutar *common/main.c* datoteke i predstavlja sljedeću veoma važnu funkciju.

```
        /* ... */
        for (;;) {
                main_loop ();
        }
}
```

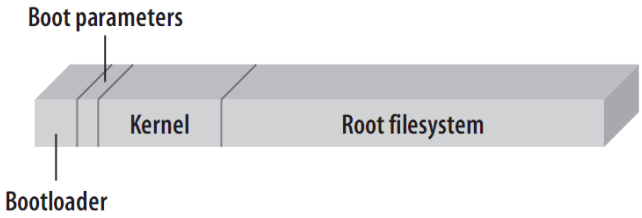
Već smo naglasili da se U-boot konfigurira pomoću konstanti koje se nalaze unutar zaglavne datoteke za pojedinu platformu, npr. za korišteni Olimex-ov razvojni sustav zaglavnu datoteku možemo pronaći unutar *include/configs/lpc_2478_stk.h*. Pomoću konfiguracijskih konstanti U-boot kontrolira izvođenje kôda i vrijednosti raznih parametara koji predstavljaju okolišne varijable U-boot-a. Primjerice, konfiguracijska konstanta CONFIG_BOOTDELAY određuje vrijednost parametra *bootdelay* koji je odgovoran za vrijeme čekanja prije podizanja odgovarajućeg operacijskog sustava. U-boot koristi funkciju *getenv()* kako bi dohvatio vrijednost tih parametara i inicijalizirao odgovarajuće lokalne varijable funkcija. Na sljedećoj slici možemo vidjeti tok izvođenja U-boot programa unutar glavne petlje, tj. unutar funkcije *main_loop()*. Nakon što istekne određeno vrijeme čekanja (ako nije prekinuto od korisnika), U-boot pokreće naredbu koja je postavljena unutar parametra *bootcmd* kao inicijalnu naredbu koja određuje kako će se podizati operacijski sustav, koristeći se funkcijom *run_command()*. Inicijalna naredba može biti bilo koja od podržanih unutar U-boot-a, ali obično se vrijeme čekanja koristi u slučajevima kada možemo birati između više operacijskih sustava na nekoj platformi. Ako korisnik prekine vrijeme čekanja onda se jednostavno preskače izvođenje inicijalne naredbe i kontrola se predaje korisniku koji unosi naredbe unutar konzole. U-boot čita naredbe pomoću funkcije *readline()*, pronalazi odgovarajuću naredbu pomoću funkcije *find_cmd()* i poziva funkciju koja će izvršiti unesenu naredbu. Nakon toga se program ponovno vraća na naredbu *readline()* i tako se zatvara petlja izvođenja.



Slika 22. Tok izvođenja glavne petlje U-boot-a

5.3. U-boot i Linux

U-boot bootloader je zamišljen da se primarno koristi kod ugradbenih računalnih sustava koji imaju instaliran operacijski sustav zasnovan na jezgri linux-a, stoga je primarna uporaba U-boot-a upravo u takvim situacijama. Naravno da se U-boot može upotrijebiti i za druge operacijske sustave, ali prilagodba U-boot-a takvim operacijskim sustavima će biti teža, tj. zahtjeva duboko poznavanje U-boot-a. Da bi U-boot mogao pokrenuti Linux operacijske sustave mora imati na raspolaganju dvije stavke unutar memorije: jezgru (engl. *kernel*) i *root* datotečni sustav, kako prikazuje sljedeća slika.



Slika 23. Raspored jezgre i datotečnog sustava u memoriji

Dakako, U-boot može podizati jezgru i datotečni sustav na dva načina: preko unutrašnje memorije koja je već prije napunjena sa jezgrom i datotečnim sustavom ili preko nekog vanjskog uređaja s kojim se može komunicirati podržanim protokolima od bootloader-a (ethernet, usb, serijski...).

Mnoge platforme nemaju dovoljno stalne memorije gdje bi mogli sačuvati komprimiranu jezgru i datotečni sustav pa stoga projektanti pribjegavaju podizanju jezgre i datotečnog sustav preko nekog vanjske uređaja. U tom slučaju potrebno je napisati upravljački program za periferiju koja podržava odgovarajući protokol komunikacije sa vanjskim uređajem[32]. Obično vanjski uređaj predstavlja stolno računalo s kojim U-boot najčešće komunicira putem serijskog ili ethernet protokola. Također, vanjski uređaj može biti USB flash memorija ili MMC kartica. U prilogu se može pročitati o tome kako se koriste naredbe U-boot-a kako bi se pribavila jezgra i datotečni sustav kod Olimex-ovog LPC2478STK razvojnog sustava. Detalji implementacije upravljačkih programa koji bi upravljali takvim uređajima neće biti opisivani u nastavku.

[32] To prelazi granice ovoga rada, iako treba naglasiti da se upravljački programi napisani za (uC)Linux mogu iskoristiti za U-boot jer se pišu na sličan način. Naravno, uz prihvaćanje uvjeta GPL licence.

Također, postoje platforme koje imaju dovoljno memorije da mogu spremiti komprimiranu jezgru i datotečni sustav. Ako u takvim slučajevima želimo pozvati jezgru operacijskog sustava i pripadajući datotečni sustav onda koristimo posebnu naredbu – *bootm*. Obično se u takvim slučajevima jezgra operacijskog sustava i pripadajući datotečni sustav komprimiraju i tako spremaju u memoriju. Da bi se takva komprimirana datoteka prilagodila U-boot bootloader-u koristi se jedan alat koji se nalazi unutar */u-boot-x.x.x/tools* mape pod nazivom *mkimage*. To je naredba koja dodaje određeno zaglavlje na arhiviranu datoteku koje sadrži informacije o adresi punjenja i početka izvođenja jezgre, korištenom algoritmu za kompresiju, CRC provjeri sigurnosti, arhitekturi procesora, tipu operacijskog sustava kako je prikazano sljedećim odsječkom kôda iz zaglavne datoteke */include/image.h*.

```
/*
 * Operating System Codes
 */
#define IH_OS_INVALID         0
#define IH_OS_LINUX           5
#define IH_OS_SOLARIS         8
#define IH_OS_LYNXOS          13
#define IH_OS_VXWORKS         14
#define IH_OS_ARTOS           19
#define IH_OS_UNITY           20
/*
 * CPU Architecture Codes (supported by Linux)
 */
#define IH_CPU_INVALID        0
#define IH_CPU_ARM            2
#define IH_CPU_I386           3
#define IH_CPU_MIPS           5
#define IH_CPU_PPC            7
#define IH_CPU_M68K           12
#define IH_CPU_NIOS           13
#define IH_CPU_MICROBLAZE     14
#define IH_CPU_BLACKFIN       16
#define IH_CPU_AVR32          17
#define IH_CPU_ST200          18
/*
 * Image Type Codes
 */
#define IH_TYPE_INVALID       0
#define IH_TYPE_STANDALONE    1
#define IH_TYPE_KERNEL        2
#define IH_TYPE_RAMDISK       3
#define IH_TYPE_MULTI         4
#define IH_TYPE_FIRMWARE      5
#define IH_TYPE_SCRIPT        6
#define IH_TYPE_FILESYSTEM    7
#define IH_TYPE_FLATDT        8
/*
 * Compression Types
 */
```

```
#define IH_COMP_NONE           0
#define IH_COMP_GZIP           1
#define IH_COMP_BZIP2          2
#define IH_MAGIC       0x27051956      /* Image Magic Number          */
#define IH_NMLEN               32      /* Image Name Length           */


typedef struct image_header {
        uint32_t        ih_magic;       /* Image Header Magic Number   */
        uint32_t        ih_hcrc;        /* Image Header CRC Checksum   */
        uint32_t        ih_time;        /* Image Creation Timestamp    */
        uint32_t        ih_size;        /* Image Data Size             */
        uint32_t        ih_load;        /* Data  Load  Address         */
        uint32_t        ih_ep;          /* Entry Point Address         */
        uint32_t        ih_dcrc;        /* Image Data CRC Checksum     */
        uint8_t         ih_os;          /* Operating System            */
        uint8_t         ih_arch;        /* CPU architecture            */
        uint8_t         ih_type;        /* Image Type                  */
        uint8_t         ih_comp;        /* Compression Type            */
        uint8_t         ih_name[IH_NMLEN];     /* Image Name           */
} image_header_t;
```

Istaknuta je struktura pod nazivom *image_header* koja je potrebna da bi bootloader znao manipulirati s navedenim zaglavljem. Postoje dva načina kako instalirati *mkimage* alat: kompajlirati skup alata koji dolazi s distribucijom U-boot-a naredbom *make tools* unutar mape */u-boot-x.x.x* i nakon toga dodamo alat u PATH varijablu ili instalirati paket koji dolazi s vašom distribucijom Linux-a. Ako koristite preporučenu Linux distribuciju (Ubuntu) onda se odgovarajući paket nalazi pod nazivom *uboot-mkimage* i jednostavno se dodaje naredbom u terminalu:

```
sudo apt-get install uboot-mkimage.
```

Nakon što smo instalirali traženi alat koristimo ga na sljedeći način:

```
mkimage -A arch -O os -T type -C compress -a loadaddr -e entrypoint -n name -
d data_file[:data_file] outputimage
```

Vidimo da argumenti naredbe *mkimage* odgovaraju pojedinim poljima unutar strukture *image_header*. Primjerice, ako želimo generirati datoteku *vmlinux-2.4.18.img* od komprimirane Linux jezgre *Kernel 2.4.18* za ARM arhitekturu, koristimo sljedeći niz naredbi:

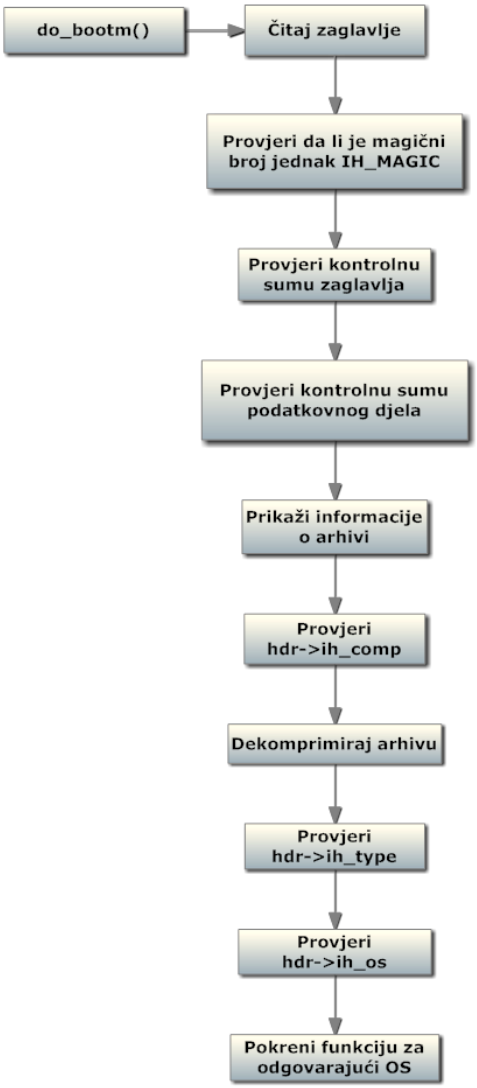
```
gzip -9 <Image> Image.gz

mkimage -n „Kernel 2.4.18“ -A arm -O linux -T kernel -C gzip   -a 30008000 -e
30008000 -d Image.gz vmlinux-2.4.18.img
```

U nastavku ćemo opisati tok izvođenja spomenute U-boot naredbe *bootm* koja se nalazi unutar */common/cmd_bootm.c* datoteke. Nakon što smo uspostavili sučelje terminala s bootloader-om U-boot možemo upisivati naredbe u komandnu

liniju. Upisom naredbe *bootm* <*adresa*> unutar konzole U-boot-a pozivamo jezgru operacijskog sustava koja je smještena u memoriji na adresi <*adresa*>. Na početku funkcije čitamo zaglavlje arhivirane datoteke koja može biti jezgra operacijskog sustava, datotečni sustav ili nešto treće. Nakon toga slijedi čitav niz provjera: provjera magičnog broja koji mora biti jedna konstanti IH_MAGIC definiranoj unutar */include/image.h*, provjera kontrolne CRC sume za zaglavlje i podatkovni dio arhive. Nakon toga prikazujemo informacije o arhivi i provjeravamo korišteni algoritam za kompresiju, dekomprimiramo arhivu i provjeravamo tip zaglavlja arhive (STANDALONE, KERNEL[33], MULTI[34] ...).

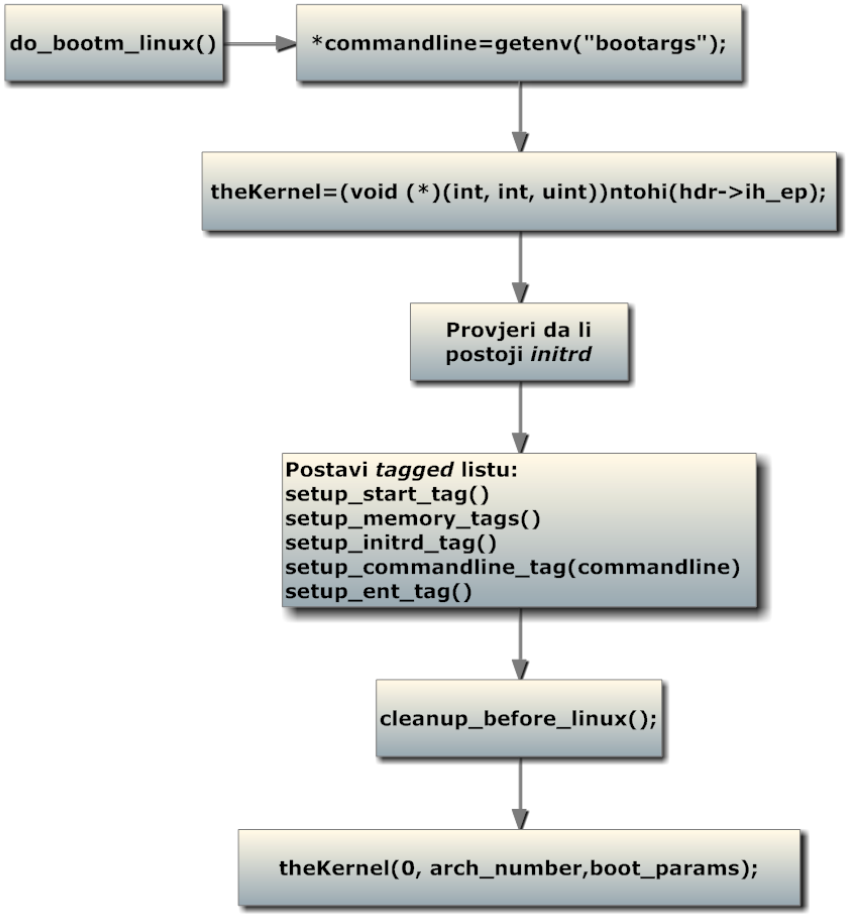
[33] Ako se radi o jezgri operacijskog sustava

[34] Ako se radi o datotečnom sustavu



Slika 7. Tok izvođenja funkcije *do_bootm()*

Zadnji korak je provjera tipa operacijskog sustava i pokretanje odgovarajuće funkcije koja će pripremiti okolinu za izvođenje odabranog operacijskog sustava. Za operacijski sustav koji je baziran na linux jezgri tip je definiran konstantom IH_OS_LINUX unutar zaglavlja arhive i pokreće se funkcija *do_bootm_linux()* koja se nalazi unutar */lib_arm/armlinux.c*. Tok izvođenja te funkcije dan je na sljedećoj slici. Uloga te funkcije jest da pribavi argumente za jezgru operacijskog sustava od bootloader-a, pročita adresu početka izvođenja jezgre (*hdr->ih_ep*), provjeri postojanje datotečnog sustava unutar memorije, postavi *tagged* listu, pripremi[35] okolinu prije pokretanja linux-a (*cleanup_before_linux()*) i pokrene jezgru.



Slika 8. Tok izvođenja funkcije *do_bootm_linux()*

[35] Onemogućavanje prekida, I/D-cache memorija (ako ih platforma podržava)

5.4. Datotečni sustav

Root datotečni sustav obično uzima formu inicijalnog datotečnog sustava kojeg nalazimo na Linux operacijskom sustavu – tzv. *initrd* (inital RAM disk). Kod Linux operacijskih sustava na osobnim računalima inicijalni datotečni sustav se montira prije određenog stalnog *root* datotečnog sustava (FAT, NFS, EXT3...) i stoga *initrd* traje jedan kratak period vremena. Naime, konačni *root* datotečni sustav se montira koristeći se funkcionalnostima jezgrenog modula. Da bi se jezgreni modul instalirao potreban je alat *insmod* i minimalni skup sistemskih direktorija koje upravo osigurava *initrd* datotečni sustav. Međutim, kod ugradbenih računalnih sustava *initrd* datotečni sustav je u većini slučajeva stalni *root* datotečni sustav jer kod ugradbenih aplikacija obično nemamo tvrdi disk (*hard drive*), tj. stalnu memoriju većeg kapaciteta. Na sljedećoj slici možemo vidjeti uobičajeni skup direktorija unutar *initrd* datotečnog sustava koji ovisi o aplikaciji za koju se *initrd* montira.

```
drwxr-xr-x  10 root root    4096 May 7 02:48 .
drwxr-x---  15 root root    4096 May 7 00:54 ..
drwxr-xr-x  2  root root    4096 May 7 02:48 bin
drwxr-xr-x  2  root root    4096 May 7 02:48 dev
drwxr-xr-x  4  root root    4096 May 7 02:48 etc
-rwxr-xr-x  1  root root     812 May 7 02:48 init
-rw-r--r--  1  root root 1723392 May 7 02:45 initrd-2.6.14.2.img
drwxr-xr-x  2  root root    4096 May 7 02:48 lib
drwxr-xr-x  2  root root    4096 May 7 02:48 loopfs
drwxr-xr-x  2  root root    4096 May 7 02:48 proc
lrwxrwxrwx  1  root root       3 May 7 02:48 sbin -> bin
drwxr-xr-x  2  root root    4096 May 7 02:48 sys
drwxr-xr-x  2  root root    4096 May 7 02:48 sysroot
```

Slika 24. Skup direktorija unutar *initrd*

Inicijalni datotečni sustav *initrd* možemo montirati kao ext2, ramfs, cramfs ili jffs2 datotečni sustav, ovisno o aplikaciji[36]. U nastavku ćemo opisati postupak izgradnje inicijalnog datotečnog sustava koji će predstavljati i konačni *root* datotečni sustav. Postupak će se predstaviti putem izgradnje takvog datotečnog sustava[37] na osobnim Linux računalima, ali postupak je sličan i za ugradbene sustave, s tom razlikom da se kompajliranje odvija pomoću *cross*-kompajlera. U

[36] Kod Olimex-ovog LPC2478STK razvojnog sustava *initrd* je montiran kao ramfs datotečni sustav

[37] U primjeru koristimo ext2 arhitekturu datotečnog sustava

nastavku je dana skripta koja stvara *initrd*. Prije svega stvorimo praznu datoteku koristeći */dev/zero* (standardni tok nula) kao ulaz koji zapisujemo u *ramdisk.img* datoteku. Nastala datoteka je veličine 4MB, tj. 4000 blokova veličine 1K. Nakon toga stvaramo ext2 datotečni sustav naredbom *mke2fs* koristeći prethodno stvorenu datoteku. Nakon što je stvorena datoteka postala ext2 datotečni sustav montiramo ju unutar */mnt/initrd* koristeći *loop device*[38]. Na mjestu gdje smo montirali datoteku nastao je direktorij koji predstavlja ext2 datotečni sustav kojeg možete napuniti sadržajem kako je opisano u nastavku. Dakle, sljedeći korak je stvaranje subdirektorija koji ostvaraju *root* datotečni sustav: */bin, /dev, /proc, /sys*... Da bi *root* datotečni sustav bio koristan trebalo bi dodati razne uslužne programe poput *ash, awk, sed, insmod* i drugih, a to nam omogućava BusyBox. Prednost BusyBox-a je u pakiranju raznih uslužnih programa u jedan jedinstveni paket tako što pronalazi zajedničke elemente svakog uslužnog programa i tako smanjuje konačnu veličinu datoteke. To je posebno pogodno za ugradbene računalne sustave zbog njihove relativno male i ograničene memorije. Nadalje, kopirajmo BusyBox u */bin* direktorij i stvorimo nekoliko simboličkih linkova koji su svi povezani na BusyBox. Imena linkova su naredbe koje želimo omogućiti, a prilikom njihova poziva BusyBox će automatski odrediti koja je naredba pozvana. Nakon toga montiramo odgovarajuće datoteke koje su potrebne za upravljačke programe. U ovom primjeru datoteke su kopirane iz */dev* direktorija koji već postoji unutar postojeće Linux distribucije na osobnom računalu. Predzadnji korak je stvaranje *linuxrc*[39] datoteke u kojoj se ostvaruje osnovno postavljanje okoline kao što je montiranje */proc* i /sys datotečnog sustava. Na kraju pozivamo *ash* kako bi mogli ostvariti interakciju s našim upravo stvorenim *root* datotečnim sustavom i komprimiramo ga koristeći *gzip* alat. Unutar uCLinux distribucije datotečni sustav se dodaje prilikom konfiguracije jezgre operacijskog sustava. Mi nećemo ulaziti u detalje toga postupka unutar ovoga rada, ali ćemo prikazati kako se mogu dodavati vlastiti direktoriji u već postojeći datotečni sustav.

