Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Medved

Optimizacija zagona operacijskega sistema GNU/Linux na vgrajenih sistemih

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Patricio Bulić
Somentor: Robert Sedevčič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Čas zagona operacijskega sistema in aplikacije na vgrajenih sistemih je običajno kritičnega pomena. Aplikacije pogosto tečejo na posebni različici sistema GNU/Linux, ki je namenjena vgrajenim sistemom, kjer pa čas zagona ni optimalen. Problema optimizacije se bomo lotili v dveh korakih. Najprej bomo s pomočjo razvojnih orodij iSYSTEM diagnosticirali zagon sistema od zaganjalnika prve stopnje (ang. first stage bootloader) do uporabniške aplikacije. Temu sledi časovna optimizacija zaganjalnika U-boot ter jedra operacijskega sistema GNU/Linux z vsemi razpoložljivimi orodji.

Zahvala gre predvsem Robertu Sedevčiču iz podjetja iSYSTEM Labs d.o.o. za vso podporo in vodenje. Prav tako se zahvaljujem podjetju za priložnost in delo na tako zanimivem projektu ter vsakemu posamezniku, ki mi je na različnih področjih pri diplomskem delu pomagal.

Hvaležen sem tudi doc. dr. Patriciu Buliću za mentorstvo pri diplomski nalogi s strani fakultete.

Zahvaljujem se seveda tudi družini, ki me je brezpogojno podpirala skozi ves čas študija.

Navsezadnje pa gre zahvala tudi prijatelju Martinu Jerončiču za lektorstvo in pomoč pri slovničnem delu diplomske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
MCU	microcontroller unit	mikrokrmilnik
SoC	system on chip	sistem na čipu
FPGA	field programmable gate array	programabilni čip
OCM	on chip memory	pomnilnik na čipu
SRAM	static RAM	statični RAM
ROM	read only memory	bralni pomnilnik
OS	operating system	operacijski sistem
QSPI	quad serial peripheral interface	serijska periferna povezava
GPIO	general purpose input/output	splošno namenski vhod-izhod
UART	universal asynchronous receiver-transmitter	univerzalni asinhroni sprejemnik-oddajnik
DDR	double data rate	dvojna hitrost podatkov
OTG	on the go	
JTAG	Joint Test Action Group	
SD	Secure Digital	Secure Digital
DHCP	dynamic host configuration protocol	protokol za dinamično konfiguracijo gostitelja

Povzetek

Naslov: Optimizacija zagona operacijskega sistema GNU/Linux na vgrajenih sistemih

Avtor: Tadej Medved

Problematika diplomskega dela zajema optimizacijo zagona in vzpostavitev razvojnega sistema (ang. toolchain) za gradnjo GNU/Linux OS. Zadevo je potrebno raziskati s stališča strojne ter programske opreme. K problemu pristopimo tako, da zgradimo vse elemente, ki so potrebni za zagon GNU-/Linux OS z ničle, torej iz izvornih datotek dostopnih prek uradnih sistemov za nadzor izvorne kode. Na takšen način si zagotovimo največ svobode pri apliciranju optimizacij. S takšnim pristopom in optimizacijami, opisanimi v diplomski nalogi, lahko čas zagona na konkretnem Avnet ZedBoard vgrajenem sistemu zmanjšamo z začetnih 35 sekund na 1 sekundo.

Ključne besede: Vgrajen sistem, zagon, GNU/Linux, U-boot, ARM.

Abstract

Title: The optimization of the GNU/Linux operating system boot time on embedded systems

Author: Tadej Medved

The thesis' problematic requires that we optimize boot time of GNU/Linux OS and establish a building environment for building GNU/Linux OS. Research from a hardware and software perspective is required. We approach the problem by building all of the elements of GNU/Linux OS boot from zero, that means from source files fetched from official source version control systems. This way we get maximum freedom for applying optimizations. With this kind of approach and boot optimizations described in the thesis, boot time can be, on an actual Avnet ZedBoard embedded system, reduced from 35 seconds to 1 second.

Keywords: Embedded system, boot, GNU/Linux, U-boot, ARM.

Poglavje 1

Uvod

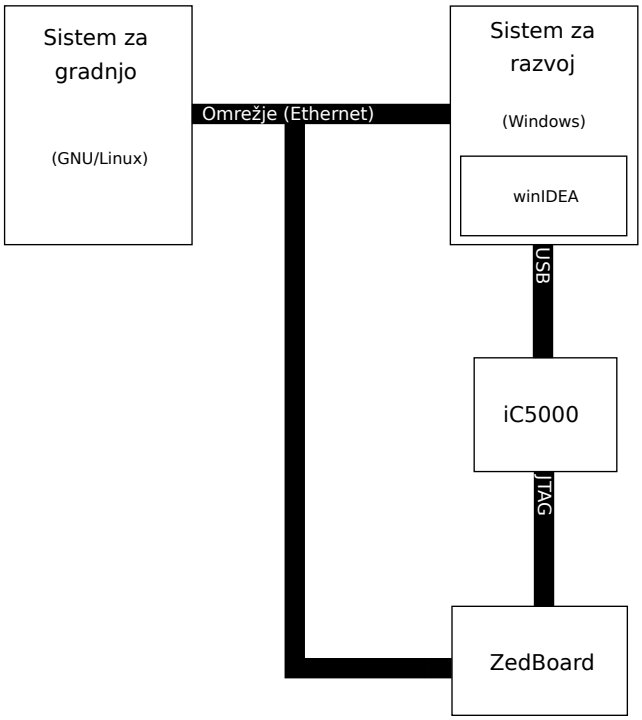
Danes se srečujemo z vgrajenimi sistemi praktično vsak dan; od trenutka, ko sedemo v avtomobil pa do trenutka, ko peremo perilo v pralnem stroju. Zato se mora z zahtevami po pametnejšem avtomobilu, ki morda omogoča celo samodejno vožnjo, in boljšim pralnim strojem istočasno razvijati tudi svet vgrajenih sistemov, da bo tem zahtevam lahko ugodil. Tako se začenja pojavljati težnja, da bi se na vgrajenem sistemu izvajal tudi pravi operacijski sistem, kot ga že mnoga leta poznamo na osebnih računalnikih, ki pa mora biti prirejen okolju. Najbolje se je tega izziva lotiti z GNU/Linux operacijskim sistemom, ki je odprtokoden in je tako mogoče urejanje izvorne kode in spreminjanje nastavitev gradnje.

Glavni izziv se skriva v samem času zagona takšnega sistema. Seveda nočemo, da bi po tem, ko obrnemo ključ v avtu, čakali še 1 minuto, da se vsi sistemi avtomobila vzpostavijo. Tu ne gre le za primer avtomobila, večina aplikacij na vgrajenih sistemih je časovno kritičnih. Ker je to še vedno velik problem v svetu vgrajenih sistemov, je bila moja naloga to raziskati v sodelovanju s podjetjem iSYSTEM Labs d.o.o. in čimbolj pospešiti sam zagon takšnega sistema.

Poglavje 2

Razvojno okolje

V tem poglavju so opisani elementi razvojnega okolja, katerega shema je razvidna na sliki 2.1.



Slika 2.1: Shema razvojnega okolja.

Sistem za razvoj predstavlja računalnik, s katerim razvijalec direktno upravlja. Na njem se izvaja razvojno okolje winIDEA. Prek omrežja je ta sistem povezan na sistem za gradnjo, na katerem poteka gradnja in križno prevajanje GNU/Linux operacijskega sistema za ciljni vgrajen sistem. Sistem za razvoj tako prek omrežja pridobiva končne binarne datoteke OS GNU-/Linux in jih prek razvojnega orodja winIDEA ter iC5000 nalaga na ciljni vgrajen sistem (ZedBoard).

2.1 ZedBoard

GNU/Linux OS je bil izdelan za vgrajen sistem ZedBoard proizvajalca Avnet. Na njem najdemo Xilinx Zynq XC7Z020 SoC, ki vključuje [40]:

- dve ARM Cortex A9 procesorski jedri,
- Xilinx FPGA,
- 256 KB OCM SRAM pomnilnika,

Ostali elementi, ki jih najdemo na vgrajenem sistemu ZedBoard so [2]:

- 512 MB DDR3 RAM pomnilnik,
- 32 MB QSPI flash pomnilnik,
- mrežna kartica,
- USB UART serijski kanal,
- USB OTG priklop,
- JTAG konektor za razhroščevanje,
- reža za priklop SD kartice.

2.2 iC5000

iC5000 je razvojno orodje podjetja iSYSTEM Labs d.o.o., ki se uporablja za razhroščevanje in analizo programske opreme na vgrajenih sistemih in omogoča [15]:

- programiranje vgrajenega sistema,
- osnovne funckije razhroščevanja (izvajanje kode po korakih, prekinitvene točke v programu ...),
- upravljanje s pomnilniki na vgrajenem sistemu,
- analizo izvajanja programske kode na vgrajenem sistemu.

iC5000 je preko USB vodila ali omrežja (Ethernet) povezan z računalnikom, kjer se izvaja integrirano razvojno okolje winIDEA. WinIDEA v povezavi z iC5000 ponuja razvijalcu možnost izvajanja zgoraj navedenih funkcionalnosti na vgrajenem sistemu.

iC5000 je bil za potrebe diplomske naloge z vgrajenim sistemom povezan preko JTAG povezave. Preko njega je potekalo zapisovanje v QSPI flash pomnilnik in razhroščevanje zaganjalnika (več o zaganjalniku v poglavju 3.2).

Poleg iC5000 je bila uporabljena tudi njegova razširitev, imenovana V/I modul (ang. I/O module). Ta modul ima več analognih in digitalnih vhodov ter izhodov za povezavo z vhodno-izhodnimi priklopi na vgrajenem sistemu [15]. Za potrebe diplomske naloge so bili uporabljeni digitalni vhodi (več o tem v poglavju 4.2).