[38] *Loop* uređaj je upravljački program koji omogućava montiranje datoteke kao blok uređaja i interpretiranje datotečnog sustava kojeg predstavlja

[39] Jezgra poziva ovu datoteku kao startup skriptu u slučaju da u konačnom datotečnom sustavu ne nađe *init* datoteku.

```
#!/bin/bash

rm -f /tmp/ramdisk.img
rm -f /tmp/ramdisk.img.gz

RDSIZE=4000
BLKSIZE=1024

dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE

/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE

mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0

mkdir /mnt/initrd/bin
mkdir /mnt/initrd/sys
mkdir /mnt/initrd/dev
mkdir /mnt/initrd/proc

pushd /mnt/initrd/bin
cp /usr/local/src/busybox-1.1.1/busybox .
ln -s busybox ash
ln -s busybox mount
ln -s busybox echo
ln -s busybox ls
ln -s busybox cat
ln -s busybox ps
ln -s busybox dmesg
ln -s busybox sysctl
popd

cp -a /dev/console /mnt/initrd/dev
cp -a /dev/ramdisk /mnt/initrd/dev
cp -a /dev/ram0 /mnt/initrd/dev
cp -a /dev/null /mnt/initrd/dev
cp -a /dev/tty1 /mnt/initrd/dev
cp -a /dev/tty2 /mnt/initrd/dev

# sbin=bin
pushd /mnt/initrd
ln -s bin sbin
popd

# Stvaranje init datoteke
cat >> /mnt/initrd/linuxrc << EOF
#!/bin/ash
echo
echo "jednostavan initrd je sada aktivan"
echo
mount -t proc /proc /proc
mount -t sysfs none /sys
/bin/ash --login
EOF

chmod +x /mnt/initrd/linuxrc

umount /mnt/initrd
gzip -9 /tmp/ramdisk.img
cp /tmp/ramdisk.img.gz /boot/ramdisk.img.gz
```

5.5. Podizanje Linux-ove jezgre pomoću U-boot bootloader-a

Prije nego preda kontrolu jezgri linux operacijskog sustava U-boot obično prolazi kroz sljedeće korake:

1. Inicijalizira osnovno sklopovlje
    a. Procesor - detekcija tipa i određivanje brzine
    b. Memorija - vremenski parametri
    c. RAM - detekcija lokacije i veličine te inicijalizacija memorije
2. Inicijalizira periferno sklopovlje
    a. Sklopovlje za koje je napisan upravljački program
    b. UART – predstavlja jezgreni terminal
3. Kopira jezgru i datotečni sustav u RAM
    a. Rezervira blok kontinuirane fizičke memorije posebno za jezgru i posebno za datotečni sustav
    b. Kopira (ili preuzima sa vanjske memorije) i dekomprimira (ako je potrebno) jezgru i datotečni sustav u rezervirane blokove memorije
4. Postavlja listu u kojoj se nalaze podaci i parametri koje bootloader prosljeđuje jezgri[40] – boot argumenti
5. Poziva jezgru linux-a sa sljedećim parametrima:
    a. Parametri procesorskih registara
        i. R0=0
        ii. R1=specifični broj koji označava procesorsku arhitekturu[41]
        iii. R2=fizička adresa *tagged* liste u RAM memoriji
    b. Način rada procesora
        i. Svi prekidi trebaju biti onemogućeni (IRQ i FIQ)
        ii. Procesor mora biti u supervisor načinu rada (SVC)
    c. Cache, MMU[42]
        i. MMU mora biti onemogućen
        ii. D-cache mora biti onemogućen, I-cache ne mora biti onemogućen
    d. DMA
        i. DMA prema/od uređaja treba biti neaktivna
6. Predaja kontrole jezgri operacijskog sustava

[40] Kernel Tagged List

[41] Mora se slagati sa jednom od pretprocesorskih konstanti unutar linux/arch/arm/tools/mach-types.

[42] Ova stavka nije bitna kod korištenog razvojnog sustava jer procesor koji se koristi ne podržava MMU i cache memoriju.

Upravo smo vidjeli da U-boot prenosi podatke jezgri linux-a preko procesorskih registara (R0, R1 i R2) i boot argumenata koji su sadržani unutar takozvane *tagged* liste. *Tagged* lista mora biti smještena na one adrese unutrašnje memorije gdje neće biti prebrisana[43] od jezgre ili datotečnog sustava, tj. alata koji nad potonjima vrše određene operacije (npr. dekomprimiranje jezgre). Preporučeno mjesto za smještanje liste je u prvih 16KB RAM memorije pri čemu fizička adresa početka *tagged* liste mora biti smještena unutar procesorskog registra R2. Lista mora biti poravnana na granicu od jedne riječi (32b), mora početi na ATAG_CORE adresi u memoriji i završiti na ATAG_NONE adresi[44]. Svaki član liste (*tag*) se sastoji od strukture koja predstavlja zaglavlje i pripadajućeg podatkovnog dijela. Zaglavlje sadrži dvije 32-bitne vrijednosti – veličinu podatkovnog dijela (u koju je uključena veličina zaglavlja) i identifikacijska vrijednost člana liste po kojoj se određuje o kojem se točno članu radi (ATAG_CORE, ATAG_NONE, ATAG_SERIAL, ATAG_MEM – vidi tablicu 1).

Bootloader mora prenijeti jezgri minimalno jedan važan argument – ATAG_MEM koji predstavlja veličinu i lokaciju sistemske memorije. Također, postoje i drugi *tag*-ovi koji su prikazani u tablici 1. Na slici ispod navedene tablice nalazi se slikoviti prikaz *tagged* liste.

Ako čitatelj želi dublje razumjeti prikazane koncepte upućuje se da prouči U-boot implementaciju *tagged*[45] liste koja se nalazi unutar datoteke *lib_arm/armlinux.c*.

[43] Jezgra ne provjerava granice pojedinih dijelova u memoriji

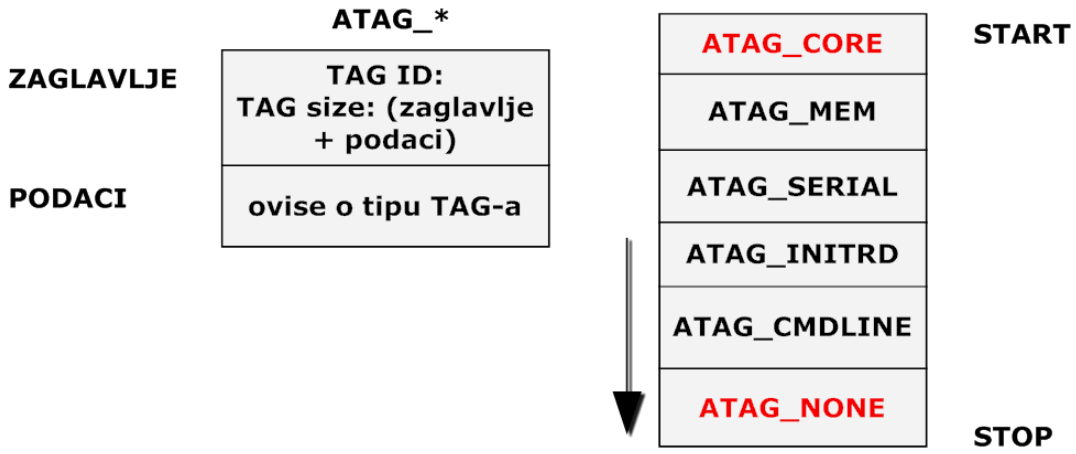
[44] Važno je istaknuti da se o svemu ovome u pozadini brine U-boot i korisnik ne mora implementirati *tagged* listu već samo specificira boot argumente unutar konfiguracijskog zaglavlja pod CONFIG_BOOTARGS.

[45] U nekim slučajevima bootloader ne treba postavljati *tagged* listu jer je ona ugrađena unutar jezgre operacijskog sustava

Tablica 6. Opis članova *tagged* liste

Ime člana liste	*Vrijednost*	*Veličina*	*Opis*
ATAG_NONE	0x00000000	2	Označava kraj liste
ATAG_CORE	0x54410001	5[46]	Označava početak liste
ATAG_MEM	0x54410002	4	Opisuje fizičku memoriju
ATAG_VIDEOTEXT	0x54410003	5	Opisuje VGA prikaz teksta
ATAG_RAMDISK	0x54410004	5	Opisuje kako će se r*amdisk* koristiti unutar jezgre
ATAG_INITRD2	0x54420005	4	Pokazuje gdje je smješten komprimirani *ramdisk* unutar memorije
ATAG_SERIAL	0x54410006	4	64b serijski broj pločice
ATAG_REVISION	0x54410007	3	32b broj verzije pločice
ATAG_VIDEOLFB	0x54410008	8	Inicijalne vrijednosti za vesafb tip framebuffer-a
ATAG_CMDLINE	0x54410009	2+((duljina_cmdline+3)/4)	Naredbena linija koja se prenosi jezgri

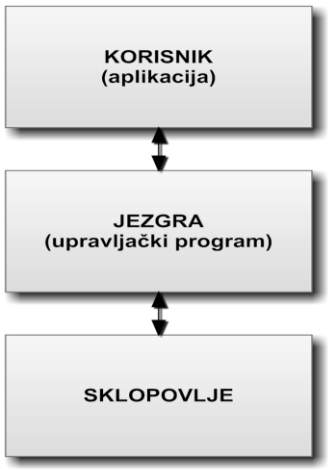
[46] Veličina je 2 ako je prazan jer je to upravo veličina strukture koja predstavlja zaglavlje



Slika 25. Prikaz *tagged* liste

6. Pisanje upravljačkih programa za uCLinux

Upravljački programi unutar jezgre Linux-a imaju posebnu ulogu. Možemo ih zamisliti kao „crne kutije“ koje omogućavaju da funkcionalnost određenih sklopovskih modula unutar nekog procesora odgovara programskom sučelju koje jezgra Linux-a pruža korisniku prilikom pisanja aplikacija. Dakle, upravljački programi „sakrivaju“ detalje o tome kako određeni uređaj radi na sklopovskoj razini i omogućavaju aplikacijskom programeru kao korisniku operacijskog sustava jednostavnije programiranje pozivanjem standardnih sistemskih funkcija koje su neovisne o specifičnom upravljačkom programu. Uloga upravljačkog programa se upravo krije u povezivanju specifičnih sklopovskih operacija sa standardnim sistemskim funkcijama koje se nude aplikaciji. To programsko sučelje omogućava da se upravljački program može implementirati odvojeno od ostatka jezgre operacijskog sustava i uključiti u onom trenutku kada ga odgovarajuća aplikacija zatreba. Upravo ovakva modularnost upravljačkih programa olakšava projektiranje cijelog sustava zasnovanog na jezgri Linux-a. Na sljedećoj slici možemo vidjeti grafički prikaz modela zasnovanog na upravljačkim programima.



Slika 26. Model zasnovan na upravljačkim programima

Prilikom pisanja upravljačkih programa programer mora obratiti pozornost na osnovni princip ovakvog modela: upravljački program treba pisati tako da omogućava pristup sklopovlju, ali nije dozvoljeno pritom prisiljavati i ograničavati korisnika na uzak spektar aplikacija koje će na određeni način koristiti upravljački program za sklopovlje. Dakle, na korisniku je da odluči kako će koristiti sklopovlje za svoje aplikacije, a upravljački program treba biti dovoljno fleksibilan da to

omogući bez dodavanja specifičnih zahtjeva. Većina upravljačkih programa dolazi zajedno za korisničkim aplikacijama koje služe kao dobar ogledni primjer i ujedno omogućavaju pristup sklopovlju. Ove aplikacije mogu biti jednostavni alati ili kompletna grafička sučelja.

Linux razlikuje tri osnovna tipa uređaja (engl. *devices*), tj. tri osnovna tipa upravljačkih programa koji upravljaju tim uređajima:

- *Character* uređaji/upravljački programi
    - ovakvom tipu uređaja se može pristupiti putem niza podataka veličine jednog okteta (8 bitova), kao da se radi o običnoj datoteci. *Char* upravljački program ima funkciju da implementira takav pristup od strane korisničkih aplikacija. Ovakav upravljački program obično implementira minimalno četiri važna sistemska poziva: *open*, *read*, *write* i *close*. Primjeri takvih uređaja unutar Linux distribucije na osobnim računalima su konzola (*/dev/console*), serijski portovi (*/dev/ttyS0* i ostali), itd. Navedene datoteke u zagradama pored odgovarajućih *char* uređaja predstavljaju tzv. čvorove unutar datotečnog sustava koji predstavljaju podatkovne kanale komunikacije između aplikacije i upravljačkog programa.
- *Block* uređaji/upravljački programi
    - Primjer takvog uređaja unutar Linux operacijskog sustava je tvrdi disk. Ovakvi uređaji mogu na sebi implementirati cijeli datotečni sustav i zahtijevaju drugačiji način pristupa od *char* uređaja. *Block* uređaji mogu upravljati takvim ulazno/izlaznim operacijama koje prenose jedan ili više cijelih blokova podataka koji su obično veličine 512 okteta ili neka veća potencija broja 2. Međutim, Linux dopušta korisničkim aplikacijama i oktetni pristup takvih uređajima, tj. da šalju ili primaju podatke od takvih uređaja kao da su *char* tipa. Dakle, jedina razlika između *block* i *char* uređaja je u načinu na koji jezgra upravlja podacima i programskom sučelju između jezgre i takvog upravljačkog programa. Također, svakom *block* uređaju se može pristupiti preko čvorova u datotečnom sustavu kao i *char* uređaji.

- *Network* uređaj/upravljački programi
    - Ovakav tip upravljačkog programa je različit od preostala dva u tome što upravlja paketima podataka za više klijenata radije nego nizom podataka za jednog klijenta. Budući da ne upravlja nizovima podataka kao ostala dva uređaja za *network* uređaj nije moguće imati datotečni princip komunikacije korisničkih aplikacija i upravljačkog programa. Nadalje, podaci dolaze na mrežni uređaj (karticu) asinkrono iz izvora koji je van cijelog sustava, tj. s kojim sustav komunicira određenim protokolom. Upravo zbog ovih ključnih razlika u odnosu na preostala dva tipa uređaja ovakav upravljački program zahtjeva potpuno drugačije sučelje prema jezgri i zbog toga nije predmet razmatranja unutar ovog rada.

U nastavku ovoga poglavlja objasnit ćemo postupak projektiranja uCLinux upravljačkog programa za CAN modul razvojnog sustava LPC2478STK koji je detaljno prikazan u prethodnim poglavljima. Također, napravljena je aplikacija (*cantest*) koja predstavlja ogledni primjer kako koristiti napisani upravljački program. Napisani upravljački program izveden je kao tip *char* i dan je kao primjer pisanja upravljačkih programa unutar te grupe; ukoliko čitatelj želi više saznati kako napisati upravljačke programe za druge dvije važne grupe upravljačkih programa upućuje se na literaturu [1].

6.1. Specifičnosti programiranja unutar jezgre

Sada je vrijeme da naglasimo činjenice o kojima početnik često ne razmišlja dok promatra kôd jezgre operacijskog sustava. Dok gledamo jezgrin kôd možemo vidjeti da se koristi funkcija *printk* za ispisivanje poruka prema nekom od standardnim izlaza ili možemo vidjeti da prilikom uključivanja različitih zaglavnih datoteka (*.h) nema uobičajenih datoteka kao što su <stdio.h>, <stdarg.h> i ostali. Razlog tomu je što jezgreni kôd ne koristi *libc* biblioteku standardnih C funkcija jer jezgra implementira svoje sistemske funkcije i svoje zaglavne datoteke. Dakle, prilikom projektiranja upravljačkih program zaboravimo na standardno aplikacijsko programiranje i nazive funkcija na koje smo navikli jer su oni dio jezgrenog kôda. Također, postoji još jedna važna razlika između aplikacijskog programiranja i programiranja unutar jezgre – sigurnost; greška unutar korisničke aplikacije neće

biti toliko pogubne kao greške koje su nastale programiranjem unutar jezgrenog kôda jer takve dovode do toga da se barem trenutni proces terminira, ako ne i cijeli operacijski sustav. Dakle, operacijski sustavi jasno odjeljuju korisnički prostor djelovanja od jezgrenog, u terminologiji engleskog jezika razlikujemo *User space* i *Kernel space*. Najvažniji razlog tomu je sigurnost jer aplikacijski programer kao korisnik operacijskog sustava ne bi smio imati pristup svim resursima operacijskog sustava. Većina današnjih procesora omogućavaju različite načine rada koja nemaju jednaka prava nad korištenjem različitih resursa pa se upravo to svojstvo koristi u operacijskim sustavima (među ostalima o kojima neće biti riječi unutar ovog rada) da bi se jasno odvojio korisnički prostor od jezgrenog. Jezgra se izvodi u najvišem modu rada procesora (tzv. *supervisor* mod rada) u kojem sve dozvoljeno, dok se aplikacije izvode u najnižem modu rada procesora (tzv. *user* mod rada) u kojem procesor regulira direktan pristup sklopovlju ili nedozvoljeni pristup pojedinim dijelovima memorije. Kada god aplikacija izvrši sistemski poziv prema jezgri ili ako je prekinuta nekim sklopovskim prekidom, jezgra Linux-a automatski prebacuje izvođenje iz korisničkog prostora u jezgreni prostor.[47] Jezgreni kôd koji izvršava sistemski poziv kaže se da radi u *kontekstu* procesa koji je izvršio odgovarajući sistemski poziv i može pristupiti podacima adresom prostoru procesa. Zadnja činjenica u ovom izlaganju predstavlja još jednu veliku razliku između aplikacijskog programiranja i jezgrenog programiranja: istovremeno izvođenje velikog broja programa. Problem istovremenosti u aplikacijskom programiranju gotovo da ne postoji ako izuzmemo *multithreading* aplikacije. Korisnički program se izvodi od početka do kraja bez da korisnik mora razmišljati o tome da će „nešto“ prekinuti takvo izvođenje. Jezgra se ne izvodi u tako jednostavnom svijetu i moramo biti svjesni da se mnogo programa izvodi istovremeno i mogu pristupiti upravo istim resursima koja smo rezervirali prilikom pisanja vlastitog upravljačkog programa. Naravno, postoje različita rješenja za takve probleme unutar jezgre i objasniti ćemo njihovo korištenje unutar upravljačkog programa koji je napisan. Izvori istovremenosti unutar jezgre mogu biti sklopovski prekidi, programski prekidi (jezgreni vremenski programi) ili izvođenje jezgre na višeprocesorskim računalima. Kao posljedica toga jezgrin kôd

[47] Obadva slučaja susrećemo kod upravljačkih programa

mora biti *reentrant* što znači da ga u isto vrijeme može koristiti više različitih programa. Također, kada više programa koristi dijeljene strukture podataka moguće su razne neželjene situacije koje mogu promijeniti ili uništiti podatke. Unutar ovog rada upravljački program je napisan s velikom pozornošću na ovakve situacije.

6.2. Jezgreni moduli

Jedna od najvažnijih mogućnosti jezgre operacijskog sustava (uC)Linux je mogućnost dodavanja određenih funkcionalnosti jezgre (npr. upravljačkih programa) tijekom izvršavanja operacijskog sustava. Svaki takav dio kôda koji se može dodati jezgri operacijskog sustava za vrijeme izvršavanja naziva se *modul*. Svaki modul se sastoji od objektnog kôda koji se dinamički uključuje u jezgru koristeći *insmod* program ili isključuje iz jezgre koristeći *rmmod* program.

Svaki modul obično implementira jedan od tipova uređaja koji su navedeni u prošlom poglavlju, pa razlikujemo *char*, *block* i *network* module. Ova podjela modula nije stroga i „slobodniji“ programeri mogu stvoriti veliki modul unutar kojeg su implementirani različiti upravljački programi, ali to se ne preporuča. Dobar programer stvara novi modul za svaku novu funkcionalnost koju implementira zadržavajući osnovni zahtjev skalabilnosti i mogućnosti proširenja sustava – modularnost.

Svaki modul mora definirati dvije funkcije: jednu koja će se pozvati onda kada se modul uključi u jezgru programom *insmod* i jednu koja će se pozvati kada se modul isključi iz jezgre operacijskog sustava programom *rmmod*. U prilog tome unutar CAN upravljačkog programa za uCLinux napravljene su dvije funkcije sljedećih prototipova: `int __init lpc2xxx_can_init(void)`, `void __exit lpc2xxx_can_cleanup(void)`. Unutar kôda mogu se uočiti još dva makroa koji opisuju ulogu ovih dviju funkcija, tj. obavještavaju jezgru o kojim se funkcijama radi: `module_init(lpc2xxx_can_init)` i `module_exit(lpc2xxx_can_cleanup)`. Također, postoje još tri korištena makroa vezana za module: `MODULE_AUTHOR()`, `MODULE_DESCRIPTION` i `MODULE_LICENSE()`. Ovi makroi pružaju informaciju korisniku modula o tome tko je autor modula, čemu modul služi i pod kojom licencom se može koristiti. Moduli se mogu uključivati i isključivati po potrebi u toku rada sustava kao što smo već naglasili, ali moguće je inicijalno komapjlirati jezgru s već

uključenim modulima koji će se koristiti, tj. bez potrebe da ih se naknadno uključuje. Tada će jezgra pozivati inicijalizacijsku funkciju nekog modula pri podizanju cijelog sustava i stoga se u definiciji funkcije nalazi oznaka `__init` koja govori jezgri operacijskog sustava da funkcija uz koju stoji taj znak se koristi samo prilikom podizanja sustava, tj. nakon što se modul uključi u jezgru moguće je osloboditi memoriju od te funkcije. Postoji slična oznaka i za podatke koji se koriste samo prilikom inicijalizacije `__initdata`. S druge strane oznaka `__exit` označava da se funkcija poziva onda kada se operacijski sustav isključuje. Ovaj način manipuliranja modulima od strane jezgre operacijskog sustava podsjeća na programiranje vođeno događajima (engl. *event-driven programming*). Dakle, funkcija koja se poziva prilikom isključivanja modula (`__exit`) mora pažljivo osloboditi sve resurse koje je inicijalizacijska funkcija modula (`__init`) zauzela jer inače prilikom ponovnog pokretanja sustava mogu ostati neželjeni ostaci.

6.3. Programsko sučelje

Char uređajima se pristupa putem imena datoteka u datotečnom sustavu. Ove datoteke se često nazivaju čvorovima datotečnog sustava i smještene su unutar */dev* direktorija. Datoteke koje predstavljaju čvorove za *char* uređaje se identificiraju znakom „c“ prilikom pregledavanja sadržaja direktorija naredbom *ls –l*. Pored te oznake možemo vidjeti dva broja na mjestima gdje kod običnih datoteka piše njihova veličina u KB. Spomenuta dva broja nazivamo glavnim i sporednim brojem uređaja. Glavni broj uređaja određuje koji je upravljački program povezan s promatranim uređajem, dok sporedni broj koristi jezgra operacijskog sustava kako bi točno znala na koji se uređaj odnosi odgovarajući upravljački program jer više uređaja može biti upravljano jednakim upravljačkim programom pa stoga njihovi čvorovi unutar */dev* mogu imati jednake glavne brojeve. Često prilikom pisanja upravljačkih programa moramo znati koji je glavni ili sporedni broj kako bi bili u mogućnosti inicijalizirati važne strukture podataka unutar jezgre operacijskog sustava. Iz tog razloga jezgra koristi tip *dev_t*[48] kako bi unutar varijabli toga tipa držala dostupnima glavni i sporedni broj nekog uređaja. Također, jezgra definira dva makroa iz kojih na jednostavan način možemo doći do glavnog i sporednog

[48] Definirana unutar <linux/types.h>

broja nekog uređaja: `MAJOR(dev_t dev) i MINOR(dev_t dev)`. Također, postoji još jedan način na koji možemo saznati glavni i sporedni broj – pomoći tzv. *inode* strukture podataka. Spomenuta struktura se koristi unutar jezgre kako bi predstavljala datoteke i sadrži veliku količinu informacija o datoteci. Unutar te strukture postoji polje `dev_t i_rdev` koje sadrži odgovarajuće brojeve nekog uređaja. U svrhu čitljivijeg kôda, tj. da programeri ne bi direktno pristupali dotičnom polju *inode* strukture, razvijeni su makroi: `unsigned int iminor(struct inode *inode), unsigned int imajor(struct inode *inode)`. Ako unaprijed znamo glavni i sporedni broj uređaja onda se može pojaviti potreba pretvorbe tih dviju cjelobrojnih vrijednosti u tip *dev_t*. To možemo učiniti pomoću makro naredbe: `MKNOD(int major, int minor)`.

6.3.1. Alokacija i oslobađanje glavnog i sporednog broja uređaja

Jedna od prvih operacija koju naš upravljački program mora obaviti prilikom postavljanja *char* tipa uređaja jest da pribavi jedan ili više brojeva uređaja kojima će upravljati. Neophodna funkcija za taj zadataka je `int register_chrdev_region(dev_t first, unsigned int count, char *name)`. Prvi argument funkcije je početni broj uređaja gdje obično sporedni broj ima vrijednost 0, ali to nije obavezno. Drugi argument je ukupan broj glavnih i sporednih brojeva koje zahtijevamo, a argument *name* je ime uređaja koji je povezan s alociranim glavnim i sporednim brojevima. Specificirano ime uređaja se pojavljuje u */proc/devices* direktoriju i *sysfs* datotečnom sustavu. Povratna vrijednost bit će 0 ako je pribavljanje i alokacija odgovarajućih brojeva uspješno prošla, a u slučaju pogreške vraća se negativna vrijednost. Ova funkcija odlično radi ako unaprijed znamo kakav glavni i sporedni broj uređaja tražimo, ali težnja prilikom projektiranja unutar jezgre Linux-a jest da se koristi dinamičko alociranje odgovarajućih brojeva za uređaj(e). U svrhu toga napravljena je funkcija int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name) gdje je razlika u dva argumenta u odnosu na već spomenutu funkciju: prvi argument je izlazni parametar koji će nakon uspješne alokacije sadržavati prvi broj u alociranom rasponu, a drugi predstavlja prvi sporedni broj koji će se koristiti (obično 0). Unutar jezgre operacijskog sustava postoje uobičajeni uređaji kojima su statički dodijeljeni glavni i sporedni brojevi bez dinamičke alokacije; lista takvih uređaja se može naći unutar *Documentation/devices.txt*. Dakle, kao projektanti upravljačkih programa

unutar Linux jezgre možemo izabrati statički način dodijele gdje ćemo izabrati brojeve koji izgledaju da se ne koriste ili ćemo to napraviti na dinamički način. Prvi način može raditi sve dok smo mi jedini korisnici upravljačkog programa, ali ako ga želimo distribuirati pod GPL licencom onda može doći do konflikta u glavnim brojevima kod drugih platformi koje bi koristile naš upravljački program.

Sljedeći isječak kôda iz funkcije `int __init lpc2xxx_can_init(void)` obavlja alociranje glavnog i sporednog broja za CAN uređaj. Ako je vrijednost varijable *candev_major* jednaka 1 onda se brojevi alociraju na statičan način, inače koristi se dinamički način dodijele. Unutar kôda je namješteno da se koristi dinamičko dodjeljivanje glavnog i sporednog broja na CAN uređaj.

```
if (candev_major)
    result = register_chrdev_region(dev,candev_count,"can_lpc2xxx");
else{
    result = alloc_chrdev_region(&dev,0,candev_count,"can_lpc2xxx");
    candev_major = MAJOR(dev);
}
```

Također, bez obzira kako smo alocirali glavne i sporedne brojeve, trebamo ih osloboditi kada ih više ne koristimo. U tu svrhu koristimo funkciju `void unregister_chrdev_region(dev_t first, unsigned int count)`. Uobičajeno mjesto gdje to želimo napraviti je unutar funkcije koja oslobađa i druge resurse koje smo na početku prilikom inicijalizacije zauzeli,, a to je funkcija `void __exit lpc2xxx_can_cleanup(void)` unutar CAN upravljačkog programa. Važno je napomenuti da gornje funkcije alociraju glavne i sporedne brojeve uređaja, ali ne govore jezgri operacijskog sustava o tome koja je naša namjera s tim brojevima. Prije nego korisničke aplikacije mogu pristupiti jednom od ovih brojeva koje smo alocirali, upravljački program mora povezati te brojeve s njegovim unutarnjim funkcijama koje implementiraju operacije CAN upravljačkog programa.

6.3.2. Čvorovi uređaja unutar datotečnog sustava

Čvorovi uređaja su datoteke koje se nalaze unutar */dev* mape (uC)Linux datotečnog sustava i ostvaruju način pristupa upravljačkom programu od strane aplikacije kao da se radi o običnom pristupu datotekama. Dakle, spomenute čvorove možemo zamisliti kao kanale kojima protječu podaci i kontrolne naredbe od korisničkog prostora gdje se izvode razne aplikacije prema upravljačkom programu koji se izvodi u prostoru jezgre. Ako naredbom *ls* pregledamo sadržaj spomenutog direktorija na osobnom računalu[49] uočiti ćemo koncepte o kojima smo govorili u prošlom poglavlju (nazivi uređaja, glavni i sporedni broj itd.).

Prilikom alociranja glavnog i sporednog broja uređaja vrijeme je da se stvori odgovarajući čvor unutar */dev* mape. Ako smo brojeve alocirali na statičan način onda već znamo koje bi vrijednosti trebali imati i stvaranje čvora možemo napraviti pisanjem skripte koja bi se trebala pokrenuti prilikom podizanja operacijskog sustava. Ako smo odredili da glavni broj uređaja bude 150, a sporedni 0 onda bi se stvaranje čvora za *char* uređaj moglo obaviti slijedećom naredbom: `mknod c 150 0` unutar */dev* mape. Budući da je korišten dinamički način dodijele glavnog broja, projektant ne može znati unaprijed koju vrijednost će sadržavati glavni broj pa stoga ne može stvoriti odgovarajući čvor datotečnog sustava prije nego sazna koji glavni broj je jezgra operacijskog sustava dodijelila uređaju. Jedan od načina kako to saznati je pregledavanjem sadržaja */proc/devices* datoteke. Naravno, odgovarajući jezgreni modul, tj. upravljački program mora biti uključen unutar jezgre operacijskog sustava bilo da koristimo *insmod* program ili da je već inicijalno kompajliran s jezgrom. Nakon što smo saznali glavni broj možemo stvoriti odgovarajući čvor gornjom naredbom.

Međutim, što napraviti ako je datotečni sustav *read-only*, tj. ako je namijenjen da se iz njega može samo čitati prilikom izvođenja jezgre operacijskog sustava. Takav datotečni sustav je *romfs* koji je korišten kod korištenog razvojnog sustava LPC2478STK. U takvim slučajevima čvor se mora stvoriti prije kompajliranja u odgovarajućim datotekama izvornog kôda jezgre operacijskog sustava. Kod korištenog razvojnog sustava ta datoteka se nalazi unutar *vendors/NXP/LPC2468* pod nazivom *Makefile*. U nastavku je dan odsječak kôda spomenute datoteke koji

[49] Ako imate razvojni sustav pokraj sebe, onda to možete učiniti i na uCLinux operacijskom sustavu

prikazuje kako se dodaje čvor uređaja unutar */dev* uređaja. Čitatelj se upućuje da pregleda cijelu datoteku i pokuša ju razumjeti jer je to način kako napraviti *romfs* datotečni sustav unutar Linux jezgre operacijskog sustava.

```
ROMFSIMG = $(IMAGEDIR)/romfs_5.img
IMAGE    = $(IMAGEDIR)/vmlinux.bin
ELFIMAGE = $(IMAGEDIR)/image.elf

ROMFS_DIRS = bin dev etc home lib mnt proc usr var

DEVICES = \
        tty,c,5,0       console,c,5,1       cua0,c,5,64       cua1,c,5,65   \
        \
        mem,c,1,1       kmem,c,1,2          null,c,1,3        ram0,b,1,0 \
        ram1,b,1,1 \
        \
        ptyp0,c,2,0     ptyp1,c,2,1         ptyp2,c,2,2       ptyp3,c,2,3 \
        ptyp4,c,2,4     ptyp5,c,2,5         ptyp6,c,2,6       ptyp7,c,2,7 \
        ptyp8,c,2,8     ptyp9,c,2,9         ptypa,c,2,10      ptypb,c,2,11 \
        ptypc,c,2,12    ptypd,c,2,13        ptype,c,2,14      ptypf,c,2,15 \
        \
        rom0,b,31,0     rom1,b,31,1         rom2,b,31,2       rom3,b,31,3 \
        rom4,b,31,4     rom5,b,31,5         rom6,b,31,6       rom7,b,31,7 \
        rom8,b,31,8     rom9,b,31,9 \
        \
        tty0,c,4,0      tty1,c,4,1          tty2,c,4,2        tty3,c,4,3 \
        ttyS0,c,4,64    ttyS1,c,4,65        ttyS2,c,4,66      ttyS3,c,4,67
\
        \
        ttyp0,c,3,0     ttyp1,c,3,1         ttyp2,c,3,2       ttyp3,c,3,3 \
        ttyp4,c,3,4     ttyp5,c,3,5         ttyp6,c,3,6       ttyp7,c,3,7 \
        ttyp8,c,3,8     ttyp9,c,3,9         ttypa,c,3,10      ttypb,c,3,11 \
        ttypc,c,3,12    ttypd,c,3,13        ttype,c,3,14      ttypf,c,3,15 \
        \
        zero,c,1,5        random,c,1,8      urandom,c,1,9 \
        mtd0,c,90,0       mtd1,c,90,1       mtd2,c,90,2 \
        mtdblock0,b,31,0 mtdblock1,b,31,1 mtdblock2,b,31,2 \
        fb0,c,29,0      fb1,c,29,1         can0,c,254,0
```

Neka čitatelj primijeti zadnju stavku koja predstavlja čvor za CAN uređaj koji će se koristiti unutar upravljačkog programa.

6.3.3. Važne strukture podataka

Većina osnovnih operacija koje mora izvršiti upravljački program uključuje važne strukture podataka Linux jezgre operacijskog sustava kao što su *file_operations* i *file* strukture podataka. U ovom poglavlju ćemo se upoznati površno s ovim strukturama kako bi bili u mogućnosti razumjeti osnovne koncepte upravljačkih programa koje ćemo objasniti u nastavku ovog poglavlja.

6.3.3.1. *file* struktura podataka

Prije sve moramo naglasiti da *file* nema nikakve veze s FILE pokazivačem koji se koristi u korisničkim aplikacijama. Bolje rečeno, FILE je definiran u korisničkim C bibliotekama i nikada se ne pojavljuje u izvornom kôdu jezgre jer jezgra ne koristi standardne C biblioteke. S druge strane, *struct file* je struktura podataka koja se nikada ne pojavljuje u korisničkim aplikacijama.

Struktura *struct file* predstavlja otvorenu datoteku i kao takva ne upotrebljava se samo u upravljačkim programima. Spomenuta struktura se stvara prilikom poziva metode *open* i prenosi se kao argument funkcijama koje upravljaju sadržajem datoteke. Pozivom funkcije *close* jezgra operacijskog sustav oslobađa spomenutu strukturu podataka. U izvornom kôdu jezgre operacijskog sustava *struct file* se obično naziva *file* ili *filp*[50].

Najvažnija polja strukture *struct file* su prikazana u nastavku:

- *mode_t f_mode*: tip datoteke određuje da li se datoteku može čitati, pisati ili oboje. Informaciju koju pruža ovo polje možemo iskoristiti kako bi provjerili kakva je dozvola za pisanje ili čitanje odabrane datoteke unutar *open* ili *ioctl* funkcije. Važno je uočiti da ne trebamo provjeravati dozvolu prilikom poziva funkcija *read* ili *write* jer jezgra operacijskog sustava odbacuje bilo kakav pokušaj nedozvoljene operacije nad otvorenom datotekom.
- *loff_t f_pos*: Trenutna pozicija čitanja iz datoteke ili pisanja u datoteku. Upravljački program može pročitati ovu vrijednost da bi saznao trenutnu poziciju kursora unutar datoteci, ali ju ne bi smio promijeniti. Jedine dvije funkcije koje mijenjaju poziciju kursora unutar datoteke su *read* i *write*

[50] engl. file pointer

preko zadnjeg argumenta kojeg primaju prilikom poziva umjesto da se direktno modificira vrijednost *filp->f_pos*. Jedina iznimka je funkcija *llseek* koja ima funkciju da promijeni trenutnu poziciju kursora.

- *unsigned int f_flags*: ovo polje su zastavice koje odražavaju trenutno stanje datoteke (O_RDONLY, O_NONBLOCK, O_SYNC…). Upravljački program često koristi zastavicu O_NONBLOCK koju ćemo objasniti nešto kasnije. Sve zastavice su definirane unutar <*linux/fcntl.h*>.
- *struct file_operations *f_op*: ova struktura predstavlja operacije koje se mogu izvršiti nad datotekom. Jezgra inicijalizira pokazivač na spomenutu strukturu unutar *open* funkcije i onda ga čita kada treba pozvati određenu operaciju koju zahtjeva odgovarajući sistemski poziv. Zanimljivo je da se ovaj pokazivač može usmjeriti prilikom otvaranja nove datoteke na druge strukture istog tipa jer jezgra ne pamti prošlu vrijednost pokazivača prilikom ponovnog pokretanja neke aplikacije. Primjerice, *open* metoda koja se povezuje sa glavnim brojem 1 zamjenjuje operacije nad datotekom u *filp->f_op* ovisno o tome koji je sporedni broj uređaja. Ova struktura će detaljnije biti opisana u sljedećem poglavlju.
- *void *private_data*: *open* sistemski poziv postavlja ovaj pokazivač na NULL prije pozivanja *open* funkcije upravljačkog programa. Programer može koristiti ovo polje kako on želi ili ga može jednostavno ignorirati. Ako koristite ovaj pokazivač za alocirane podatke unutar memorije onda nesmijete zaboraviti osloboditi tu memoriju u *candev_release* metodi prije nego jezgra operacijskog sustava uništi *file* strukturu.

Stvarna struktura ima još nekoliko polja, ali nisu relevantni za pisanje upravljačkih programa jer upravljački program nikada ne stvara *file* strukture; one su uvijek stvorene izvan upravljačkog programa.

6.3.3.2. *file_operations* struktura podataka

Do sada smo rezervirali glavne i sporedne brojeve uređaja, ali još nismo povezali funkcije upravljačkog programa s rezerviranim brojevima. Upravo *file_operations* struktura predstavlja način kako *char* upravljački program povezuje svoje funkcije s rezerviranim brojevima. Ova struktura je definirana unutar <*linux/fs.h*> i predstavlja jednostavnu kolekciju pokazivača na funkcije. Svaka otvorena datoteka koja je predstavljena strukturom *file* povezana je sa setom

funkcija koje se nad otvorenom datotekom mogu pozvati. To je omogućeno tako što unutar strukture *file* postoji polje naziva *f_op* koje pokazuje na odgovarajuću strukturu *file_operations*. Funkcije koje se pozivaju nad datotekom implementiraju sistemske pozive pa su stoga nazvane *open*, *read*, *write*, *close* itd. Koristeći objektno-orijentiranu paradigmu datoteku možemo smatrati objektom, a funkcije koje se pozivaju nad datotekom objektnim metodama. Pokazivač na strukturu *file_operations* se dogovorno naziva *fops*. Svako polje unutar strukture mora pokazivati na funkciju koju upravljački program implementira radi ostvarenja specifične funkcionalnosti ili na NULL ako dotična operacija nije podržana nad otvorenom datotekom, tj. uređajem.

Sljedeći prikaz polja *file_operations* strukture navodi najčešće operacije koje aplikacija može pozvati nad uređajem, tj. nad datotekom koja predstavlja čvor unutar datotečnog sustava između aplikacije i upravljačkog programa:

- *struct module *owner*: prvo polje strukture uopće ne predstavlja nikakvu operaciju nego pokazivač na modul koji posjeduje dotičnu strukturu. Ovo polje sprječava da se modul isključi iz jezgre, tj. da se oslobode njegovi resursi dok se njegove operacije koriste od aplikacija putem sistemskih poziva.
- *ssize_t (*read)(struct file *, char __user *, size_t, loff_t *)*: ovo je pokazivač na funkciju koja pribavlja podatke iz uređaja kojim upravlja neki upravljački program. Ako ovaj pokazivač inicijaliziramo na NULL znači da operacija čitanja neće biti podržana što uzrokuje da sistemski poziv vrati negativnu vrijednost –EINVAL (engl. invalid argument). Ako povratna vrijednost nije negativna onda ona predstavlja broj okteta koji su uspješno pročitani.
- *ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *)*: ovo je pokazivač na funkciju koja omogućava slanje podataka uređaju, tj. omogućava pisanje u uređaj. Ako je ovo polje NULL onda se vraća negativna vrijednost –EINVAL programu koji je pozvao *write* sistemski poziv. Ako je povratna vrijednost pozitivna onda predstavlja broj okteta koji su uspješno upisani.
- *unsigned int (*poll)(struct file *, struct poll_table_struct *)*: kada korisnička aplikacija pozove funkciju *select*, *poll* ili *epoll* onda se pozove funkcija

upravljačkog programa čija je adresa zapisana unutar ovog pokazivača. Ta funkcija daje povratnu vrijednost kao masku bitova koji pokazuju da li se iz uređaja može čitati ili pisati bez da proces mora biti blokiran u slučaju da se tražena operacija ne može izvršiti. Također, prosljeđuje informacije jezgri operacijskog sustava kako bi bila u mogućnosti staviti proces u čekanje dok tražena operacija ne postane dostupna.

- *int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long)*: sistemski poziv pod imenom *ioctl* omogućava da korisnička aplikacija uputi uređaju specifične naredbe preko upravljačkog programa. Postoje operacije nad uređajem koje ne spadaju u grupu pisanja ni čitanja, stoga takve operacije svrstavamo u ovaj sistemski poziv. Primjerice, postavljanje bitrate-a na kojem će raditi CAN modul predstavlja takvu jednu operaciju. Ako programer nije implementirao ovu metodu sistemski poziv vraća negativnu povratnu vrijednost, tj. –ENOTTY (engl. no such ioctl for device).
- *int (*open)(struct inode *, struct file *)*: ova operacija omogućava da upravljački program dobije obavijest da je aplikacija pokušala otvoriti uređaj, tj. čvor unutar datotečnog sustava. Ona ne mora biti implementirana, tj. ovo polje strukture *file_operations* može biti NULL, ali upravljački program neće dobiti potonju obavijest.
- *int (*release)(struct inode *, struct file *)*: ova funkcija se poziva onda kada se struktura *file* oslobodi, tj. kada aplikacije pozove funkciju *close*. Međutim, treba napomenu da se ova metoda ne poziva svaki put kada se pozove *close* od strane neke aplikacije. Gdje god unutar jezgre postoji kopija trenutno korištene *file* strukture, *release* neće biti pozvana sve dok se ne zatvore sve kopije funkcijom *close*.

Unutar CAN upravljačkog programa implementirane su sljedeće operacije:

```
struct file_operations candev_fops = {
        .owner    = THIS_MODULE,
        .open     = candev_open,
        .read     = candev_read,
        .write    = candev_write,
        .ioctl    = candev_ioctl,
        .release = candev_release,
        .poll     = candev_poll,
};
```

6.3.4. Registracija *char* uređaja

Jezgra koristi strukturu tipa *struct cdev* kako bi reprezentirala uređaj ostalim dijelovima operacijskog sustava. Prije nego jezgra pozove funkcije upravljačkog programa moramo alocirati i registrirati najmanje jednu ovakvu strukturu. Da bi smo bili u mogućnosti obaviti tražene operacije nad *struct cdev* moramo uključiti zaglavnu datoteku <*linux/cdev.h*> gdje je definirana struktura i ostale pomoćne funkcije.