Slika 2.2: iC5000 (levo) ter ZedBoard (desno).

2.3 Sistem za gradnjo

Ta računalniški sistem je namenjen gradnji in križnemu prevajanju (ang. cross compiling) vseh komponent GNU/Linux operacijskega sistema za ciljni vgrajen sistem.

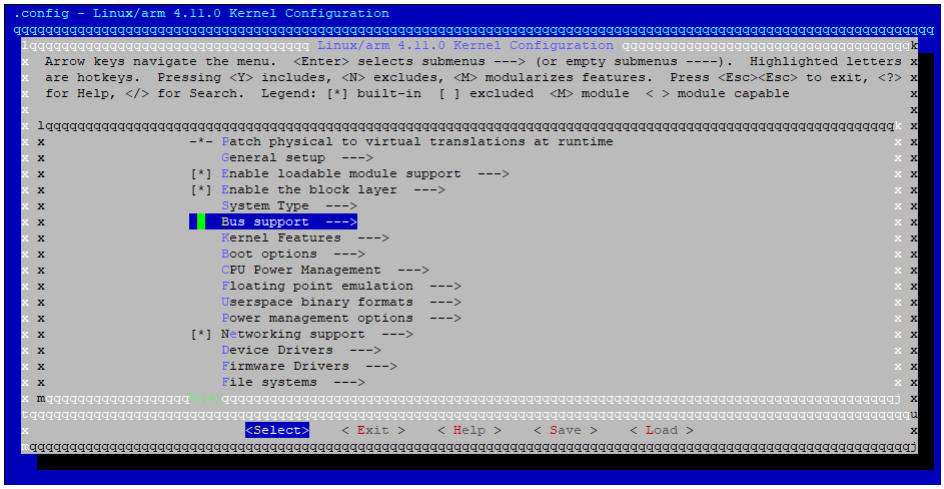
Sam sistem je zasnovan na operacijskem sistemu GNU/Linux. Gradnja in križno prevajanje elementov GNU/Linux OS je namreč na tem operacijskem sistemu bolje dokumentirana in posledično lažja kot na drugih sistemih (npr. Windows). Sistem je tako zasnovan na OS Ubuntu 14.04. Za križni prevajalnik pa je bila uporabljena ARM-ova izvedenka GCC prevajalnika za ARM arhitekturo.

Na samem sistemu najdemo tudi okolje za gradnjo, ki je opisano v naslednjem poglavju.

2.3.1 Okolje za gradnjo

V tem okolju poteka gradnja vseh elementov GNU/Linux OS za ciljni vgrajen sistem. Da je uporaba okolja preprostejša, najdemo v samem okolju skripte, napisane v skriptnem jeziku Bash, ki so nam v pomoč pri gradnji GNU/Linux OS.

Vsi elementi GNU/Linux OS pri gradnji uporabljajo Make sistem za gradnjo, ki temelji na Makefile datotekah. Te določajo, kako gradnja poteka [14]. Skripte omogočajo tudi zagon menijskega uporabniškega vmesnika (menuconfig), ki je del konfiguracijskega sistema gradnje [20]. Ta vmesnik omogoča konfiguracijo gradnje preko 'grafičnega' terminalskega vmesnika, ki ga vidimo na sliki 2.3.



Slika 2.3: Menijski vmesnik menuconfig.

Gradnja preko že pripravljenih Bash skript upošteva dva nivoja konfiguracije. Prvi je na nivoju samega Make sistema, drugi pa je na nivoju okolja, preko konfiguracijske datoteke *config.cfg*. Na tem nivoju določamo:

- pot do križnega prevajalnika,
- privzeto konfiguracijsko datoteko gradnje za določen element,
- ciljno arhitekturo.

Okolje je dostopno prek servisov SSH in Samba. To nam omogoča delo iz katerega koli sistema, ki podpira odjemalca za ta dva servisa. Pridobimo tudi na skalabilnosti, saj lahko več razvijalcev skupaj dela na tem projektu in si celoten sistem s pripadajočimi izvornimi datotekami in konfiguracijami deli.

Če okolje za gradnjo sami postavimo, vemo točno, kaj je potrebno za izgradnjo in delovanje GNU/Linux OS in kako gradnja poteka. To je akademsko gledano najboljša izbira, saj se tako največ naučimo. Problem nastane, ker je potrebno takšno okolje vzdrževati. To je pomembno v industriji, kjer vzdrževanje takšnega okolja vzame čas in delo, kar pa predstavlja dodaten strošek. Iz tega stališča je zato bolje v industrijskem okolju uporabiti že obstoječe okolje za gradnjo, ki je vzdrževano s strani odprtokodnih projektov. Primera takšnih okolij sta Yocto Project ter BuildRoot.