Postoje dva načina kako alocirati i inicijalizirati *cdev* strukturu. Ako želite rezervirati *cdev* strukturu u toku izvođenja možete to učiniti slijedećim odsječkom kôda:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

Međutim, često ta struktura predstavlja polje u našoj vlastitoj strukturi koja objedinjuje i neka druga polja specifična za odgovarajući uređaj ili jednostavno predstavlja globalnu varijablu koju vide sve funkcije unutar izvornog kôda našeg upravljačkog programa. U tom slučaju strukturu možemo alocirati koristeći razne funkcije za alociranje memorije kao što je prikazano u sljedećem odsječku za CAN upravljački program koji se nalazi unutar *lpc2xxx_can_init* funkcije:

```
/* candev_count = 1, jer koristimo jedan CAN modul */
canchardev = kmalloc(sizeof(struct cdev)*candev_count,GFP_KERNEL);
    if (!canchardev) {
        printk(KERN_WARNING "\tNe mogu alocirati memoriju za cdev\n");
        result = -1;
        goto err_kmalloc_cdev;
    }
```

Gdje je *canchardev* globalna *cdev* struktura. Alociranje memorije obavljamo pomoću funkcije *kmalloc* koja je dosta slična *malloc* funkciji. Ta funkcija samo rezervira memoriju dok ju ne inicijalizira. Prvi argument *kmalloc* funkcije je veličina bloka memorije koji se rezervira, a drugi argument predstavlja zastavicu koja kontrolira ponašanje funkcije. U ovom slučaju zastavica GFP_KERNEL znači da *kmalloc* funkcija može staviti proces u stanje spavanja u slučaju da nema dovoljno memorije u trenutku rezerviranja.[51] Nadalje, nakon što smo alocirali memoriju za *cdev* strukturu potrebno ju je inicijalizirati pomoću funkcije: *cdev_init*. Nakon inicijalizacije strukture trebamo inicijalizirati polje *owner* i *ops*. Zadnji korak je poziv

[51] Za druge zastavice molimo pogledajte literaturu [1]

funkcije *cdev_add* koja treba obavijestiti jezgru operacijskog sustava da je struktura *cdev*, koju prenosimo kao prvi argument te funkcije, upravo inicijalizirana. Drugi argument predstavlja prvi broj uređaja (glavni i sporedni) preko kojeg se može pristupiti uređaju, a treći argument je ukupan broj glavnih i sporednih brojeva koji bi se trebali povezivati s uređajem.

Sve navedeno predstavljeno je funkcijom koja je navedena u nastavku:

```
void can_setup_cdev(struct cdev *dev, int minor,
                struct file_operations *fops)
{
        int err, devno = MKDEV(candev_major, minor);
        cdev_init(dev, fops);
        dev->owner = THIS_MODULE;
        dev->ops = fops;
        err = cdev_add (dev, devno, 1);
        if (err)
                printk (KERN_NOTICE "\tGreska %d u dodavanju
can_lpc2xxx%d\n", err, minor);
}
```

Također, ako želimo isključiti *char* uređaj iz sustava pozivamo funkciju *cdev_del* koja ima sljedeći prototip:

```
void cdev_del(struct cdev *dev);
```

Ova funkcija se unutar CAN upravljačkog programa poziva unutar funkcije *lpc2xxx_can_cleanup*.

6.3.5. Funkcija *open*

Unutar funkcije *open* upravljački program obavlja osnovnu inicijalizaciju koja služi kao priprema za ostale operacije koje će uslijediti kasnije. U većini upravljačkih programa, *open* funkcija treba moći obavljati sljedeće zadatke:

- provjeriti specifične greške unutar uređaja (npr. „uređaj nije spreman“ ili „uređaj se već koristi“)
- inicijalizirati uređaj ukoliko se otvara po prvi puta
- ako je potrebno ponovno inicijalizirati polje *f_op*
- alocirati i inicijalizirati bilo kakvu strukturu podataka koja će biti dostupna preko *filp->private_data*

Funkcija *open* CAN upravljačkog programa je dana u nastavku:

```
int candev_open (struct inode *inode, struct file *filp)
{
    int dev = iminor(inode);
    filp->private_data = (void*)iminor(inode);

    if (!atomic_dec_and_test(&candev[dev].can_available)) {
        atomic_inc(&candev[dev].can_available);
        return -EBUSY; /* GRESKA: uredjaj je vec otvoren */
    }
    candev[dev].rfinish = 0;
    candev[dev].srr_bit = 0;
lpc2xxx_can_setup(dev,CAN_OPERATING_MODE,LPC2000_CANDRIVER_CANBITRATE100K
28_8MHZ);
    return 0;
}
```

Unutar *inode* strukture podataka postoji polje *struct cdev *i_cdev* koje sadrži pokazivač na odgovarajuću *cdev* strukturu unutar upravljačkog programa. Drugi argument je pokazivač na odgovarajuću strukturu tipa *struct file*. Unutar *open* metode odlučili smo se da ćemo unutar *filp->private_data* čuvati sporedni broj uređaja. Naime, unutar mikrokontrolera LPC2478 postoje dva CAN modula – CAN0 i CAN1. Budući da je na razvojnog sustavu sklopovski izveden jedino CAN0 modul odlučili smo se staviti njegov sporedni broj na vrijednost 0 i to prenositi preko privatnog polja *file* strukture podataka. Kada bi koristili CAN1 modul sporedni broj bi imao vrijednost 1. Varijabla *atomic_t can_available* predstavlja cjelobrojnu atomsku varijablu kojoj se vrijednost može mijenjati samo preko funkcija koje su navedene unutar <*asm/atomic.h*>. Atomska operacija se odnosi na skup operacija koje se moraju izvršiti bez bilo kakvog oblika prekida tog izvođenja. Spomenuta varijabla služi da se osigura jedinstveno korištenje CAN modula unutar jezgre operacijskog sustava. Prilikom uključivanja modula upravljačkog programa u jezgru, tj. prilikom poziva funkcije *lpc2xxx_can_init* ova varijabla se inicijalizira na vrijednost 1. Nakon provjere zauzetosti CAN modula resetiramo zastavice (o njima će više riječi biti kasnije) i postavimo parametre CAN modula.

6.3.6. Funkcija *release*

Ova funkcija je reverzna prethodnoj funkciji, tj. trebala bi obaviti sljedeće zadatke:

- Oslobodi sve resurse koje je *open* funkcija alocirala, posebice unutar *filp->private_data* polju
- Onemogućiti sklopovske prekide modula, tj. isključiti modul iz interakcije s upravljačkim programom

U nastavku je dan odsječak kôda funkcije koja obavlja traženu funkcionalnosti.

```
int candev_release(struct inode *inode, struct file *filp)
{
    int dev = iminor(inode);

lpc2xxx_can_setup(dev,CAN_RESET_MODE,LPC2000_CANDRIVER_CANBITRATE100K28_8
MHZ);
    if (dev == 0)
        CAN1IER = 0; /* onemogucavamo prekide */
    if (dev == 1)
        CAN2IER = 0;

    (candev[dev].fifo).count = 0; /* resetiramo zastavice */
    (candev[dev].fifo).wp = 0;
    (candev[dev].fifo).rp = 0;
    candev[dev].rfinish = 1;
    wake_up_interruptible(&candev[dev].fifoque);
    atomic_inc(&candev[dev].can_available); /* oslobadjamo uredjaj */
    return 0;
}
```

Oprezni čitatelj može se upitati o tome kako upravljački program zna kada se otvoreni uređaj (ili datoteka) zaista zatvorio? Odgovor je jednostavan: funkcija *release* se ne poziva prilikom svakog izvršavanja *close* sistemskog poziva. Jezgra čuva brojač koji broji koliko se puta koristila struktura *file*. Dakle, *close* sistemski poziv će izvršiti funkciju *release* samo onda kada brojač *file* strukture padne na 0. Ova veza između funkcije *release* i *close* sistemskog poziva garantira da upravljački program vidi samo jedan poziv funkcije *release* za svaki poziv funkcije *open*.

6.3.7. Funkcije *read* i *write*

Ove dvije funkcije obavljaju slične zadaće: kopiranje podataka iz aplikacije prema upravljačkom programu i obrnuto. Iz tog razloga njihovi prototipovi su prilično slični (vidi poglavlje 5.3.3.2). Za obje funkcije vrijedi da je *filp* pokazivač na datoteku, a *count* je količina podataka koji se prenose. Argument *buff* pokazuje na

korisničke podatke koji trebaju biti upisani u upravljački program ili pokazuje na prazan spremnik podataka gdje će tek pročitani podaci biti smješteni. Bitno je naglasiti da je *buff* argument pokazivač unutar korisničkog prostora adresa i stoga ne može direktno biti korišten unutar jezgrinog kôda. Postoji više razloga tomu, a jedan od glavnih razloga jest što je memorija unutar korisničkog prostora podijeljena u tzv. stranice pa stoga memorija kojoj želimo pristupiti za vrijeme sistemskog poziva uopće ne mora biti unutar RAM memorije. Sigurnosni razlozi su također jako važni jer pokazivač koji se nalazi u korisničkom prostoru adresa izvodi se unutar korisničke aplikacije koja može biti prepravljena programerskim pogreškama pa ako bi upravljački program na „slijepo“ koristio takav pokazivač omogućio bi korisničkoj aplikaciji da pristupi memoriji bilo gdje unutar jezgre operacijskog sustava. S druge strane, očito je da upravljački program mora moći pristupiti spremnicima podataka (engl. *buffer*) unutar korisničkog prostora. Ovaj pristup mora biti obavljen koristeći posebne funkcije koje pruža jezgra operacijskog sustava. Ove funkcije će osigurati potrebnu sigurnost jezgrinog kôda. Unutar ovog poglavlja navesti ćemo samo osnovne funkcije koje su definirane unutar <*asm/uaccess.h*>. Dakle, *read* i *write* funkcije unutar CAN upravljačkog programa moraju pristupiti cijelom bloku podataka unutar korisničkog prostora. Takva potreba je omogućena sljedećim funkcijama koje kopiraju proizvoljnu količinu podataka iz korisničkog prostora ili u korisnički prostor (aplikaciju) i predstavljaju osnovu za implementaciju *read* i *write* funkcija upravljačkog programa:

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);

unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
```

Dakle, pored kopiranja podataka ove funkcije provjeravaju da li je pokazivač unutar korisničkog prostora validan. Ako pokazivač nije validan, kopiranje podataka se neće izvršiti; također, ako se tijekom kopiranja pojavi pogrešna adresa odmah se prekida kopiranje. U jednom i drugom slučaju povratna vrijednost je količina podataka koji se još trebaju kopirati. Koliku god količinu podataka funkcije *read* i *write* prenosile trebaju na kraju modificirati vrijednost zadnjeg argumenta *loff_t *f_pos* koji predstavlja poziciju kursora unutar datoteke.

Ukoliko dođe do pogreške te dvije metode vraćaju negativnu vrijednost. Nenegativna povratna vrijednost govori pozivnom programu (aplikaciji) koliko okteta podataka je uspješno zapisano ili pročitano. Ako se u toku pisanja ili čitanja dogodi pogreška povratna vrijednost mora biti broj uspješno prenesenih okteta podataka. Iako funkcije koje su dio jezgrenog kôda vraćaju negativnu vrijednost kako bi signalizirale pogrešku, aplikacija koja se izvodi u korisničkom prostoru vidi -1 kao povratku vrijednost. Ako korisnik želi doznati koja se točno pogreška dogodila mora pristupiti *errno* varijabli.

Povratna vrijednost za funkciju *read* interpretira se na sljedeći način:

- Ako je povratna vrijednost jednaka *count* argumentu *read* sistemskog poziva onda to znači da su svi podaci pročitani. To je najbolji slučaj.
- Ako je povratna vrijednost pozitivna i manja od argumenta *count* to znači da je pročitan samo dio podataka.
- Ako je povratna vrijednost točno jednaka 0 onda nije pročitaj niti jedan podatak jer je dosegnut kraj datoteke (engl. *end-of-file*)
- Ukoliko je povratna vrijednost negativna tada se dogodila pogreška. Povratna vrijednost specificira koja se pogreška točno dogodila. Opis pojedine vrijednosti pogreške možemo pogledati unutar <*linux/errno.h*>.

```
ssize_t candev_read(struct file *filp, char __user *buf, size_t count,
loff_t *f_pos)
{
    ssize_t retval = 0;
    int dev;
    int *fcount;
    int toread;
    unsigned int pbuf;
    struct can_lpc2000_message *pmsg, msg;

    dev = (int)filp->private_data;
    if (count == 0 || count%sizeof(struct can_lpc2000_message) != 0 ) {
       retval = -EINVAL;
       goto out;
    }

    spin_lock(&candev[dev].slock);
    fcount = &(candev[dev].fifo).count;
    spin_unlock(&candev[dev].slock);
    if (wait_event_interruptible(candev[dev].rque, *fcount > 0) == -
ERESTARTSYS)
    {
       retval = -ERESTARTSYS;
       goto out;
    }
```

```
    toread = count/sizeof(struct can_lpc2000_message) < *fcount ?
       count/sizeof(struct can_lpc2000_message) : *fcount;
    retval = toread;

    for (pbuf = 0; toread > 0; toread--)
    {
       spin_lock(&candev[dev].slock);
       pmsg = fifo_remove(&candev[dev].fifo);
       if (!pmsg)
           break;
       msg.FI = pmsg->FI;
       msg.ID = pmsg->ID;
       msg.DA = pmsg->DA;
       msg.DB = pmsg->DB;
       spin_unlock(&candev[dev].slock);

       if (copy_to_user (&buf[pbuf], &msg, sizeof(struct
can_lpc2000_message) )) {
           retval = -EFAULT;
           goto out;
       }

       pbuf += sizeof(struct can_lpc2000_message);
    }

    *f_pos = (candev[dev].fifo).rp;

out:
    candev[dev].rfinish = 1;
    wake_up_interruptible(&candev[dev].fifoque);
    return retval;
}
```

U početku pročitamo privatno polje *file* strukture koje je prije inicijalizirano sporednim brojem koji određuje broj modula – CAN0 ili CAN1. U našem slučaju koristiti ćemo CAN0 jer jedino njega možemo testirati na razvojnom sustavu. Nakon toga provjeravamo da li ima podataka unutar cirkularnog fifo spremnika koji se mogu pročitati. U slučaju da podataka nema unutar spremnika proces ulazi u red čekanja pomoću funkcije *wait_event_interruptible*[52] dok ne dođe podatak unutar cirkularnog fifo spremnika. Kada CAN primopredajnik razvojnog sustava primi poruku onda se generira prekid koji obrađuje upravljački program. Unutar prekida primljena poruka se sprema u cirkularni fifo spremnik i funkcija *read* može nastaviti dalje. Nakon toga čitamo određeni broj poruka i kopiramo ih u korisnički prostor, tj. šaljemo ih aplikaciji, modificiramo *f_pos* i vraćamo odgovarajuću povratnu vrijednost.

[52] Redove za čekanje odgovarajućeg događaja ćemo objasniti nešto kasnije

Povratna vrijednost za funkciju *write* interpretira se na sljedeći način:

- Ako je povratna vrijednost jednaka argumentu *count* svi podaci su pročitani.
- Ako je vrijednost pozitivna, ali manja od argumenta *count* to znači da je samo dio podataka prenesen u upravljački program.
- Ako je povratna vrijednost točno jednaka 0 onda ništa nije preneseno u upravljački program. Važno je pripomenuti da ovo nije pogreška i ne treba se tako smatrati.
- Negativna vrijednost označuje pogrešku. Značenje pogreške možemo interpretirati pomoću zaglavne datoteke <*linux/errno.h*>.

U nastavku je dan kôd funkcije koja odgovara na sistemski poziv *write* kod CAN upravljačkog programa. Budući da CAN modul mikrokontrolera LPC2478 ima tri spremnika za slanje poruka onda najprije provjeravao koji je od njih slobodan za upotrebu. Nakon toga kopiramo podatke iz korisničkog prostora pomoću funkcije *copy_from_user* i šaljemo poruku pomoću jednog od slobodnih spremnika za slanje. Važno je pripomenuti da nemamo izlazni cirkularni fifo spremnik koji bi primao poruke iz korisničkog prostora dok upravljački program nije spreman za slanje preko CAN sabirnice. Iz tog razloga uvijek vraćamo veličinu jedne poruke jer upisujemo uvijek jednu poruku po svakom pozivu *write* funkcije. Iako poruke mogu biti izgubljene ako se primaju dok ne postoji ulazni fifo spremnik; kod izlaznog spremnika nema gubitka poruka jer one ostaju u spremniku koji je definiran unutar korisničke aplikacije. Ipak, to može biti jedno poboljšanje za CAN upravljački program jer se smanjuje broj prijenosa podataka između korisničkog i jezgrenog prostora - izlazni cirkularni fifo spremnik poruka.

```
ssize_t candev_write(struct file *filp, const char __user *buf, size_t
count, loff_t *f_pos)
{
    ssize_t retval;
    unsigned int tbuf = -1;
    struct can_lpc2000_message msg;
    int dev;

    if (count != sizeof(struct can_lpc2000_message)) {
        retval = -EINVAL;
        goto out;
    }

    dev = (int)filp->private_data;
    switch (dev) {
```

```
        case 0:
            if ( CAN1SR & CANSR_TBS1 )
                tbuf = CANCMR_STB1;
            if ( CAN1SR & CANSR_TBS2 )
                tbuf = CANCMR_STB2;
            if ( CAN1SR & CANSR_TBS3 )
                tbuf = CANCMR_STB3;
            break;
        case 1:
            if ( CAN2SR & CANSR_TBS1 )
                tbuf = CANCMR_STB1;
            if ( CAN2SR & CANSR_TBS2 )
                tbuf = CANCMR_STB2;
            if ( CAN2SR & CANSR_TBS3 )
                tbuf = CANCMR_STB3;
            break;
    }

    if (tbuf == -1) {
        retval = -EBUSY;
        goto out;
    }

    if (copy_from_user (&msg, buf, sizeof(struct can_lpc2000_message))) {
        retval = -EFAULT;
        goto out;
    }

    switch (dev)
    {
        case 0:
            switch (tbuf)
            {
                case CANCMR_STB1:
                    CAN1TFI1 = msg.FI;
                    CAN1TID1 = msg.ID;
                    CAN1TDA1 = msg.DA;
                    CAN1TDB1 = msg.DB;
                    break;
                case CANCMR_STB2:
                    CAN1TFI2 = msg.FI;
                    CAN1TID2 = msg.ID;
                    CAN1TDA2 = msg.DA;
                    CAN1TDB2 = msg.DB;
                    break;
                case CANCMR_STB3:
                    CAN1TFI3 = msg.FI;
                    CAN1TID3 = msg.ID;
                    CAN1TDA3 = msg.DA;
                    CAN1TDB3 = msg.DB;
                    break;
            }
            if (candev[dev].srr_bit > 0)
                CAN1CMR = CANCMR_SRR | tbuf;
            else
                 CAN1CMR = CANCMR_TR | tbuf;
            break;
        case 1:
            switch (tbuf)
            {
                case CANCMR_STB1:
```

```
                    CAN2TFI1 = msg.FI;
                    CAN2TID1 = msg.ID;
                    CAN2TDA1 = msg.DA;
                    CAN2TDB1 = msg.DB;
                    break;
                case CANCMR_STB2:
                    CAN2TFI2 = msg.FI;
                    CAN2TID2 = msg.ID;
                    CAN2TDA2 = msg.DA;
                    CAN2TDB2 = msg.DB;
                    break;
                case CANCMR_STB3:
                    CAN2TFI3 = msg.FI;
                    CAN2TID3 = msg.ID;
                    CAN2TDA3 = msg.DA;
                    CAN2TDB3 = msg.DB;
                    break;
            }
            if (candev[dev].srr_bit > 0)
                CAN2CMR = CANCMR_SRR | tbuf;
            else
                CAN2CMR = CANCMR_TR | tbuf;
            break;
    }

    retval = sizeof(struct can_lpc2000_message);

out:
    *f_pos = tbuf;
    return retval;
}
```

6.3.8. Konkurentnost

Prilikom projektiranja upravljačkih programa moramo uvijek imati na umu da jezgreni kôd podržava istovremeno izvršavanje više različitih programa koji mogu koristiti jednake resurse. Kao jedan od primjera je posluživanje sklopovskih prekida. S druge strane, SMP sustavi mogu izvoditi određeni kôd istovremeno na različitim procesorima. Unutar operacijskog sustava (uC)Linux postoje brojni izvori konkurentnosti koji mogu pristupiti kôdu kojeg razvijamo na iznenađujuće načine. Pogreške koje su povezane s konkurentnošću je najlakše napraviti, ali ih je najteže pronaći i ispraviti. Uvijek treba imati na umu da naš upravljački program kojeg razvijamo može u bilo kojem trenutku izgubiti procesor, a proces koji se u tom trenutku izvodi umjesto upravljačkog programa ili nekog njegovog dijela može se također izvoditi unutar upravljačkog programa kojeg razvijamo. Jezgra Linux operacijskog sustava također omogućava izvođenje nekog kôda nakon određenog vremena pomoću različitih vremenskih modula jezgre ili *workqueue* redova koji osiguravaju izvođenje odabrane funkcije u „bliskoj“ budućnosti. Svaki od navedenih može predstavljati izvor konkurentnosti unutar jezgrinog kôda, tj. može prekinuti normalno izvođenje upravljačkog programa u nezgodnim situacijama. Programer mora predvidjeti ovakve nezgodne situacije i upotrijebiti neki od alata koje pruža jezgra operacijskog sustava kako bi mogao kontrolirati takve situacije.

Kada dva procesa pristupaju dijeljenim resursima operacijskog sustava uvijek postoji mogućnost da dođe do pogrešaka koje su uzrokovane problemom istovremenosti i mogućnosti korupcije podataka koji se dijele. Dakle, prvo pravilo bi bilo da uvijek izbjegavamo dijeljene resurse unutar jezgrinog kôda. Odmah na pamet pada korištenje globalnih varijabli pa stoga imajte pravi razlog zašto ih morate koristiti. Globalne varijable nisu jedini problem; kada god prosljeđujemo pokazivač na neki drugi dio jezgrenog kôda dovodimo taj dio kôda u mogućnost da bude korišten od više procesa. Dakle, kada god imamo dijeljene resurse između više procesa moramo osigurati eksplicitni pristup tim resursima. Tehnika upravljanja pristupom nekom dijelu kôda se često naziva **međusobno isključenje** (engl. *mutual exclusion*) ili **zaključavanje** (engl. *locking*).

6.3.8.1. Spinlock

Kako bi se riješio problem konkurentnog pristupa određenim resursima unutar CAN upravljačkog programa korišten je jezgreni alat pod nazivom *spinlock*. On je jednostavan u svom osnovnom konceptu jer ima samo dva stanja: **zatvoren** (engl. *lock*) ili **otvoren** (engl. *unlock*), i obično je implementiran kao jedan bit u cjelobrojnoj vrijednosti varijable. Proces koji želi preuzeti kritični dio kôda provjerava odgovarajući bit te ako je pristup dozvoljen, tj. ako je *spinlock* otvoren onda proces može preuzeti izvođenje kritičnog bloka kôda. Ako je *spinlock* zatvoren, tj. ako drugi proces izvodi kritičnu sekciju kôda, onda potonji proces ulazi u petlju u kojoj konstantno provjerava da li je *spinlock* postao otvoren za pristup. Upravo ova petlja opravdava upotrebu engleske riječi *spin* (okretanje, rotacija) u nazivu ovoga alata. Naravno, stvarna implementacija je malo kompleksnija, ali je princip isti. Također, bitno je naglasiti da operacija provjere otvorenosti *spinlock*-a mora biti atomska operacija tako da samo jedan proces može dobiti dozvolu korištenja kritičnog dijela kôda. Ovaj alat se najviše koristi na sustavima s više procesora, ali sustavi s jednim procesorom na kojima se izvršava jezgra koja podržava istovremeno izvršavanje više procesa ponašaju se kao i višeprocesorski sustavi u smislu konkurentnosti.

Ako želimo koristiti ovaj alat prilikom programiranja potrebno je uključiti zaglavlje <*linux/spinlock.h*>. Unutar CAN upravljačkog programa *spinlock* je deklariran unutar strukture koja predstavlja tip CAN uređaja (*struct candevi_t*) pod identifikatorom *slock*. Inicijalizacija ovog alata se može napraviti za vrijeme kompajliranja:

```
spinlock_t slock = SPIN_LOCK_UNLOCKED;
```

ili za vrijeme izvođenja:

```
void spin_lock_init(spinlock_t *lock);
```

Provjera zauzetosti i zatvaranje *spinlock*-a obavlja se funkcijom:

```
void spin_lock(spinlock_t *lock);
```

Kako bi otvorili *spinlock* koristimo funkciju:

```
void spin_unlock(spinlock_t *lock);
```

Vrlo je važno da proces koji je zatvorio *spinlock* izvodi kritični kôd sastavljen od atomskih operacija. Ako operacije nisu atomske to znači da bilo koji proces većeg prioriteta od procesa koji je zatvorio *spinlock* može prekinuti izvođenje unutar kritične sekcije kôda i preuzeti procesorsko vrijeme. To dalje znači da proces koji je zatvorio *spinlock* nikada neće otvoriti isti, tj. nikada neće dosegnuti funkciju *spin_unlock*. Iz upravo spomenutih razloga veoma je bitno da kôd unutar kritične sekcije bude sastavljen od atomskih operacija i da ne ulazi u stanje spavanja. Izbjegavanje stanja spavanja dok je *spinlock* zatvoren može biti jako težak zadatak jer brojne funkcije unutar jezgre operacijskog sustava podržavaju stanje spavanja i ovakvo ponašanje nije uvijek dobro dokumentirano. Primjer takve funkcije je kopiranje podataka u korisnički prostor ili iz korisničkog prostora (*copy_to_user*, *copy_from_user*), tj. bilo koja funkcija koja mora alocirati memoriju može spavati[53].

Unutar kritične sekcije kôda moramo osigurati onemogućene sklopovske prekide. Razlog tomu je slijedeća situacija u kojoj proces drži *spinlock* zatvorenim prilikom generiranja sklopovskog prekida. Unutar prekidne funkcije također postoje kritične sekcije kôda koje su zaštićene istim *spinlock*-om. Budući da je taj *spinlock* već zatvoren procesor čeka unutar prekidne funkcije i „beskonačno“ provjerava kada će *spinlock* postati slobodan, taj se slučaj nikada neće dogoditi. Općenito vrijedi da ako jedna funkcija zatvara *spinlock* i poziva drugu funkciju koja također pokušava zatvoriti *spinlock* onda dobivamo problem koji se naziva *deadlock*. Da bi izbjegli ovakav slučaj moramo onemogućiti sklopovske prekide dok je odgovarajući *spinlock* zatvoren. Postoje posebne funkcije koje onemogućuju prekide prilikom zatvaranja *spinlock*-a, tj. u trenutku otvaranja *spinlock*-a ponovno vraćaju stanje prekidnih zastavica koje je bilo prije zatvaranja:

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

void spin_lock_irq(spinlock_t *lock);

void spin_lock_bh(spinlock_t *lock);
```

Zadnja funkcija onemogućava programske prekide, ali ostavlja sklopovske prekide omogućenima. Ako koristimo *spinlock* unutar prekidne funkcije moramo

[53] Primjerice funkcija *kmalloc* koja može prepustiti procesor jer čeka da još memorije postane slobodno

koristiti jednu od gore navedenih funkcija koje onemogućuju prekide, ali ako koristimo *spinlock* unutar programskog prekida onda možemo koristiti zadnju funkciju u gornjem navodu koja onemogućuje programske prekide dok ostavlja omogućene sklopovske prekide. Također, za svaku gornju funkciju koja zatvara *spinlock* postoji paralelna funkcija koja otvara *spinlock* kako slijedi:

```
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

void spin_unlock_irq(spinlock_t *lock);

void spin_unlock_bh(spinlock_t *lock);
```

Zadnja važna napomena jest da *spinlock* mora biti zatvoren što manje vremena. Što više držimo *spinlock* zatvorenim to više drugi procesi moraju čekati da ga otvorimo. S druge strane, unutar jezgre operacijskog sustava postoje procesi jako visokog prioriteta pa je stoga poželjno da ne zatvaramo *spinlock* dugo vremena jer onda procesi visokog prioriteta ne mogu dobiti prijeko potrebno procesorsko vrijeme.

Čitatelj se upućuje da pogleda unutar izvornog kôda upravljačkog programa gdje se sve koristi alat *spinlock* te da razmisli koji sve problemi mogu nastati ukoliko se potonji ne koristi.

6.3.9. Funkcija *ioctl*

Većina upravljačkih programa može obavljati više operacija osim jednostavnog prijenosa podataka. Korisnička aplikacija često puta mora moći narediti sklopovlju da napravi neku akciju koja ne spada u standardni prijenos podataka, npr. postavi *bitrate* CAN modula, resetiraj razne dijelove sklopovlja, postavi različite načine rada CAN modula itd. Prototip *ioctl* funkcije upravljačkog programa i ekvivalentne korisničke funkcije je:

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);

int ioctl(int fd, unsigned long cmd, ...);
```

Prva dva argumenta funkcije koju koristi upravljački program odgovaraju prvom argumentu korisničke *ioctl* funkcije i predstavlja tzv. *file descriptor* otvorene datoteke unutar korisničke aplikacije koja predstavlja čvor uređaja */dev/can0*. Drugi argument *ioctl* funkcije unutar upravljačkog programa prosljeđuje se bez promjene i predstavlja specifičnu naredbu upućenu sklopovlju. Ako aplikacija ne

proslijedi drugi argument, tj. treći argument iste funkcije upravljačkog programa, onda *arg* argument nije definiran. Tri točkice unutar definicije funkcije *ioctl* govore kompajleru da ne provjerava tip trećeg argumenta pa stoga kompajler ne može upozoriti programera ako taj argument nije definiran i teško je ponekad uočiti pogrešku. Kao što možemo vidjeti unutar izvornog kôda ove funkcije da se sastoji od velike *switch* naredbe koja odabire odgovarajuću operaciju u ovisnosti o *cmd* argumentu. Različite naredbe imaju različite numeričke vrijednosti kojima se obično daju simbolička imena kako bi se povećala čitljivost kôda. Simbolička imena su definirana pretprocesorskim definicijama unutar zaglavne datoteke.[54] Brojevi pojedine naredbe trebaju biti jedinstveni unutar jezgre operacijskog sustava kako bi se izbjegle neželjene pogreške krivog povezivanja naredbe i odgovarajućeg uređaja. Ako je *ioctl* naredba jedinstvena tada aplikacija prepoznaje povratnu vrijednost –EINVAL[55] (*invalid argument*) koja predstavlja pogrešku da naredba nije prepoznata.[56] Kako bi se pomoglo programerima da implementiraju jedinstvene naredbe, takvi kôdovi su podijeljeni u četiri polja bitova koja imaju sljedeća značenja[57]:

- tip (engl. *type*)
    - Ovo polje predstavlja čarobni broj. Programer jednostavno treba izabrati jedan broj i koristiti ga prilikom pisanja upravljačkog programa[58]. Ovo polje ima osam bita (_IOC_TYPEBITS).
- broj (engl. *number*)
    - Ovo polje predstavlja redni broj naredbe. Također ima osam bita (_IOC_NRBITS). Redni broj naredbe nema neko specifično značenje osim što razdvaja naredbe međusobno. Možemo čak koristiti jednak redni broj za naredbe čitanja kao i za naredbe pisanja jer se ta dva tipa naredbi razlikuju po smjeru prijenosa podataka.

[54] Unutar CAN upravljačkog programa dotična zaglavna datoteka je *can-lpc2000-user.h*

[55] POSIX standard tvrdi da ukoliko je naredba nepoznata treba vratiti –ENOTTY vrijednost. Ovu pogrešku C biblioteke interpretiraju kao „neprikladnu *ioctl* naredbu za uređaj“ (engl. *inappropriate ioctl for device*).

[56] Bolje i to nego da upravljački program napravi nešto što nismo namjeravali.

[57] Za detalje pogledajte <*linux/ioctl.h*>

[58] Poželjno je pogledati dokumentaciju *ioctl-number.txt*

- smjer (engl. *direction*)
    - Ako naredba uključuje prijenos podataka onda ovo polje označuje smjer prijenosa podataka. Moguće vrijednosti su _IOC_NONE (nema prijenosa podataka), _IOC_READ (čitanje), _IOC_WRITE (pisanje) i _IOC_READ | _IOC_WRITE (čitanje i pisanje). Referentni smjer prijenosa podataka se određuje gledajući od aplikacije prema upravljačkom programu. Dakle, _IOC_READ bi značilo da se čita iz uređaja što znači da upravljački program mora kopirati podatke u korisnički prostor.
- veličina (engl. *size*)
    - Širina ovoga polja bitova ovisi o arhitekturi ali obično se kreće od 13 do 14 bita. Možete naći specifičnu vrijednost za vašu korištenu arhitekturu koristeći makro _IOC_SIZEBITS. Ovo polje nije obavezno koristiti jer jezgra ga ne provjerava, ali se preporuča. Pravilno korištenje ovoga polja može pomoći pri detekciji aplikacijskih pogrešaka.

Unutar <*asm/ioctl.h*> definirani su makroi koji pomažu prilikom postavljanja naredbenih kôdova:

- `_IO(type,nr)`: za naredbu koja nema argumenata
- `_IOR(type,nr,datatype)`: za naredbu koja čita podatke iz upravljačkog programa
- `_IOW(type,nr,datatype)`: za naredbu koja upisuje podatke u upravljački program
- `_IOWR(type,nr,datatype)`: za naredbu koja može upisivati i čitati podatke

Polje bitova koje označuje veličinu (engl. *size*) se dobiva tako da se primjeni *sizeof* na tip podatka (*datatype*) trećeg argumenta. U nastavku dajemo izvadak iz datoteke *can-lpc2000-user.h* koji pokazuje *ioctl* naredbe[59]:

[59] Također, postoje već predefinirane *ioctl* naredbe koje jezgra operacijskog sustava prepoznaje. Više o ovim naredbama možete pročitati u literaturi [1].

```
#define CANLPC2000_IOC_MAGIC     'c'

/* Naredbe */
#define CANLPC2000_IOCRESET       _IO(CANLPC2000_IOC_MAGIC,0)
#define CANLPC2000_IOCSBITRATE    _IOW(CANLPC2000_IOC_MAGIC,1,int)
#define CANLPC2000_IOCSMODE       _IOW(CANLPC2000_IOC_MAGIC,2,int)
#define CANLPC2000_IOCSSRR        _IOW(CANLPC2000_IOC_MAGIC,3,int)
#define CANLPC2000_IOCSAFTABLE    _IOW(CANLPC2000_IOC_MAGIC,4,int)

#define CANLPC2000_IOC_MAXNR      5
```

Ako pogledamo prototipove gornjih *ioctl* funkcija vidimo da se koristi poseban treći argument koji ima ulogu argumenta naredbe koja se nastoji poslati upravljačkom programu. Ako je taj argument cjelobrojnog tipa onda se može koristiti direktno, ali ako je pokazivač onda treba poduzeti neke druge mjere opreza. Kada se koristi pokazivač koji pokazuje na korisnički prostor adresa onda moramo osigurati da je korisnička adresa ispravna. Pokušaj pristupa adresi korisničkog prostora koja nije unaprijed provjerena može dovesti sistemskih pogrešaka, sigurnosnih problema i neadekvatnog ponašanja. Na upravljačkom programu leži odgovornost da provjeri da je li korisnička adresa ispravna te ukoliko nije da vrati –EFAULT vrijednost aplikaciji koja ga koristi. U prošlim poglavljima uveli smo funkcije *copy_from_user* i *copy_to_user* koje se koriste prilikom prebacivanja podataka iz korisničkog u jezgreni prostor i obratno. Ove funkcije se također koristi i kod funkcije *ioctl*, ali *ioctl* pozivi obično uključuju malu količinu podataka za prijenos. Stoga su implementirane razne druge funkcije koje su optimizirane za prijenos najčešće korištenih veličina podataka (1B, 2B, 4B i 8B)[60]:

- __put_user(datum, ptr): ovaj makro se koristi kako bi se zapisala određena vrijednost tipičnog tipa podatka u korisnički prostor. Relativno je brz i trebao bi se koristiti umjesto funkcije *copy_to_user* kada god se prenose manje količine podatka. Također, ovaj makro je napisan tako da dozvoljava primanje bilo kakvog tipa pokazivača sve dok pokazuje na adrese korisničkog prostora. Količina prenesenih podataka ovisi o tipu pokazivača i određena je prilikom kompajliranja koristeći *sizeof* i *typeof* funkcije. Primjerice, ako je pokazivač *char* tipa onda je količina podataka koja se treba prenijeti jednaka jednom oktetu. Ovaj makro provjerava da

[60] Ove funkcije se mogu naći unutar <*asm/uaccess.h*>

li proces može pisati na adresu zapisanu unutar pokazivača. Vraća vrijednost 0 ako je uspješno zapisano ili –EFAULT ako se dogodi pogreška.

- __get_user(local, ptr): ovaj makro se koristi kako bi preuzeli određenu vrijednost tipičnog tipa podatka iz korisničkog prostora adresa. Ponaša se jednako kao i prošli makro s tom razlikom što ima drugačiji smjer prijenosa podataka. Preuzeta vrijednost se sačuva unutar lokalne varijable *local* dok povratna vrijednost govori da li je operacija uspjela ili ne.

Prije poziva navedenih makroa moramo pozvati funkciju koja će provjeriti ispravnost adrese koja se nalazi unutar korisničkog prostora:

```
int access_ok(int type, const void *addr, unsigned long size);
```

Prvi argument treba biti VERIFY_READ ili VERIFY_WRITE ovisno o tome da li čitamo iz korisničkog prostora ili pišemo u njega. Drugi argument sadrži adresu korisničkog prostora, a *size* je količina okteta podataka (*sizeof(datatype)*). Ova funkcija vraća logičku vrijednost: 1 ako je adresa validna, a 0 ako adresa nije validna. Ukoliko ova funkcija vrati 0 tada upravljački program mora vratiti vrijednost -EFAULT korisničkog aplikaciji. Postoji još nekoliko napomena u vezi spomenute funkcije: ona ne provodi cijeli posao provjere pristupa memoriji nego samo provjerava da li pokazivač referencira memoriju kojoj proces može pristupiti i osigurava da adresa ne označuje memoriju unutar jezgrinog prostora.

U nastavku ćemo dati cijeli izvorni kôd funkcije *ioctl* koju implementira CAN upravljački program:

```
int candev_ioctl (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg)
{
    int retval = 0;
    int err = 0;
    int tempint;
    int dev;
    struct can_lpc2000_aftable aftable;

    if (_IOC_TYPE(cmd) != CANLPC2000_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > CANLPC2000_IOC_MAXNR) return -ENOTTY;

    if (_IOC_DIR(cmd) & _IOC_READ)
    {
        err = !access_ok(VERIFY_WRITE, (void __user *)arg,
_IOC_SIZE(cmd));
    }else if (_IOC_DIR(cmd) & _IOC_WRITE)
```

```
    {
        err = !access_ok(VERIFY_READ, (void __user *)arg,
_IOC_SIZE(cmd));
    }
    if (err)
        return -EFAULT;

    dev = iminor(inode);

    switch (cmd) {
        case CANLPC2000_IOCRESET:
        lpc2xxx_can_setup(dev,CAN_OPERATING_MODE,LPC2000_CANDRIVER_CANBIT
RATE100K28_8MHZ);
            break;
        case CANLPC2000_IOCSBITRATE:
            retval = __get_user(tempint,(int __user*)arg);
            if (retval == 0) {
                lpc2xxx_can_setup(dev,lpc2xxx_can_get_mode(dev),tempint);
            }
            break;
        case CANLPC2000_IOCSMODE:
            retval = __get_user(tempint,(int __user*)arg);
            if (retval == 0) {

        lpc2xxx_can_setup(dev,tempint,lpc2xxx_can_get_bitrate(dev));
            }
            break;
        case CANLPC2000_IOCSSRR:
            retval = __get_user(tempint,(int __user*)arg);
            if (retval == 0) {
                candev[dev].srr_bit = tempint;
            }
            break;
        case CANLPC2000_IOCSAFTABLE:
            if (copy_from_user (&aftable, (void*)arg, sizeof(struct
can_lpc2000_aftable))) {
                retval = -EFAULT;
                break;
            }
            lpc2xxx_can_set_afmode(CAN_OFF_AFMODE);

            CANSFF_sa     = aftable.sff_sa;
            CANSFF_GRP_sa = aftable.sff_grp_sa;
            CANEFF_sa     = aftable.eff_sa;
            CANEFF_GRP_sa = aftable.eff_grp_sa;
            CANENDofTable = aftable.endoftable;

            memcpy((void *)(volatile unsigned int *) CAFMEM,
&aftable.table, aftable.endoftable);

            lpc2xxx_can_set_afmode(aftable.afmode);
            break;
        default:
            return -ENOTTY;
    }

    return retval;
}
```

Unutar funkcije možemo vidjeti koncepte koji su prikazani u ovom poglavlju. Nakon provjere adrese na koju pokazuje pokazivač *arg* slijedi odabir između nekoliko naredbi koje se mogu uputiti CAN modulu: resetiranje CAN modula, postavljanje *bitrate*-a, postavljanje načina rada CAN modula, zahtjev za primanjem poruke od drugog aktivnog CAN modula (CAN1) i postavljanje modula za filtriranje nadolazećih CAN poruka.

6.3.10. Blokirajuće ulazno-izlazne operacije

Što se događa ako upravljačkog programu dođe zahtjev za čitanjem podataka dok podataka još nema ili nisu dostupni za čitanje? S druge strane, što napraviti ako aplikacija nastoji slati podatke upravljačkom programu većom brzinom nego što ih on može obraditi? Aplikacijske programere obično ne zanimaju ovakve situacije jer kada pozovu *read* ili *write* funkcije očekuju da će upravljački program reagirati po upućenom zahtjevu. U ovakvim situacijama upravljački program treba **blokirati** takav proces, tj. staviti ga u stanje spavanja sve dok se odgovarajući zahtjev ne može nastaviti izvršavati. Kada je proces stavljen u režim spavanja to znači da je u posebnom stanju i kao takav izbačen je iz standardnog reda izvršavanja skupa aktivnih procesa (engl. *scheduler*). Sve dok neki događaj ne promijeni to stanje proces neće ući u red za izvršavanje, tj. neće dobivati procesorsko vrijeme.

Već postoje ugrađene funkcije koje jezgra operacijskog sustava pruža upravljačkim programima kako bi bili u stanju staviti proces u stanje spavanja. Prije nego uvedemo takve funkcije navesti ćemo nekoliko napomena koje su jako bitne kada govorimo o stavljanju procesa u stanje spavanja. Prva napomena jest da nikada ne stavljamo proces u stanje spavanja kada se izvršavaju atomske operacije, tj. blok kôda kojem se ne smije prekinuti izvođenje. To znači da ne možemo stavljati proces u stanje spavanja dok izvršavamo kôd koji je zatvorio *spinlock*; to također znači da ne možemo spavati dok imamo onemogućene prekide. Druga napomena koju bi trebalo zapamtiti je da kada se proces probudi onda on ne zna što se u međuvremenu dogodilo i promijenilo. Također, nikada ne znamo da li je možda neki drugi proces bio u stanju spavanja i čeka na isti događaj kao i prvi proces; u tom slučaju bi mogao zauzeti resurse na koje je prvi proces čekao. Dakle, ne smijemo pretpostavljati kakvo je stanje okoline kada se proces probudi nego uvijek treba provjeriti da li se zaista ispunio uvijek kojeg smo

čekali. Treća i najvažnija napomena jest da proces nikada ne smijemo stavljati u stanje spavanja ako prije toga nismo osigurali da neki drugi proces ili događaj probudi proces u spavanju. Da bismo to osigurali moramo dobro razmisliti kako kodiramo i znati točan slijed događaja koji se moraju dogoditi kako bi se proces u spavanju probudio. Postavlja se pitanje: kako pronaći proces koji je u stanju spavanja? To je riješeno unutar jezgre operacijskog sustava pomoću mehanizma, tj. pomoću strukture podataka pod nazivom *wait queue* ili **red za čekanje**. Redom za čekanje se upravlja pomoću strukture podataka tipa *wait_queue_head_t*[61] koja predstavlja vrh (glavu) takvog reda za čekanje. Spomenuta struktura je u osnovi jednostavna i sastoji se od *spinlock*-a i povezane liste. Ono što ide unutar takve povezane liste jest struktura tipa *wait_queue_t* koja sadrži informacije o procesu u stanju spavanja i o načinu kako proces želi da ga se probudi. Takva struktura se može definirati i inicijalizirati na statični i dinamični način:

```
DECLARE_WAIT_QUEUE_HEAD(name);

wait_queue_head_t moj_red;

init_waitqueue_head(&moj_red);
```

Unutar CAN upravljačkog programa korišten je dinamički pristup pomoću funkcije *init_waitqueue_head* pri čemu su definirana dva reda: *rque* i *fifoque*[62]. Inicijalizacija je provedena unutar funkcije *lpc2xxx_can_init* koju zbog veličine nećemo navoditi unutar ovoga rada.[63]

Kada je proces u stanju spavanja onda on očekuje da će se neki uvijek zadovoljiti u budućnosti. Kao što smo rekli, svaki proces koji je u stanju spavanja mora prilikom buđenja provjeriti da li se ispunio uvjet za koji je čekao u stanju spavanja. Postoje četiri tipa makroa koji se mogu koristiti unutar jezgre operacijskog sustava:

```
wait_event(queue_head, condition);

wait_event_interruptible(queue_head, condition);

wait_event_timeout(queue_head, condition, timeout);
```

[61] Definirana unutar <*linux/wait.h*>

[62] Deklarirani su unutar strukture tipa *candev_t*.

[63] Čitatelj se upućuje da pogleda izvorni kôd upravljačkog programa.

```
wait_event_interruptible_timeout(queue_head, condition, timeout);
```

Drugi argument predstavlja uvjet koji se provjerava prije i nakon spavanja nekog procesa; sve dok uvjet ne postane istinit, spavanje se nastavlja. Prvi makro postavlja proces u stanje spavanja kojeg nikakav prekid ne može probudit što obično nije ono što bi programer htio. Unutar CAN upravljačkog programa koristi se druga funkcija koja omogućava da se spavanje prekine signalom. Ova verzija vraća cjelobrojnu vrijednost koju treba provjeriti: nenegativna vrijednost znači da je spavanje prekinuto nekim signalom i upravljački program treba vratiti negativnu vrijednost -ERESTARTSYS korisničkoj aplikaciji. Dvije zadnje verzije čekaju jedan određeni vremenski period koji kada istekne makroi vraćaju 0 bez obzira na istinitost uvjeta.

Druga polovica slike su funkcije za buđenje procesa u stanju spavanja. Dakle, neki drugi proces ili prekid mora obaviti sigurno buđenje procesa koji je u stanju spavanja. Navodimo dvije najčešće korištene funkcije za buđenje:

```
void wake_up(wait_queue_head_t *queue);

void wake_up_interruptible(wait_queue_head_t *queue);
```

Prva funkcija će probuditi sve procese koji spavaju unutar danog reda za čekanje. Druga će se ograničiti na buđenje procesa koji su u stanju spavanja koje može biti prekinuto raznim prekidnim signalima. U narednim poglavljima će biti opisani blokovi kôda u kojima se pozivaju gornje funkcije unutar CAN upravljačkog programa, ali nakon što objasnimo još jedan važan koncept unutar jezgre operacijskog sustava - redovi za izvršavanje funkcija[64] (engl. *work queue*).

6.3.11. Ne blokirajuće ulazno-izlazne operacije

Postoje situacije u kojima neki proces ne može ili ne želi blokirati, tj. biti stavljen u stanje spavanja. Tada govorimo o ne blokirajućim ulazno-izlaznim operacijama koje se prepoznaju po O_NONBLOCK zastavici u polju *flip->f_flags*[65]. Naziv zastavice vodi podrijetlo od engleskog izraza *open-nonblock* zato što se specificira prilikom otvaranja. Ukoliko je ova zastavica specificirana i pozove se funkcija *read* ili *write* dok podaci nisu spremni onda se treba vratiti vrijednost -EAGAIN koja

[64] Ovo je slobodan prijevod ovog mehanizma pa se preporučuje koristiti engleski termin.

[65] Ova zastavica je definirana unutar <*linux/fcntl.h*> koji je automatski uključen unutar <*linux/fs.h*>.

govori aplikaciji da pokuša ponovno kasnije. Dakle, ne blokirajuće operacije se vraćaju odmah ukoliko podaci nisu dostupni i dozvoljavaju aplikaciji da „propitkuje“ upravljački program da li se može izvršiti određena operacija. Jedino *read*, *write* i *open* sistemski pozivi mogu biti ne blokirajući.

6.3.11.1. Funkcija *poll*

Aplikacije koje koriste ne blokirajuće ulazno-izlazne operacije često koriste funkcije *poll*, *select* i *epoll*. Unutar CAN upravljačkog programa korištena je funkcija *poll* koju ćemo detaljnije opisati u nastavku. Sve navedene funkcije u osnovi imaju jednaku ulogu koja se očituje u tome da daju informaciju aplikaciji da li može izvršiti operaciju koju je namjeravala. Ove funkcije mogu također blokirati procese dok odgovarajuće operacije ne postanu dostupne. Za sva tri sistemska poziva koja su istaknuta na početku poglavlja zajednička je *poll* metoda upravljačkog programa koja ima sljedeći prototip:

```
unsigned int (*poll)(struct file *filp, poll_table *wait);
```

Kada god korisnička aplikacija pozove *poll*, *select* i *epoll* sistemske pozive upravljački program pozove prethodnu *poll* funkcije. Ova funkcija poziva funkciju *poll_wait* nad jednim ili više redova za čekanje koji mogu dojaviti promjenu u statusu neke operacije. Na kraju vraća bit masku koja opisuje operacije koje mogu biti izvedene odmah bez blokiranja. Drugi argument (struktura *poll_table*[66]) funkcije *poll* se koristi unutar jezgre kako bi se implementirali tri spomenuta sistemska poziva. Programeri ne trebaju uopće znati kako se interno koristi takva struktura nego ju trebaju koristiti kako je opisano unutar funkcije *poll*. Upravljački program dodaje redove za čekanje, koji bi mogli probuditi proces i promijeniti status *poll* operacije, u strukturu *poll_table* pomoću funkcije *poll_wait*:

```
void poll_wait(struct file *, wait_queue_head_t *, poll_table *);
```

Povratna vrijednost je bit maska koja opisuje koje se operacije mogu izvršiti bez blokiranja, tj. odmah prilikom zahtjeva. Primjerice, ako su podaci spremni za čitanje onda bi funkcija *poll* trebala vratiti slijedeću vrijednosti: POLLIN | POLLRDNORM. Za cijeli skup povratnih zastavica molimo pogledajte literaturu [1].

[66] Deklarirana je unutar <*linux/poll.h*>.

U nastavku je dan kôd funkcije *candev_poll* koju implementira CAN upravljački program.

```
unsigned int candev_poll(struct file *filp, poll_table *wait)
{
    unsigned int mask = 0;
    int dev = (int)filp->private_data;

    poll_wait(filp,&candev[dev].rque,wait);
    spin_lock(&candev[dev].slock);
    if ((candev[dev].fifo).count > 0)
    {
        mask |= POLLIN | POLLRDNORM;
    }
    spin_unlock(&candev[dev].slock);

    return mask;
}
```

Ova funkcija se poziva u svrhu dobivanja informacije da li je podatak dostupan za čitanje, tj. preuzimanje od korisničke aplikacije koja u tom slučaju poziva funkciju *select* koja ima sljedeći prototip:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

Čitatelj se upućuje na literaturu [1] da više sazna o ovoj funkciji te kako se koristi. Važno je zapamtiti da je ovo funkcija koja je definirana za korisničke aplikacije. Primjer korištenja je dan unutar aplikacije koja koristi CAN upravljački program (*cantest.c*) o kojoj će više riječi biti kasnije.

6.3.12. Redovi za izvršavanje funkcija

Redovi za izvršavanje funkcija (engl. *workqueues*) osiguravaju izvršavanje određene funkcije u nekom skorašnjem budućem vremenu. Također, programer može implementirati da se izvođenje određene funkcije odgodi za točno određeni vremenski interval. Redovi za izvršavanje funkcija su tipa *struct workqueue_struct* koja je definirana unutar <*linux/workqueue.h*>. Red za izvršavanje funkcija mora se eksplicitno stvoriti prije nego se koristi. Funkcije za takvu upotrebu su:

```
struct workqueue_struct *create_workqueue(const char *name);

struct workqueue_struct *create_signlethread_workqueue(const char *name);
```

Svaki red za izvršavanje funkcija ima jedan ili više pridijeljenih procesa, tj. jezgrenih dretvi koje izvršavaju funkcije koje su unutar odgovarajućeg reda. Ako koristimo *create_workqueue* dobivamo red za izvršavanje koji ima svoju vlastitu

dretvu za svaki procesor koji se nalazi u sustavu. Obično je dovoljna jedna dretva pa stoga možemo koristiti posljednju funkciju *create_singlethread_workqueue*. Da bi pridijelili funkciju odgovarajućem redu za izvršavanje trebamo inicijalizirati strukturu *work_struct*. Ovo može biti napravljeno u vrijeme kompajliranja:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Prvi argument označuje ime strukture koju namjeravamo deklarirati, drugi argument je pokazivač na funkciju koja će se izvršavati, a zadnji argument predstavlja podatke koji se šalju u funkciju. Ako trebamo postaviti red za izvršavanje funkcija u toku izvršavanja koristimo sljedeća dva makroa:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

```
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

Postoje dvije funkcije koje povezuju odgovarajuću strukturu *work_struct* s odgovarajućim redom:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
```

```
int queue_delayed_work(struct workqueue_struct *queue, struct
                                work_struct *work, unsigned long delay);
```

Ukoliko se koristi zadnja funkcija onda se određeni zadatak ne izvodi dok ne prođe određeno vrijeme (zadnji argument). Povratna vrijednost je 0 ako je zadatak uspješno dodan dok povratna vrijednost koja nije jednaka 0 znači da je odgovarajuća struktura *work_struct* već dodana unutar reda pa neće biti dodana drugi put. Dakle, u nekom vremenu u budućnosti odgovarajuća funkcija će biti pozvana s parametrom *data*. Takva funkcija može staviti proces u stanje spavanja, ali ne može pristupiti korisničkom prostoru jer se izvodi unutar jezgrene dretve. Na kraju, ako ne želimo više koristiti odgovarajući red za izvršavanje funkcija onda možemo pozvati sljedeću funkciju:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

Upravljački program koji je implementiran unutar ovog rada koristi redove za izvršavanje funkcija koji već postoje i koje također koriste neki drugi resursi. Upravljački program često puta uopće ne treba vlastite redove za izvršavanje jer obično se ne zahtjeva tako intenzivno korištenje redova pa je jednostavnije i

isplativije koristiti redove koje pruža jezgra operacijskog sustava. Dakako, moramo biti svjesni da ovakve redove koriste i drugi resursi unutar operacijskog sustava pa stoga nije poželjno da vršimo bilo kakav monopol nad takvih redom. Primjerice, nije dozvoljeno predugo spavanje. Unutar CAN upravljačkog programa deklaracija korištenja takvog reda se obavlja unutar strukture *candev_t* koju ovdje navodimo u cijelosti tako da se čitatelj može podsjetiti i ostalih pojmova koje smo spominjali:

```
struct candev_t {
    int dev;
    atomic_t can_available;
    struct work_struct read_wq; /* uočiti */
    wait_queue_head_t rque, fifoque;
    int srr_bit;
    int rfinish ;
    spinlock_t slock;
    struct can_fifo fifo;
};
```

Ova struktura definirana je u *can-lpc2000.h* datoteci. Unutar funkcije *lpc2xxx_can_init* inicijalizirana je odgovarajuća funkcija koja će se izvršavati unutar reda *read_wq* koji predstavlja jezgrin inicijalni red. Sljedeća linija kôda unutar spomenute funkcije dodaje funkciju *wq_reed_feed* unutar *read_wq* reda za izvršavanje:

```
INIT_WORK(&candev[i].read_wq, wq_read_feed);
```

Sada još samo treba dati zahtjev globalnom jezgrinom redu za izvršavanje da stavi *read_wq* u svoj raspored kako bi se izvršila funkcija *wq_read_feed*. To se može napraviti koristeći sljedeću funkciju:

```
int schedule_work(struct work_struct *work);
```

Ova funkcija se koristi unutar prekidne funkcije za primitak poruke CAN modula mikrokontrolera LPC2478. Prije nego što prikažemo tok izvođenja cijelog upravljačkog programa prilikom slanja i primitka poruke moramo obraditi prekidni sustav unutar operacijskog sustava (uC)Linux.

6.3.13. Prekidni sustav

Budući da je gotovo uvijek nepoželjno da procesor čeka na neki vanjski događaj potrebno je implementirati prekidni sustav koji će obavijstiti procesor kada se nešto dogodi. Unutar CAN upravljačkog programa spomenuti vanjski događaj predstavlja primitak CAN poruke. Jezgra operacijskog sustava (uC)Linux upravlja prekidima na isti način kao i signalima unutar korisničkog prostora. Upravljački program jedino mora registrirati prekidnu funkciju za odgovarajući prekid i poslužiti ih pravilno. Naglašavamo ponovno da se prekidi izvode istovremeno kada i ostali jezgreni kôd pa je stoga potrebno paziti na tehnike međusobnog isključivanja.

Od jezgrenog modula, tj. od upravljačkog programa se očekuje da položi zahtjev jezgri operacijskog sustava za prekidnim kanalom prije nego ga koristi, odnosno da ga oslobodi u trenutku kada mu više nije potreban. Sljedeća funkcija, deklarirana unutar <*linux/interrupt.h*>, implementira način registracije 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);

Argumenti ove dvije funkcije imaju sljedeća značenja:

- insigned int irq
  - broj prekida koji se zahtjeva
- irqreturn_t (*handler)(int, void *, struct pt_regs *)
  - pokazivač na prekidnu funkciju koju želimo instalirati
- unsigned long flags
  - bit maska koja označuje razne opcije za ovu funkciju
- const char *dev_name
  - ovo ime se koristi unutar */proc/interrupts* kako bi se prikazao vlasnik prekida
- void *dev_id
  - pokazivač koji se koristi ako se prekidna linija dijeli i s drugim upravljačkim programima. Ako se prekidna linija ne dijeli onda se ovo polje može ostaviti na NULL.

Funkcija *request_irq* se poziva unutar inicijalizacijske funkcije modula *lpc2xxx_can_init* što baš i nije dobra ideja jer sprječavamo da drugi upravljački program koristi prekid koji smo rezervirali u slučaju da se uređaj za koji je rezerviran prekid nikada i ne koristi[67]. Iz tog razloga dobra je ideja instalirati prekid unutar funkcije *open*. Unutar CAN upravljačkog programa pretpostavlja se da prekidna linija ipak neće ostati ne iskorištena. Također, najbolje mjesto za poziv funkcije *free_irq* je u trenutku kada se pozove funkcija *close*. Kada god procesor primi sklopovski prekid CAN modula poveća se unutrašnji brojač omogućujući tako da provjerimo ispravnost rada samog uređaja. Prijavljeni prekidi su prikazani unutar */proc/interrupts* direktorija pa se čitatelj upućuje da prikupi odgovarajuće informacije u cilju provjere ispravnosti rada modula na razvojnom sustavu. Ukoliko čitatelj želi saznati više o prekidnom sustavu operacijskog sustava (uC)Linux upućuje se na literaturu [1].

U nastavku je dana implementacija prekidne funkcije CAN upravljačkog programa. Najprije prihvatimo prekid upisujući vrijednost 0 u registar *VICVectAddr* pa nakon toga provjerimo da li je stigla poruka. Ukoliko je poruka stigla onda ju spremamo u cirkularni fifo spremnik i pozovemo funkciju *schedule_work* koja osigurava da će se u skoroj budućnosti pozvati funkcija *wq_read_feed*.

```
irqreturn_t lpc2xxx_can_interrupt(int irq, void *dev_id)
{
    struct can_lpc2000_message msg;
    int rdev;

    VICVectAddr = 0;

    if (CAN1GSR & CANGSR_DOS)
    {
        CAN1CMR = CANCMR_CDO;
    }

    rdev = -1;
    if (CAN1GSR & CANGSR_RBS)
    {
        msg.FI = CAN1RFS;
        msg.ID = CAN1RID & 0x1fffffff;
        msg.DA = CAN1RDA;
        msg.DB = CAN1RDB;
        rdev = 0;

        CAN1CMR = CANCMR_RRB;
```

[67] Možemo postaviti da više uređaja koriste istu prekidnu liniju (isti prekid) sve dok se ne koriste u isto vrijeme

```
    }

    if (CAN2GSR & CANGSR_DOS)
    {
        CAN2CMR = CANCMR_CDO;
    }

    if (CAN2GSR & CANGSR_RBS)
    {
        msg.FI = CAN2RFS;
        msg.ID = CAN2RID & 0x1fffffff;
        msg.DA = CAN2RDA;
        msg.DB = CAN2RDB;
        rdev = 1;

        CAN2CMR = CANCMR_RRB;
    }

    /* Ako je poruka primljena */
    if (rdev !=-1)
    {
         /* Spremi u cirkularni fifo spremnik */
        spin_lock(&candev[rdev].slock);
        if(fifo_add(&candev[rdev].fifo,&msg) < 0)
        {
            if (printk_ratelimit())
                printk(KERN_WARNING "\tcan%d fifo buffer se
prepunio!!!\n", rdev);
        }
        spin_unlock(&candev[rdev].slock);
        schedule_work(&candev[rdev].read_wq);
    }

    return IRQ_HANDLED;
}
```

Uloga funkcije *wq_read_feed* jest da probudi procese koji čekaju unutar *rque* reda za čekanje. Svi takvi procesi čekaju unutar funkcije *candev_read* da pristigne poruka u cirkularni fifo spremnik. Budući da se poziv funkcije *wq_read_feed* inicira samo unutar prekidne rutine, u kojoj se primljena poruka već spremila u cirkularni fifo spremnik, očito da će procesi koji se probude unutar funkcije *candev_read* imati što pročitati. Unutar funkcije *wq_read_feed* procesi se stavljaju u *fifoque* red za čekanje u kojem čekaju da čitanje poruke završi, tj. da se dođe do kraja funkcije *candev_read* prilikom čega će se i čekanje unutar funkcije *wq_read_feed* završiti pozivom funkcije za buđenje procesa koji čekaju unutar *fifoque*. Sve se to

kontrolira zastavicom *rfinish*[68] koja je definirana unutar strukture tipa *candev_t*. Da bi sve bilo jasnije navodimo kôd funkcije *wq_read_feed*.