2.3.2 Struktura okolja za gradnjo

Samo okolje za gradnjo je direktorijsko strukturirano za lažjo uporabo na sledeč način:

```
/
├── dependencies
│   └── root_fs
├── output
│   ├── linux
│   ├── root_fs
│   └── u-boot
└── sources
    ├── linux
    ├── root_fs
    └── u-boot
```

V korenskem direktoriju najdemo že prej omenjene Bash skripte ter konfiguracijsko datoteko *config.cfg*. Ko skripto za gradnjo nekega določenega elementa GNU/Linux OS poženemo, najprej prebere datoteko *config.cfg* in glede na vsebino nastavi konfiguracijo gradnje. Skripta nato izgradi element iz izvornih datotek, ki se nahajajo v direktoriju *sources*. Pri tem, če je to potrebno, uporabi dodatne datoteke iz direktorija *dependencies*. Končni rezultat gradnje pa da v *output* direktorij.

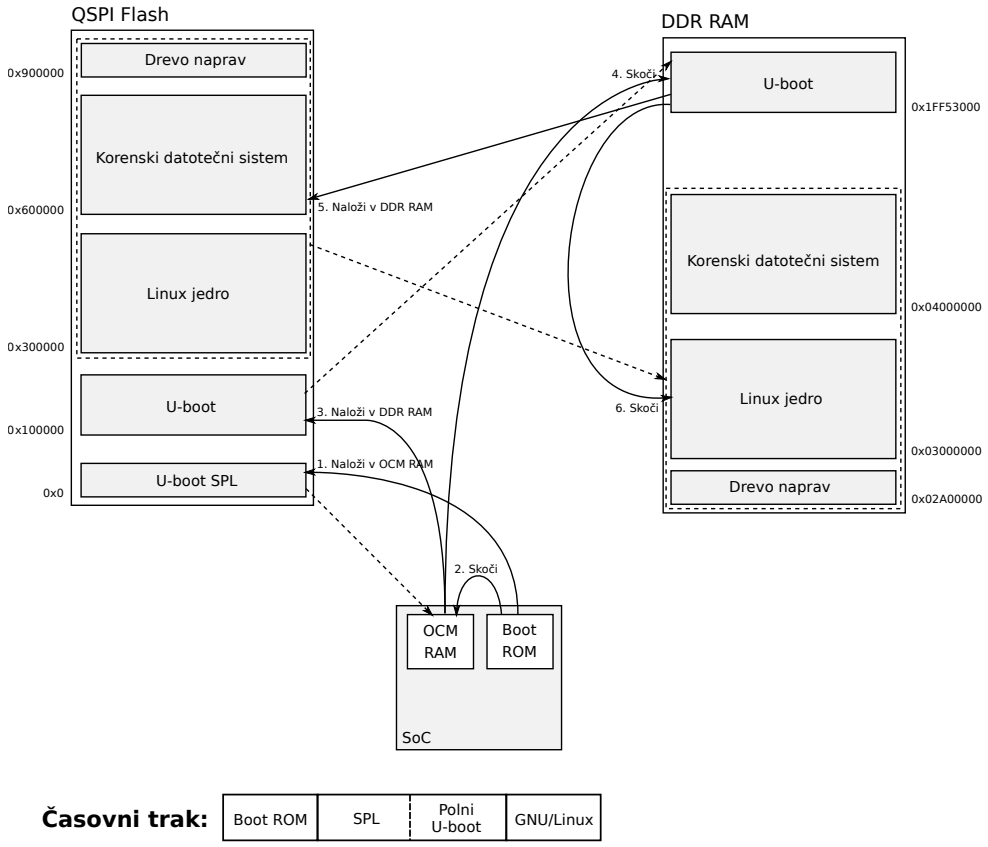
2.4 Sistem za razvoj

Sistem za razvoj je računalniški sistem, ki ga razvijalec neposredno uporablja. Do okolja za gradnjo dostopa preko SSH ter Samba protokolov. Uporablja pa se tudi za programiranje QSPI flash pomnilnika ter razhroščevanje na ciljnem vgrajenem sistemu prek okolja winIDEA. Posebnih zahtev glede operacijskega sistema tu ni. Podpirati mora SSH ter Samba odjemalca ter orodje winIDEA. Trenutno sta to Windows ter GNU/Linux.

Poglavje 3

Zagon operacijskega sistema GNU/Linux na vgrajenih sistemih

Če hočemo pospešiti zagon operacijskega sistema GNU/Linux na vgrajenih sistemih, moramo najprej razumeti, kako zagon poteka. Razdeljen je na več faz, ki so precej podobne zagonu običajnega računalniškega sistema. Vsaka od faz je v tem poglavju opisana. Shemo zagona lahko vidimo na sliki 3.1.



Slika 3.1: Shema zagona OS GNU/Linux na vgrajenem sistemu.

3.1 Boot ROM

Boot ROM je program, namenjen osnovni konfiguraciji sistema in prenosu nalagalnika prve stopnje (ang. First stage bootloader) iz zagonskega medija (ang. boot device) v delovni pomnilnik [39]. Običajno je zapisan v interni ROM pomnilnik in uporabnik same kode ne more urejati. Nastavljiv je v manjši meri preko določenih vhodnih signalov sistema SoC. Pri vgrajenih sistemih bi lahko rekli, da ima podobno vlogo kot BIOS na namiznem (ang. desktop) računalniškem sistemu.

V primeru Xilinx XC7Z020 SoC je Boot ROM tovarniško zapisan v interni ROM pomnilnik SoC [39]. V manjši meri je nastavljiv preko tako imenovanih MODE vhodov sistema SoC. Programski tok programa Boot ROM je v grobem sledeč:

1. Inicializacija zunanjih naprav, ki se lahko obnašajo kot nalagalni medij.

2. Določitev nalagalnega medija glede na stanja zunanjih MODE vhodov procesorja.

3. Kopiranje nalagalnika prve stopnje iz nalagalnega medija v interni RAM pomnilnik.

3.2 Das U-boot

Nalagalnikov prve (in druge) stopnje je danes tudi na področju vgrajenih sistemov kar nekaj. Eden od njih je tudi odprtokodni Das U-boot, ki je bil uporabljen v tej diplomski nalogi. Izbran je bil iz več razlogov, med drugim, ker je v sami skupnosti razvijalcev dobro podprt in že sam po sebi podpira veliko vgrajenih sistemov, med njimi tudi ZedBoard. U-boot je razdeljen na dva ločena nalagalnika: SPL ter polni U-boot.

3.2.1 SPL

SPL (Second program loader) predstavlja nalagalnik prve stopnje [6]. Zadolžen je za nalaganje in zagon nalagalnika druge stopnje (polni U-boot).

SPL se zdi nesmiselen, saj je polni U-boot nalagalnik prav tako lahko nalagalnik prve stopnje. Na tej točki se lahko vprašamo, zakaj potemtakem sploh potrebujemo SPL? Običajno ga potrebujemo zato, ker je že inicializirani delovni pomnilnik premajhen za polni U-boot. V tem primeru je to interni RAM pomnilnik, ki je na Xilinx XC7Z020 SoC sistemu velikosti 256 KB, polni U-boot pa zasede 460 KB pomnilnika. Zato SPL inicializira 512 MB velik DDR3 RAM pomnilnik in tja naloži polni U-boot.

3.2.2 Polni U-boot

Polni U-boot je v tem primeru nalagalnik druge stopnje [6]. Podpira več nalagalnih medijev, med drugim [6]:

- mrežo (protokol TFTP),
- (Q)SPI flash pomnilnik,
- SD kartico.

Ima tudi ukazno vrstico, preko katere lahko nastavljamo parametre U-boot nalgalnika, upravljamo s pomnilniki, itd. Naloga U-boot nalagalnika je v končni fazi pripraviti vse potrebno za zagon operacijskega sistema GNU-/Linux. Iz nalagalnega medija mora tako v delovni pomnilnik prenesti tri elemente operacijskega sistema [23]:

- jedro Linux (ang. Linux kernel),
- drevo naprav (ang. device tree),
- korenski datotečni sistem (ang. root file system).

3.3 GNU/Linux

Jedro predstavlja le del operacijskega sistema, ne pa njegovo celoto. Ime GNU/Linux iz zgodovinskih razlogov prav zaradi tega izhaja iz dejstva, da je Linus Torvalds takratnemu odprtokodnemu projektu GNU dodal le svoje jedro (Linux), vsi ostali gradniki operacijskega sistema pa so bili narejeni s strani projekta GNU [13]. Na samem vgrajenem sistemu poleg jedra potrebujemo še korenski datotečni sistem in drevo naprav.

3.3.1 Korenski datotečni sistem

Korenski datotečni sistem je zelo pomemben element operacijskih sistemov, kot jih poznamo danes. Pod korenski datotečni sistem pa ne spada le sama datotečna struktura na sistemu, ampak še veliko več.

Kot narekuje ena od filozofij Unix sistema: *"Vse je datoteka"* (ang. *"Everything is a file"*) [12], je tudi pri njegovem potomcu, sistemu GNU-/Linux, vse predstavljeno kot datoteka (naprave, gonilniki, moduli, ukazi v lupini ...). Zato je korenski datotečni sistem v primeru GNU/Linux OS še toliko bolj pomemben. Za gradnjo korenskega datotečnega sistema obstaja več okolij. Defacto standard na tem področju je BusyBox, ki je bil uporabljen tudi v tej diplomski nalogi. Kot vsi ostali gradniki je seveda odprtokoden.

BusyBox

BusyBox sistem ne zgradi korenskega datotečnega sistema v celoti, ampak mu do delujočega datotečnega sistema manjka še precej elementov. BusyBox izgradi neko osnovno direktorijsko strukturo in v bin direktorij vnese Unix orodja (ang. Unix utils). Lepota BusyBox sistema je v tem, da so Unix orodja zapakirana v eno samo izvršiljivo datoteko, datoteke Unix orodij pa so tako le simbolične povezave na to izvršljivo datoteko z različnimi parametri [4].

Spodaj so naštete dodatne datoteke, potrebne v korenskem datotečnem sistemu [3].