```
void wq_read_feed (struct work_struct *work)
{
    int dev;
    struct candev_t *pdev;

    pdev = container_of(work, struct candev_t, read_wq);
    dev = pdev->dev;

    while ((pdev->fifo).count > 0)
    {
        wake_up_interruptible(&pdev->rque);
        if (wait_event_interruptible(pdev->fifoque, pdev->rfinish == 1)
== -ERESTARTSYS)
        {
            pdev->rfinish = 0;
            break;
        }
        pdev->rfinish = 0;
    }
}
```

Oprezni čitatelj bi mogao primijetiti da smo cijelu funkciju *wq_read_feed* mogli uključiti u prekidnu funkciju. Da li bi to bila dobra ideja? Prilikom pisanja prekidnih funkcija uvijek se moramo voditi pravilom da ne smijemo izvoditi preduge zadatke unutar prekidne funkcije. Često puta prilikom primitka sklopovskog prekida moramo obaviti dugotrajne zadatke, ali istovremeno prekidnu funkciju moramo zadržati jednostavnom i ne blokirati prekid na dugo vremena. Jezgra (uC)Linux operacijskog sustava rješava ovaj problem na već opisan način tako što razdvaja prekid na dva dijela: gornju polovicu (engl. *top half*) i donju polovicu (engl. *bottom half*). Gornju polovicu sačinjava odgovarajuća prekidna funkcija koju smo instalirali koristeći funkciju *request_irq* i koja odgovara prilikom primitka sklopovskog prekida. S druge strane, donja polovica prekida odgovara redovima za izvršavanje funkcija[69], tj. odgovara funkciji koja će napraviti ostatak posla kojeg prekidna funkcija nije stigla napraviti, ali u neko sigurnije vrijeme koje će odrediti red za izvršavanje funkcija. Bitna razlika gornje i donje polovice prekida jest što su svi prekidi omogućeni dok se izvodi donja polovica prekida. U tipičnom scenariju

[68] engl. read finish

[69] Unutar donje polovice spadaju i drugi mehanizmi unutar jezgre kao što su *tasklet*-i, koje nećemo obrađivati unutar ovoga rada jer nisu korišteni prilikom pisanja upravljačkog programa za CAN modul.

gornja polovica prekida sačuva primljene podatke od nekog uređaja unutar spremnika i rasporedi izvođenje donje polovice prekida koja će se izvesti u neko buduće vrijeme. Nakon takve brze operacije primitka prekida izvodi se donja polovica koja obavlja većinu zadataka: budi procese koji čekaju na neki poseban događaj, pokreću druge ulazno-izlazne operacije itd. Ono što je važno primijetiti jest da gornja polovica čim završi si primitkom tekućeg prekida može nastaviti dalje primati prekide dok se donja polovica izvodi. Na ovaj način je također izveden CAN upravljački program gdje donju polovicu prekida upravo predstavlja funkcija *wq_read_feed* koja je iskorištena da bi se probudili odgovarajući procesi na čekanju za primitak poruka.

6.3.14. Korisnička aplikacija

Unutar ovog rada također je implementirana aplikacija koja koristi CAN upravljački program. Aplikacija se može pronaći unutar *cantest.c* datoteke izvornog kôda. Podrazumijeva se da čitatelj razumije osnove C/C++ programskog jezika i osnovne funkcije standardne biblioteke C funkcija. Unutar ovog poglavlja prikazati ćemo izvorni kôd aplikacije i ukratko objasniti tok izvođenja.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include "can-lpc2000-user.h"

#define DEVICE "/dev/can0"
#define MSG_BUF_SIZE 16

void print_frame(struct can_lpc2000_message *msg)
{
    int j;

    if (msg->FI & CAN_BP)
        printf("\tUSER:BP ");
    if (msg->FI & CAN_RTR)
        printf("\tUSER:RTR ");
    if (msg->FI & CAN_EFF)
        printf("\tUSER:EFF ");

    for (j = CAN_DLC_VAL(msg->FI) - 1; j >= 0 ; j--)
    {
        if (j < 4)
            printf("%02X ", (msg->DA >> j*8) & 0xff);
        else
            printf("%02X ", (msg->DB >> (j-4)*8) & 0xff);
    }
```

```
    printf("\n");
}

int main (int argc,char** argv)
{
    int i;
    int fd;
    fd_set fds;
    int rd,wr,sl;
    unsigned int ID_high, ID_low;
    unsigned int address = 0;
    unsigned int j = 0;
    struct can_lpc2000_message msg[MSG_BUF_SIZE];
    struct can_lpc2000_aftable aftable;
    int bitrate;
    struct timeval tv;
    char buf[256];
    int count = 5;

    fd = open(DEVICE,O_RDWR);
    if (fd<0)
    {
        perror("open "DEVICE);
        printf("\t\tUSER:Greska u otvaranju CAN device\n");
        return 0;
    }

    i = 0;
    memset(&aftable,0,sizeof(struct can_lpc2000_aftable));
    aftable.sff_sa = address;
    // Explicit Standard Frame
    for ( j = 0; j < 4; j += 2 )
    {
        ID_low = (j << 29) | (0x100 << 16);
        ID_high = ((j+1) << 13) | (0x100 << 0);
        aftable.table[i++] = ID_low | ID_high;
        address += 4;
    }
    // Group Standard Frame
    aftable.sff_grp_sa = address;
    for ( j = 0; j < 4; j += 2 )
    {
        ID_low = (j << 29) | (0x200 << 16);
        ID_high = ((j+1) << 13) | (0x200 << 0);
        aftable.table[i++] = ID_low | ID_high;
        address += 4;
    }
    // Explicit Extended Frame
    aftable.eff_sa = address;
    for ( j = 0; j < 4; j++ )
    {
        ID_low = (j << 29) | (0x100000 << 0);
        aftable.table[i++] = ID_low;
        address += 4;
    }
    // Group Extended Frame
    aftable.eff_grp_sa = address;
    for ( j = 0; j < 4; j++ )
    {
        ID_low = (j << 29) | (0x200000 << 0);
        aftable.table[i++] = ID_low;
```

```
        address += 4;
    }
    // End of Table
    aftable.endoftable = address;

    aftable.afmode = CAN_NORMAL_AFMODE;
    if (ioctl(fd,CANLPC2000_IOCSAFTABLE,&aftable) == -1)
    {
        perror("ioctl aftable");
        printf("\t\tUSER:Greska u postavljanju ioctl aftable\n");
        goto out;
    }

    bitrate = LPC2000_CANDRIVER_CANBITRATE100K28_8MHZ;
    ioctl(fd,CANLPC2000_IOCSBITRATE,&bitrate);

    while (count>0) // while(1)
    {
        --count;
        tv.tv_sec = 1;
        tv.tv_usec = 100000;

        msg->FI = 0x00080000; // 11b, no RTR, DLC=8B
        msg->ID = 0x100;
        msg->DA = 0x55AA55AA;
        msg->DB = 0xAA55AA55;
        wr = write(fd,&msg,sizeof(struct can_lpc2000_message));
        if (wr < 0)
        {
            perror("write");
            printf("\t\tUSER:Greska u slanju poruke upravljackom
programu\n");
        }

        if ((sl = select(fd+1, &fds, NULL, NULL, &tv)) == -1) {
            if (errno == EINTR)
                continue;
            perror("select");
            printf("\t\tUSER:Select nije uspio\n");
            goto out;
        }

        rd = read(fd,msg,MSG_BUF_SIZE*sizeof(struct
can_lpc2000_message));
        for (i=0; i < rd; i++)
        {
            if (rd > 0) {
                printf("\t\t\tUSER:Procitao sam nesto... slijedi poruka:
");
                print_frame(&msg[i]);
            } else {
                perror("read "DEVICE);
                printf("\t\t\tUSER:Nisam nista procitao\n");
                break;
            }
        }
    }
out:
    close(fd);
    return 0;
}
```

Unutar aplikacije prva naredba pokušava otvoriti CAN uređaj, tj. čvor */dev/can0* unutar datotečnog sustava pomoću funkcije *open*. Nakon što smo uspješno otvorili uređaj postavljamo modul koji filtrira nadolazeće CAN poruke[70]. Da bi omogućili konfiguriranje spomenutog modula moramo pozvati *ioctl* sistemski poziv koji će kopirati cijelu konfiguraciju koju smo upravo postavili u odgovarajući dio memorije kojoj upravljački program ima pristup. Nakon postavljanja odgovarajućeg modula za filtriranje poruka postavimo *bitrate* na kojemu će raditi CAN modul koristeći ponovno funkciju *ioctl*. Ulazimo unutar *while* petlje koja se ponavlja *count* broj puta. Unutar petlje postavljamo odgovarajuća polja CAN poruke i šaljemo ju upravljačkom programu pomoću *write* funkcije. Nakon toga pozivamo funkciju *select* koja poziva funkciju *poll* upravljačkog programa u kojoj se čeka primitak CAN poruke. Kada se primi CAN poruka poziva se funkcija *read* pomoću koje aplikacija može pročitati primljenu poruku od upravljačkog programa. Nakon što je poruka dostupa korisničkoj aplikaciji ispisujemo ju na konzolu[71] pomoću funkcije *print_frame*. Na kraju aplikacije pozivom funkcije *close* zatvaramo CAN uređaj.

6.4. Tok izvođenja upravljačkog programa

Korištenjem jezgrene funkcije *printk* možemo ispisivati razne poruke na konzolu pri čemu možemo obavljati *debugging* cijelog upravljačkog programa. Unutar originalnog izvornog kôda možemo vidjeti odsječke kôda kako slijedi:

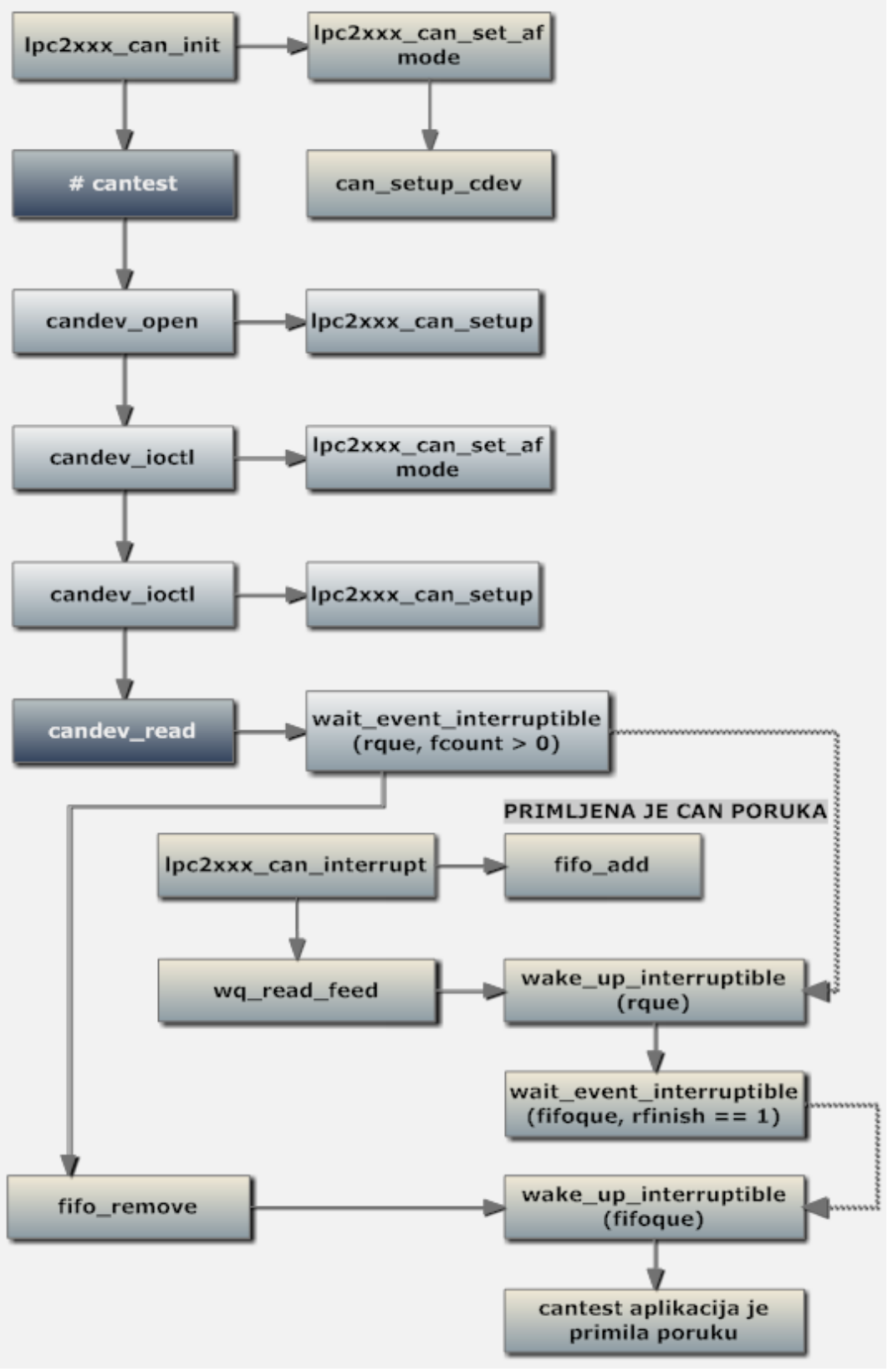
```
#ifdef CAN_LPC2000_DEBUG
        printk(KERN_INFO "\tPozdrav svijetu!\n");
#endif
```

Ako je definirana konstanta CAN_LPC2000_DEBUG onda se kompajlira kôd koji ispisuje poruke na konzolu. Ispisivanje poruka iskorišteno je da bi se odredio tok izvođenja upravljačkog programa i također da bi se otklonile pogrješke koje su nastale prilikom programiranja. Ova funkcija omogućuje klasifikaciju poruka po razini važnosti koja se određuje pomoću makroa koji se nalazi odmah ispred poruke koja se ispisuje; u gornjem primjeru se radi o KERN_INFO. Velika razina važnosti poruke dozvoljava joj ispis na konzolu.

[70] Neka čitatelj pogleda literaturu [1].

[71] Podsjećamo da je konzola ostvarena putem serijske veze razvojnog sustava i osobnog računala.

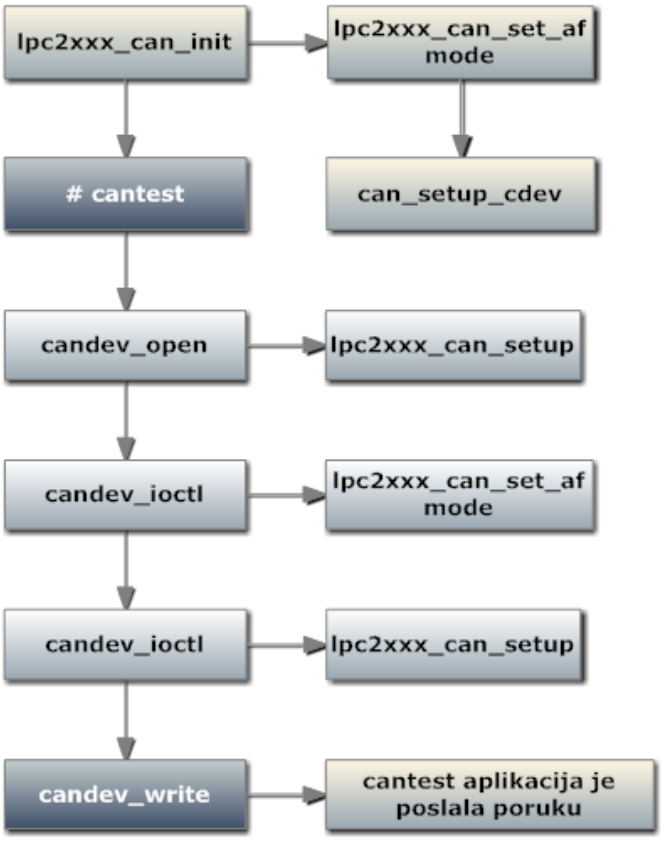
6.4.1. Tok izvođenja prilikom čitanja iz upravljačkog programa



Slika 27. Tok izvođenja prilikom primitka CAN poruke

Prva funkcija *lpc2xxx_can_init* poziva se prilikom podizanja operacijskog sustava i obavlja početnu inicijalizaciju upravljačkog programa. Nakon toga posebno je označeno pojavljivanje sučelja konzole gdje korisnik može upravljati operacijskim sustavom. Unutar terminala upišemo naziv aplikacije (*cantest*) koja će koristiti CAN upravljački program. Prilikom izvođenja aplikacije poziva se niz sistemskim poziva: *open*, *ioctl* i *read*. Ako cirkularni fifo spremnik nema spremljenu CAN poruku onda se proces stavlja u čekanje unutar *rque* sve dok se ne probudi unutar funkcije *wq_read_feed* čije se izvršavanje odvija ukoliko se poruka primila. Isprekidanim strelicama su prikazani procesi spavanja i buđenja. Nakon što smo probudili procese koji su čekali da stigne poruka u cirkularni fifo spremnik ponovno čekamo dok ne završi sistemski poziv čitanja koji uzima poruku iz cirkularnog fifo spremnika i budi procese koji su čekali na kraj sistemskog poziva *read.* Neka čitatelj pogleda izvorni kôd u svrhu detaljnijeg razumijevanja pojedine funkcije.[72]

6.4.2. Tok izvođenja prilikom pisanja u upravljački program



Slika 28. Tok izvođenja prilikom slanja CAN poruke

[72] Unutar toka izvođenja izostavljen je sistemski poziv *close* radi preglednosti. On nije previše bitan za razumijevanje onoga što se željelo prikazati.

Tok izvođenja prilikom slanja CAN poruke je očigledno jednostavniji od onoga prilikom primitka poruke jer nije korišten izlazni cirkularni spremnik i nisu korišteni napredni mehanizmi koje pruža jezgra operacijskog sustava (uC)Linux kao što je to kod čitanja. Tok izvođenja je identičan sve do sistemskog poziva *write* pomoću kojeg aplikacije šalje poruku upravljačkom programu koji se brine za odgovarajuću konfiguraciju CAN modula.

7. Kompajliranje jezgre

Unutar ovog poglavlja prikazat ćemo postupak ugrađivanja CAN upravljačkog programa u složeni sustav kompajliranja jezgre operacijskog sustava (uC)Linux.[73] Koliko je takav sustav složen svjedoči činjenica da jezgra operacijskog sustava Linux sadrži više od 6.000.000 linija kôda. Nadalje, složenosti sustava za kompajliranje pridonosi svojstvo relativno jednostavnog dodavanja novih modula koji se trebaju kompajlirati; to se postiže velikim brojem (oko 800) *makefile*-ova[74] unutar izvornog kôda jezgre. Već je rečeno da Linux ima monolitnu strukturu jezgre što znači da se cijela jezgra prevodi i povezuje unutar jedinstvene statički povezane izvršne datoteke. Također, moguće je kompajlirati i naknadno povezati (engl. *incrementally link*) određeni jezgrin kôd u odgovarajuće module koji su pogodni za naknadno uključivanje prilikom izvođenja jezgre. Ovo je uobičajena metoda uključivanja upravljačkih programa u jezgru operacijskog sustava, ali nije korištena prilikom projektiranja CAN upravljačkog programa jer se nastojalo odmah statički povezati taj modul u jezgru. Također, nakon što se jezgra inicijalizira i korisniku se omogući interakcija preko terminala potrebno je pozvati poseban program koji uključuje module u jezgru (*insmod*), ako modul već prilikom kompajliranja nije uključen u jezgru.

7.1. Konfiguracija jezgre

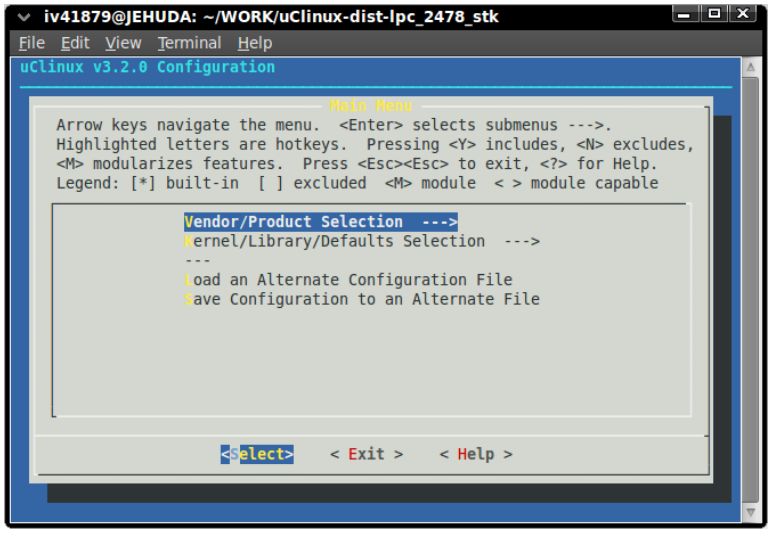
U ranim fazama razvitka jezgre Linux operacijskog sustava konfiguracija se odvijala preko skripte koja je (doslovno) postavljala pitanja razvojnom programeru ukoliko želi uključiti odgovarajuću funkcionalnost jezgre ili ne. To ne zvuči loše ako zanemarimo činjenicu da konfiguracijskih parametara jezgre ima jako puno, tj. programer je morao odgovoriti na više od 600 pitanja[75]. Da bi se izbjegla frustrirajuća konfiguracija jezgre uvedena su grafička sučelja koja su uvelike

[73] Budući da cilj ovog rada nije konfiguracija i prilagodba (uC)Linux jezgre razvojnom sustavu LPC2478STK, nećemo taj posao raditi od nule već će se smatrati da je takav posao napravljen. Kompanija Olimex je napravila taj posao i ponudila gotovu razvojnu okolinu za upravljačke programe.

[74] Datoteka koja sadrži specifične naredbe za kompiliranje izvornog kôda kojeg obavlja alat pod imenom *make*.

[75] Situacija postaje gora ako ste slučajno krivo odgovorili na jedno pitanje jer se onda postupak mora ponoviti za sva ostala pitanja koja su već dobro odgovorena.

olakšala konfiguraciju. Danas postoji 10 grafičkih sučelja pomoću kojih se konfigurira jezgra među kojima je najpoznatiji *menuconfig* koji je prikazan na sljedećoj slici.



Slika 29. Izbornik grafičkog sučelja *menuconfig*

Gornje grafičko sučelje pokreće se naredbom *make menuconfig* pri čemu moramo biti pozicionirani unutar glavnog direktorija distribucije uCLinux-a[76]. Unutar distribucije koja dolazi s razvojnim sustavom LPC2478STK napravljen je odgovarajući izbornik u kojem izabiremo sljedeće opcije:

- Vendor Selection -> NXP
- Product Selection -> LPC2468

Nakon izlaska iz *menuconfig* slijedi čitav niz konfiguriranja raznih parametara u svrhu pripreme jezgre operacijskog sustava za kompajliranje koje će biti prikazano u narednim poglavljima.

7.2. Kompajliranje upravljačkog programa i aplikacije

U nastavku će biti opisan postupak dodavanja CAN upravljačkog programa i odgovarajuće aplikacije u sustav kompajliranja. Važnu ulogu u tom postupku imaju *Kconfig*[77] i *Makefile* datoteke za svaki pojedini direktorij u kojem se nalazi izvorni

[76] Na slici se može vidjeti da se koristi distribucija koja se nalazi na pratećem CD-u Olimex-ovog razvojnog sustava LPC2478STK. Već je naglašeno da se podrazumijeva konfigurirana jezgra za odgovarajući razvojni sustav.