- Moduli ter zaglavja jedra Linux – v datotečni sistem jih vnesemo z *make install* ukazom (Make okolje jedra).

- Standardna C knjižnica – implementira funkcije (npr. printf) in definicije tipov (npr. niz), definirane v ANSI C standardu.

- Konfiguracijska datoteka inittab – določa obnašanje operacijskega sistema ob določenih dogodkih (npr. zagon sistema). Datoteka v tem primeru določa, da se ob zagonu izvede zagonska skripta rcS.

- Zagonska skripta rcS – je napisana v skriptnem jeziku Bash in določa inicializacijo sistema pred zagonom lupine.

- Tabela datotečnih sistemov fstab – določa druge datotečne sisteme (npr. sysfs, devfs ...) in njihove priklopne točke (ang. mount point) v korenskem datotečnem sistemu. Ti datotečni sistemi se priklopijo na pripadajoče priklopne točke ob klicu ukaza *mount -a* v lupini.

- Skripta, ki določa obnašanje DHCP odjemalca udhcpc (simple.script) – potrebujemo jo za delovanje udhcpc odjemalca.

3.3.2 Drevo naprav

Linux jedro se ne zaveda strojne opreme vgrajenega sistema. Če želi komunicirati s katerokoli izmed naprav, mora zato od nekje pridobiti informacije o napravah, ki jih najdemo na vgrajenem sistemu. V ta namen se vpelje drevo naprav, ki na preprost način v obliki drevesa opisuje strojno opremo na vgrajenem sistemu [10].

Spodaj vidimo primer vnosa v drevesu naprav za serijsko UART povezavo na sistemu ZedBoard.

```
uart1: serial@e0001000 {
    compatible = "xlnx,xuartps", "cdns,uart-r1p8";
    status = "disabled";
    clocks = <&clkc 24>, <&clkc 41>;
    clock-names = "uart_clk", "pclk";
    reg = <0xE0001000 0x1000>;
    interrupts = <0 50 4>;
};
```

Kot vidimo, so opisani kompatibilni gonilniki za napravo (polje compatible), status naprave, urini signali, na katere je naprava povezana, lokacija registrov naprave v pomnilniku (polje reg) ter opis prekinitev naprave.

Možnih je več nivojev takšnih dreves, ki jih potem verižno povezujemo preko *include* direktiv [10]. V tem primeru drevo Zedboard sistema vključuje drevo procesorske družine Zynq-7000 prek include direktive.

Drevo naprav se običajno zgradi pri gradnji jedra Linux s posebnim prevajalnikom, imenovanim device tree compiler (dtc) [10].

3.3.3 Jedro Linux

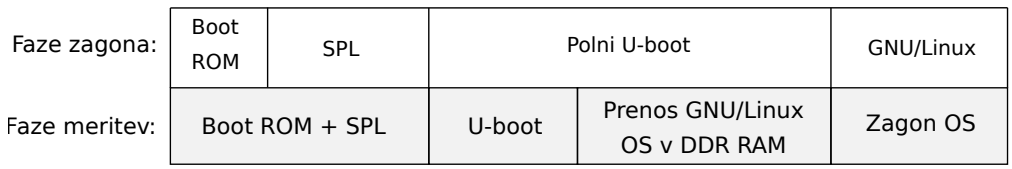
Naloge jedra Linux so, da inicializira naprave s pomočjo drevesa naprav, priklopi ter po potrebi razpakira korenski datotečni sistem in operacijski sistem s pomočjo procesa init pripelje do točke, kjer lahko uporabnik z njim upravlja [24]. To je tudi zadnja faza zagona operacijskega sistema GNU/Linux.

Poglavje 4

Časovna analiza zagona sistema

Ko operacijski sistem GNU/Linux deluje na vgrajenem sistemu, potrebujemo še metodo, s katero lahko dovolj natančno in zanesljivo določimo čas zagona. Tako vidimo tudi natančne časovne pridobitve na vsakem koraku optimizacije.

Preden se lotimo implementacije neke metode, je smiselno meritev zagona razdeliti na več faz. Tako lahko prepoznamo, katera faza je najbolj potratna in jo skušamo optimizirati. Faze meritev se razdelijo podobno kot faze zagona sistema (več o samem zagonu v poglavju 3). Shemo razdelitve vidimo na sliki 4.1.



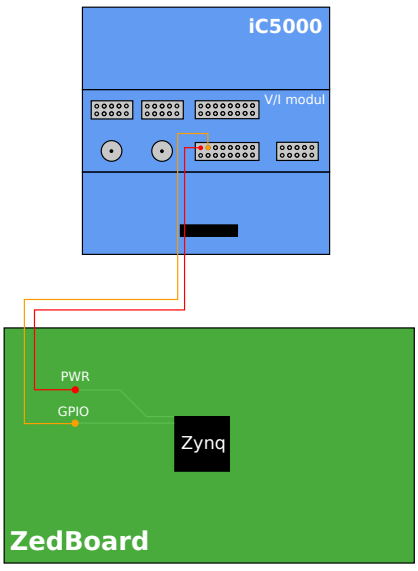
Slika 4.1: Razdelitev časovne meritve zagona na več faz.

4.1 Ročna štoparica

Čeprav je to najbolj trivialna in nenatančna metoda, je uporabna za že obstoječe distribucije operacijskega sistema GNU/Linux. Seveda ni dovolj natančna metoda za merjenje optimizacij, vendar pa tako lahko vidimo, kakšne čase dosega neka že obstoječa distribucija (npr. Arch ARM Linux). Tukaj se meri samo skupen čas zagona brez faz.

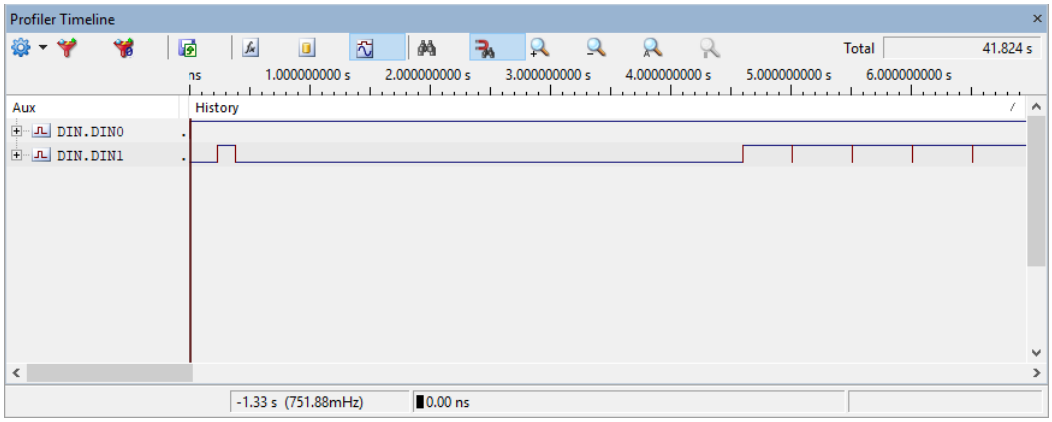
4.2 Preklapljanje stanja na GPIO pinu

Ideja je, da na prehodih med fazami preklopimo digitalno stanje zunanjega splošnonamenskega (GPIO) pina, katerega stanji sta lahko logična 0 ali logična 1. Natančne časovne meritve preklopov GPIO pina pa lahko delamo z razširitvijo orodja iC5000, V/I modulom. Na sliki 4.2 vidimo vezavo orodja iC5000 z ZedBoard sistemom.



Slika 4.2: Vezava iC5000 z ZedBoard sistemom.

En digitalni vhod V/I modula povežemo na GPIO pin, drugega pa na napajalni pin. Napajalni pin opazujemo zato, ker gre njegovo stanje iz logične 0 na logično 1 takoj, ko sistem vklopimo. Tako pridobimo referenco za čas med vklopom vgrajenega sistema in prvim preklopom stanja na GPIO pinu. Na sliki 4.3 vidimo posnetek V/I modula v orodju winIDEA.



Slika 4.3: Posnetek I/O modula v orodju winIDEA.

WinIDEA začne prikazovati dogajanje na digitalnih vhodih ob prvem preklopu kateregakoli od vhodov, v našem primeru torej ob preklopu napajalnega pina (DIN.DIN0). Tako dobimo naslednje čase posameznih faz.

- **Boot ROM + SPL**: čas med preklopom napajalnega pina in prvim preklopom GPIO pina (DIN.DIN1).
- **U-Boot**: čas med prvim in drugim preklopom GPIO pina.
- **Prenos GNU/Linux OS v DDR RAM**: čas med drugim in tretjim preklopom GPIO pina.

Ta metoda zahteva poseg v kodo nalagalnika do te mere, da na posameznih prehodih dodamo kodo za preklop GPIO pina. To seveda ni zahtevna operacija, saj predstavlja le nekaj vpisov v pomnilniško preslikane registre GPIO naprave.

Manjka meritev faze zagona operacijskega sistema. Na tej točki se zalomi, saj imamo lahko zaradi zaščite pomnilnika s strani Linux jedra kar precej težav s pomnilniškimi vpisi.

4.3 Branje časovnika

Težavi z metodo preklapljanja stanja na GPIO pinu sta dve.

1. Pisalni dostop do pomnilnika v OS GNU/Linux (ta problem je sicer rešljiv).

2. Potrebna je dodatna strojna in programska oprema za časovno merjenje preklopov stanj GPIO pina.

Želimo metodo, ki bo enostavna za implementacijo in ne bo potrebovala zunanje strojne ali programske opreme. Možnost, ki se tukaj ponudi, je vzorčenje časovnika. Časovnik vzorčimo ob vsakem prehodu med fazami in glede na znano frekvenco časovnika lahko preprosto določimo čas med vzorčenimi vrednostmi. Bralni dostop do pomnilnika prek Linux jedra običajno ni problematičen.