[77] engl. *Kernel Configuration*

kôd upravljačkog programa ili aplikacije. *Kconfig* datoteke upravljaju procesom konfiguracije za onaj kôd koji se nalazi unutar direktorija takve datoteke. Konfiguracijski sustav jezgre parsira sadržaj *Kconfig* datoteke koji predstavlja odgovarajući izbor, tj. pitanje prema korisniku o vrijednosti odgovarajućeg parametra. Konfiguracijski parametar popraćen je opisnim tekstom kako bi korisnik mogao lakše razumjeti što konfigurira. Da bi omogućili odabir funkcionalnosti CAN upravljačkog programa i aplikacije od korisnika potrebno je modificirati *Kconfig* datoteku unutar *linux-2.6.x/drivers/char* direktorija. U nastavku je prikazana *diff* datoteka koja s znakom (+) označuje koje linije kôda treba dodati, a znakom (-) označuje koje linije kôda treba oduzeti unutar spomenute *Kconfig* datoteke. Ove dvije opcije biti će dostupne korisniku prilikom konfiguracije jezgre.

```
        help
          Driver for the M41T11M6 Real Time Clock Chip.

+config CAN_DRV_LPC2XXX
+       tristate "NXP LPC2XXX CAN support"
+       depends on ARCH_LPC22XX
+       default n
+       help
+         CAN upravljacki program za lpc2xxx
+
+config CAN_DRV_LPC2XXX_DEBUG
+       tristate "NXP LPC2XXX CAN debug"
+       depends on CAN_DRV_LPC2XXX
+       default n
+       ---help---
+         Omoguci ispis poruka prilikom debugging-a upravljackog programa
+
 config TELCLOCK
        tristate "Telecom clock driver for ATCA SBC"
        depends on EXPERIMENTAL && X86
```

Sada je potrebno dodati odgovarajuće datoteke izvornog kôda[78] upravljačkog programa unutar istog direktorija. Sljedeća datoteka koju je potrebno modificirati u svrhu kompajliranja upravo dodanih datoteka izvornog kôda jest *Makefile* datoteka koja se nalazi u istom direktoriju. Slijedi prikaz odgovarajuće *diff* datoteke.

```
obj-$(CONFIG_LEDMAN)           += ledman.o
obj-$(CONFIG_M41T11M6)         += m41t11m6.o
+obj-$(CONFIG_CAN_DRV_LPC2XXX) += can-lpc2000.o
obj-$(CONFIG_RESETSWITCH)      += resetswitch.o
obj-$(CONFIG_MCFWATCHDOG)      += mcfwatchdog.o
```

[78] *can-lpc2000.c*, *can-lpc2000.h*, *can-lpc2000-user.h*.

Lako možemo uočiti da se datoteka *can-lpc2000.c* dodaje u sustav kompajliranja jezgrinog kôda. Također, vidimo upotrebu prve opcije *Kconfig* datoteke (CAN_DRV_LPC2XXX) koja određuje da li će se ova datoteka kompajlirati. Postoje dvije moguće vrijednosti koje ta opcija može sadržavati nakon što ju korisnik postavi: y ili n[79]. Ukoliko korisnik odgovori potvrdnim odgovorom onda će se upravljački program statički povezati u jezgru prilikom kompajliranja. Ako bismo umjesto ta dva odgovora omogućili vrijednost 'm'[80] to bi značilo da će se upravljački program kompajlirati, ali se neće statički povezati u jezgru operacijskog sustava prilikom kompajliranje, tj. trebat ćemo ga dinamički povezati koristeći se alatom *insmod*. To je sve što je potrebno napraviti da bi se odgovarajući upravljački program kompajlirao. Još trebamo napraviti modifikaciju datoteke izvornog kôda pod nazivom *lpc2468_devices.c* koja se nalazi unutar *linux-2.6.x/arch/arm/mach-lpc2xxx*. Slijedi sadržaj odgovarajuće *diff* datoteke gdje se konfigurira korištenje pinova P0.0 i P0.1 kao odgovarajući CAN pinovi RD1 i TD1. Da bi bilo moguće definirati odgovarajuće konstante potrebno je modificirati datoteku *lpc_2478_stk_defconfig* koja se nalazi unutar *linux-2.6.x/arch/arm/configs* direktorija.

```
static void __init lpc22xx_add_can_device(void)
 {
+#ifdef CONFIG_LPC2468_PINSEL_P0_0_RD1
+       lpc22xx_set_periph(LPC22XX_PIN_P0_0, 1, 0);   /* RD1 */
+#endif
+#ifdef CONFIG_LPC2468_PINSEL_P0_1_TD1
+       lpc22xx_set_periph(LPC22XX_PIN_P0_1, 1, 0);   /* TD1 */
+#endif

        platform_device_register(&lpc22xx_can_device);
 }
```

U nastavku je naveden sadržaj *diff* datoteke koju je potrebno promijeniti zbog ne slaganja odgovarajućih memorijskih adresa raznih modula mikrokontrolera LPC2478 s ranijim inačicama obitelji LPC2000. Ovaj dio ne spada u standardni postupak modificiranja ključnih datoteka ali je potreban kako bi bilo moguće ispravno konfiguriranje mikrokontrolera.

[79] engl. yes/no

[80] engl. module

Unutar datoteke *lpc2xxx.h* koja se nalazi unutar *linux-2.6.x/include/asm-arm* direktorija treba dodati/oduzeti sljedeće linije kôda.

```
#define CAN1BTR         (*((volatile unsigned long *) 0xE0044014))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN1EWL         (*((volatile unsigned long *) 0xE004401C))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN1SR          (*((volatile unsigned long *) 0xE0044020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN1RFS         (*((volatile unsigned long *) 0xE0044024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN1EWL         (*((volatile unsigned long *) 0xE0044018))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN1SR          (*((volatile unsigned long *) 0xE004401C))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN1RFS         (*((volatile unsigned long *) 0xE0044020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN1EWL         (*((volatile unsigned long *) 0xE004401C))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN1SR          (*((volatile unsigned long *) 0xE0044020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN1RFS         (*((volatile unsigned long *) 0xE0044024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN1RID         (*((volatile unsigned long *) 0xE0044024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
#define CAN1RDA         (*((volatile unsigned long *) 0xE0044028))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */


#define CAN2BTR         (*((volatile unsigned long *) 0xE0048014))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN2EWL         (*((volatile unsigned long *) 0xE004801C))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN2SR          (*((volatile unsigned long *) 0xE0048020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
-#define CAN2RFS         (*((volatile unsigned long *) 0xE0048024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN2EWL         (*((volatile unsigned long *) 0xE0048018))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN2SR          (*((volatile unsigned long *) 0xE004801c))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN2RFS         (*((volatile unsigned long *) 0xE0048020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN2EWL         (*((volatile unsigned long *) 0xE004801C))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN2SR          (*((volatile unsigned long *) 0xE0048020))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+//#define CAN2RFS         (*((volatile unsigned long *) 0xE0048024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
+#define CAN2RID         (*((volatile unsigned long *) 0xE0048024))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
#define CAN2RDA         (*((volatile unsigned long *) 0xE0048028))
/* lpc2119\lpc2129\lpc2292\lpc2294 only */
```

Još samo trebamo dodati aplikaciju *cantest* unutar sustava za kompajliranje jezgre operacijskog sustava. Korisničke aplikacije se dodaju unutar *user/cantest*

direktorija kojeg prethodno trebamo napraviti. Dodajemo dvije datoteke izvornog kôda: *cantest.c* i *cantest.h*. Budući da su aplikaciji također potrebne ključne strukture podataka korištene unutar upravljačkog programa moramo napraviti simbolički link[81] u trenutni direktorij na datoteku *can-lpc2000-user.h*. Da bi kompajliranje aplikacije bilo moguće potrebno je dodati *Makefile* datoteku unutar trenutnog direktorija. Sadržaj takve datoteke naveden je u nastavku. Molimo čitatelja da obrati pozornost na *romfs* dio koji označava da će se odgovarajuća aplikacija instalirati u *romfs* datotečni sustav pod *bin* direktorijem.

```
PATH+=:/home/iv41879/WORK/uClinux-dist-lpc_2478_stk/tools

EXEC=cantest
OBJS=cantest.o

CC=ucfront-gcc arm-linux-gcc
CFLAGS=-Os -g -fomit-frame-pointer -pipe -msoft-float -fno-common -fno-
builtin -Wall -DEMBED \
-D__PIC__ -fpic -msingle-pic-base -Dlinux -D__linux__ -Dunix -
D__uClinux__
LDFLAGS=-Wl,--fatal-warnings -Wl,-elf2flt -msoft-float -D__PIC__ -fpic -
msingle-pic-base \
-Wl,--fatal-warnings -Wl,-elf2flt -msoft-float -D__PIC__ -fpic -msingle-
pic-base


all: ${EXEC}

${EXEC}: ${OBJS}
        $(CC) ${LDFLAGS} -o $@ $(OBJS) $(LDLIBS)

$(OBJS): cantest.h

wput: ${EXEC}
        wput -uB -nc ${EXEC} ftp://localhost/inc/${EXEC}

romfs:
        $(ROMFSINST)     /bin/$(EXEC)

clean:
        rm -f *.o ${EXEC} *.gdb
```

Slijedeći upute pod nazivom *Adding-User-Apps-HOWTO* koje se nalaze dokumentacijske mape moramo napraviti još par koraka da bi konačno mogli kompajlirati jezgru.

[81] Njega možemo stvoriti tako što na datoteku kliknemo desnom tipkom miša i odaberemo opciju *Make Link*. Nakon toga stvoreni simbolički link kopiramo u odgovarajući direktorij.

Unutar *Makefile* datoteke koja se nalazi u *user* direktoriju trebamo dodati sljedeću liniju:

```
dir_$(CONFIG_USER_CAN_EXAMPLES)             += can4linux
+dir_$(CONFIG_USER_CANTEST_CANTEST)          += cantest
dir_$(CONFIG_USER_CTORRENT_CTORRENT)        += ctorrent
```

Ova linija kôda dodaju direktorij *cantest* u listu direktorija koje je potrebno kompajlirati. U nastavku je prikazana *diff* datoteka koja prikazuje što treba dodati unutar *config/Configure.help* datoteke u kojoj se nalazi opisni tekst koji se prezentira prilikom konfiguracije.

```
CONFIG_USER_CAN_EXAMPLES
  some example and test applications for the can4linux CAN driver

+CONFIG_USER_CANTEST_CANTEST
+  testna aplikacija za CAN upravljacki program (LPC2478)

CONFIG_USER_CAN_HORCH
  advanced CAN bus analyzer
```

Na kraju je potrebno još dodati sljedeću liniju kôda unutar *config/cinfig.in* datoteke ako želimo da se naša aplikacija pokaže unutar grafičkog sučelja *menuconfig*.

```
fi
bool 'fbtest'                   CONFIG_USER_FBTEST_FBTEST
+bool 'cantest'                 CONFIG_USER_CANTEST_CANTEST
bool 'flthdr'                   CONFIG_USER_FLTHDR_FLTHDR
```

7.3. Postupak kompajliranja

U nastavku ćemo prikazati unos odgovarajućih naredbi u terminal operacijskog sustava Ubuntu prilikom kompajliranja jezgre. Prije nego što možemo kompajlirati potrebno je instalirati odgovarajuće alate za kompajliranje. Budući da koristimo Olimex-ovu distribuciju operacijskog sustava uCLinux, također moramo koristiti skup GNU alata za prevođenje koji je korišten prilikom prilagođavanja odgovarajuće distribucije. Takav skup alata možemo pronaći na pratećem CD-u unutar *Utils* direktorija pod nazivom *arm-linux-tools-20061213.tar.gz*. Ovu arhivu potrebno je dekomprimirati i unutar PATH varijable dodati *bin* direktorij kako je već opisano u prethodnim poglavljima. Također, na korištenoj Linux distribuciji (Ubuntu) nisu inicijalno instalirani programski paketi koji su potrebni prilikom kompajliranja jezgre. Stoga je potrebno instalirati 3 paketa koja navodimo u

sljedećoj tablici u kojoj drugi stupac opisuje kakva se pogreška pojavljuje prilikom prevođenja ukoliko dotični paket nije instaliran.

Tablica 7. Paketi koji su potrebni prije kompajliranja jezgre uCLinux-a

libncurses5-dev	*cannot find - lnsurses*
zlib1g-dev	*not found zlib.h*
genromfs	*bin/sh: genromfs: command not found*

Svi navedeni paketi instaliraju se sljedećom naredbom:

```
sudo apt-get install <paket>
```

U svrhu bržeg kompajliranja napravljeni su odgovarajući *alias*-i koji je potrebno pozvati i sve se automatski izvede. Da bi to ostvarili najjednostavnije je dodati sljedeće linije unutar datoteke *.bashrc* koja se nalazi u *home/<korisnik>* direktoriju.

```
alias rmlink='rm -f tools/ucfront-gcc tools/ucfront-g++
tools/cksum tools/ucfront-ld lib/libz.a lib/uClibc'
alias mklink='rm -f linux-2.6.x; ln -s
linux-2.6.24.2-lpc2478-patched linux-2.6.x'
alias domenu='mklink; make mrproper; rmlink;
make menuconfig;'
alias mymake='make > ../compile_output.txt
2> ../compile_error.txt'
```

Nakon što smo sačuvali promjene i omogućili ih naredbom *source* možemo koristiti gornje naredbe. Sve što je potrebno jest unutar terminala se pozicionirati unutar početnog direktorija distribucije uCLinux-a i izvesti naredbu *domenu* koja će nas odmah dovesti do *menuconfig* grafičkog sučelja gdje odabiremo proizvođača NXP i produkt LPC2468. Nakon konfiguriranja ponovno u terminal unesemo naredbu *mymake* koja će prevesti cijelu jezgru operacijskog sustava unutar nekih 10-ak minuta. Unutar direktorija u kojem se nalazi cijela distribucija dodane su dvije datoteke: *compile_output.txt* i *compile_error.txt* u kojima možemo vidjeti kakve su se greške pojavile prilikom kompajliranja. Nakon što smo ispravili greške unutar direktorija *images* pojaviti će se dvije datoteke (*romfs_5.img*, *vmlinux.bin*) koje je potrebno kopirati na USB FLASH memoriju koju priključimo na USB priključak razvojnog sustava LPC2478STK. Nakon priključivanja napajanja pokreće se U-boot bootloader koji pokreće jezgru operacijskog sustava uCLinux

koja se nalazi na USB FLASH memoriji zajedno s datotečnim sustavom. Nakon što se jezgra pokrenula u terminalu možemo početi unositi naredbe operacijskog sustavu. Unosom naredbe *cantest* pokrećemo odgovarajuću aplikaciju koja izvodi testiranje CAN upravljačkog programa.

8. Zaključak

Standardi koji diktiraju današnje stanje tehnologije i očekivanja korisnika pred dizajnere ugradbenih računalnih sustava postavljaju zahtjeve za integracijom različitih naprednih sklopovskih i programskih tehnologija u malenom, aplikacijski-specifičnom sklopovlju niske potrošnje. uCLinux operacijski sustav posebna je varijanta Linux-a prilagođena 32-bitnim mikrokontrolerima bez sklopovske jedinice za upravljanje memorijom (MMU), koja omogućuje brzi razvoj složenih aplikacija korištenjem široke baze postojećeg otvorenog koda. Jedan od najbitnijih elemenata kod realizacije vlastitih sklopovskih rješenja predstavljaju upravljački programi (engl. *device drivers*), koji omogućuju povezivanje jezgre operacijskog sustava i korisničkih programa sa sklopovljem. Upravljački programi unutar arhitekture operacijskog sustava služe kako bi programeru korisničkih aplikacija pojednostavili pristup sklopovlju kroz apstraktno programsko sučelje poziva standardnih sistemskih funkcija OS-a. Takva modularnost pojednostavljuje razvoj i prenosivost složenih aplikacija između različitih platformi, razdvajajući implementaciju upravljačkih programa od jezgre OS-a i korisničkih aplikacija. U okviru rada detaljno je proučena procedura izrade upravljačkih programa za uCLinux. U okviru ovog rada proučene su mogućnosti korištenja uCLinux OS-a kao baze za razvoj složene programske potpore 32-bitnih ugradbenih računalnih sustava bez MMU jedinice, s posebnim naglaskom na problem izrade upravljačkih programa. Pokazalo se da unatoč razmjerno kvalitetnoj dokumentaciji i širokoj korisničkoj zajednici uCLinux nije jednostavno uhodati za novu sklopovsku platformu, ponajprije zbog većeg broja različitih modula i alata koji trebaju besprijekorno raditi zajedno, a koji se razvijaju potpuno neovisno i decentralizirano. Uspješno implementirani CAN upravljački program pokazao je da je uCLinux unatoč tome vrlo upotrebljiv, poglavito kada je cijena razvojnih alata i licenci važan faktor prilikom odabira platforme.

9. Literatura

1. J. Corbet, A. Rubini, G. Kroah-Hartman. **Linux Device Drivers**, 3th. O'Reilly. 2005.
2. H. Mihaldinec. **Jezgra uCLinux operacijskog sustava**. Diplomski rad. 2010.
3. ARM. **ARM7TDMI-S Technical Reference Manual**. 2001.
4. NXP. **LPC24XX User Manual**. 2009.
5. Olimex. **LPC2478STK Users Manual**. 2008.
6. USB-CAN. http://www.systec-electronic.com/html/index.pl/en_home. Datum pristupa: 29.6.2010.
7. GNUARM. http://www.gnuarm.com/. Datum pristupa: 29.6.2010.
8. Yahoo-GNUARM. http://groups.yahoo.com/group/gnuarm. Datum pristupa: 29.6.2010.
9. Buildlogs. http://kegel.com/crosstool/crosstool-0.43/buildlogs. Datum pristupa: 29.6.2010.
10. Gedit plugins. http://live.gnome.org/Gedit/Plugins. Datum pristupa: 29.6.2010.
11. K. Yaghmour, J. Masters, G. Ben-Yossef, P. Gerum. **Building Embedded Linux Systems**. O'Reilly. 2008.
12. P. Raghavan, A. Lad, S. Neelakandan. **Embedded Linux System Design and Development**. A. Publications. 2006.
13. W. E. Shotts. **The Linux Command Line**. 2008.-2009.

10. Dodatak A

10.1. GNUARM instalacijska skripta

```
#!/bin/sh

ROOT=`pwd`
SRCDIR=$ROOT/src
BUILDDIR=$ROOT/build
PREFIX=$ROOT/install

GCC_SRC=gcc-4.3.2.tar.bz2
GCC_VERSION=4.3.2
GCC_DIR=gcc-$GCC_VERSION

BINUTILS_SRC=binutils-2.19.tar.bz2
BINUTILS_VERSION=2.19
BINUTILS_DIR=binutils-$BINUTILS_VERSION

NEWLIB_SRC=newlib-1.16.0.tar.gz
NEWLIB_VERSION=1.16.0
NEWLIB_DIR=newlib-$NEWLIB_VERSION

INSIGHT_SRC=insight-6.8.tar.bz2
INSIGHT_VERSION=6.8
INSIGHT_DIR=insight-$INSIGHT_VERSION

echo "Instalirati cu arm-elf cross-kompajler:

  Prefix: $PREFIX
  Source: $SRCDIR
  Build: $BUILDDIR"
read IGNORE

#
# Pomocne funkcije
#
unpack_source()
{
(
    cd $SRCDIR
    ARCHIVE_SUFFIX=${1##*.}
    if [ "$ARCHIVE_SUFFIX" = "gz" ]; then
      tar zxvf $1
    elif [ "$ARCHIVE_SUFFIX" = "bz2" ]; then
      tar jxvf $1
    else
      echo "Unknown archive format for $1"
      exit 1
    fi
)
}

(
cd $SRCDIR


# Dekomprimiraj arhive izvornih kodova
unpack_source $(basename $GCC_SRC)
```

```
unpack_source $(basename $BINUTILS_SRC)
unpack_source $(basename $NEWLIB_SRC)
unpack_source $(basename $INSIGHT_SRC)
)

# Postavi PATH varijablu
OLD_PATH=$PATH
export PATH=$PREFIX/bin:$PATH

#
# 1: Kompajliraj binutils
#
(
(
# provjera autoconf
cd $SRCDIR/$BINUTILS_DIR

) || exit 1

mkdir -p $BUILDDIR/$BINUTILS_DIR
cd $BUILDDIR/$BINUTILS_DIR

$SRCDIR/$BINUTILS_DIR/configure --target=arm-elf --prefix=$PREFIX \
    --enable-interwork --enable-multilib --with-float=soft --disable-
werror \
    && make all install

) || exit 1

#
# 2: Kompajliraj GCC
#
(
MULTILIB_CONFIG=$SRCDIR/$GCC_DIR/gcc/config/arm/t-arm-elf

echo "

MULTILIB_OPTIONS += mno-thumb-interwork/mthumb-interwork
MULTILIB_DIRNAMES += normal interwork

" >> $MULTILIB_CONFIG

mkdir -p $BUILDDIR/$GCC_DIR
cd $BUILDDIR/$GCC_DIR

$SRCDIR/$GCC_DIR/configure --target=arm-elf --prefix=$PREFIX \
    --enable-interwork --enable-multilib --with-float=soft --disable-
werror \
    --enable-languages="c,c++" --with-newlib \
    --with-headers=$SRCDIR/$NEWLIB_DIR/newlib/libc/include \
    && make all-gcc install-gcc

) || exit 1

#
# 3: Kompajliraj i instaliraj newlib
#
(
(

cd $SRCDIR/$NEWLIB_DIR
```

```
) || exit 1

mkdir -p $BUILDDIR/$NEWLIB_DIR
cd $BUILDDIR/$NEWLIB_DIR

$SRCDIR/$NEWLIB_DIR/configure --target=arm-elf --prefix=$PREFIX \
    --enable-interwork --enable-multilib --with-float=soft --disable-
werror \
    && make all install

) || exit 1

#
# 4: Kompajliraj i instaliraj ostatak GCC-a
#
(
cd $BUILDDIR/$GCC_DIR

make all install

) || exit 1

#
# 5: Kompajliraj i instaliraj GDB INSIGHT.
#

(

mkdir -p $BUILDDIR/$INSIGHT_DIR
cd $BUILDDIR/$INSIGHT_DIR

$SRCDIR/$INSIGHT_DIR/configure --target=arm-elf --prefix=$PREFIX \
    --enable-interwork --enable-multilib --with-float=soft --disable-
werror \
    && make all install

) || exit 1


export PATH=$OLD_PATH

echo "Kraj! Dodaj $PREFIX/bin u PATH varijablu kako bi omogucili pozive
arm-elf-gcc alata."
```