Takšen časovnik v Xilinx Zynq XC7Z020 SoC je ARM-ov 64-bitni globalni časovnik, ki ga najdemo na vseh ARM Cortex A9 procesorskih enotah [1]. Ta časovnik začne delovati takoj ob zagonu sistema s konstantno frekvenco $333.\overline{3}$ MHz. Prehod iz prejšnje metode merjenja (Preklapljanje stanja na GPIO pinu) na to je precej trivialen. Zamenjati moramo le kodo za preklop GPIO pina z bralnim dostopom do pomnilniško preslikane vrednosti časovnika.

Seveda je ta metoda specifična glede na procesor, koda ni generična in delovala bo le na ARM Cortex A9 procesorskih enotah. Vseeno pa sama implementacija ne bi smela povzročati veliko težav na drugih procesorjih. Navsezadnje imajo vsi vgrajeni sistemi nek časovnik.

4.3.1 Prenos vzorčenih časov iz nalagalnika v Linux jedro

Ko gre Linux jedro v izvajanje, izgubimo sklad in kopico nalagalnika. Posledično izgubimo tudi vse spremenljivke nalagalnika, oziroma se Linux ne zaveda, kje v pomnilniku so. Zato potrebujemo nek mehanizem, da vzorčene vrednosti časovnika prenesemo iz nalagalnika v Linux jedro.

ARM tags

Včasih je bil ta mehanizem na ARM procesorjih ARM Tags (ATAGS). ATAGS je posebna struktura, kamor so se vnesle dodatne informacije za Linux jedro. Nalagalnik je to strukturo dal na neko lokacijo v pomnilnik, to lokacijo zapisal v interni register procesorja (register R2) in predal kontrolo jedru. Jedro je nato iz lokacije, zapisane v R2, prebralo to strukturo. V ATAGS strukturi so se prenašale le dodatne informacije, celoten opis strojne opreme je bil namreč vsebovan že v samem jedru. Mehanizem ATAGS je sicer še podprt, ampak ni priporočljiv in je za uporabo tudi nemogoč, če že uporabljamo njegovega naslednika, drevo naprav [10].

Drevo naprav

Zaradi želje po večji prenosljivosti samega jedra se je opis strojne opreme kasneje odstranil iz jedra in prenesel v posebno strukturo, imenovano drevo naprav. Podrobnejši opis drevesa naprav je na voljo v poglavju 3.3.2. Ta mehanizem je s časom postal splošen za vse vgrajene sisteme [10]. Drevo naprav pa ne vsebuje le opisa strojne opreme, ampak omogoča tudi prenos dodatnih informacij jedru, kot so recimo vzorčeni časi časovnika.

Poglavje 5

Optimizacija časa zagona

V tem poglavju so opisani koraki optimizacije časa zagona GNU/Linux sistema. Natančne časovne pridobitve pri vsaki od uporabljenih optimizacij so na voljo v prilogi A.

5.1 Optimizacije nalagalnika

5.1.1 Premestitev nalagalnika v pomnilniku

Bazni naslov nalagalnika določa, kje v delovnem pomnilniku se nalagalnik izvaja. U-boot nalagalniku lahko kot bazni naslov določimo katerikoli naslov v delovnem pomnilniku. To storimo pri gradnji nalagalnika z nastavitvijo konstante SYS_TEXT_BASE [7] na željeni bazni naslov, a tu nastane problem. U-boot namreč zahteva, da se nahaja na koncu delovnega pomnilnika, saj tako zagotovi kar se da velik sklenjen del delovnega pomnilnika aplikaciji (GNU/Linux OS) [36]. Če temu ni tako, se U-boot med izvajanjem v celoti prekopira na konec delovnega pomnilnika. To pa trati nepotreben čas. Z ustrezno vrednostjo nastavitve SYS_TEXT_BASE se lahko temu izognemo.

5.1.2 Tihi zagon nalagalnika

Naslednji korak optimizacije časa zagona nalagalnika je tihi zagon nalagalnika. To pomeni, da nalagalnik med samim izvajanjem ničesar ne izpisuje na terminal. Konkretno za U-boot nalagalnik to postorimo z definicijami naslednjih konstant pri gradnji [9]:

- CONFIG_SILENT_CONSOLE – definiramo tihi zagon.
- CONFIG_SILENT_U_BOOT_ONLY – definiramo tihi zagon le za U-boot nalagalnik, sicer nam tudi pod Linux sistemom serijski terminal ne deluje.

5.1.3 Optimizacija prenosa OS GNU/Linux iz QSPI flash pomnilnika

Pomembna faza zagona je tudi populacija delovnega pomnilnika z vsemi elementi operacijskega sistema GNU/Linux, kar je delo nalagalnika. Običajno je ta faza zelo potratna, saj so velikosti vseh konstruktov nekega prirejenega OS GNU/Linux za vgrajene sisteme skupaj nekje v rangu 10 MB.

Če želimo izkoristiti vse, kar nam ponuja QSPI flash pomnilnik glede hitrosti prenosa, moramo biti pozorni na dve spremenljivki:

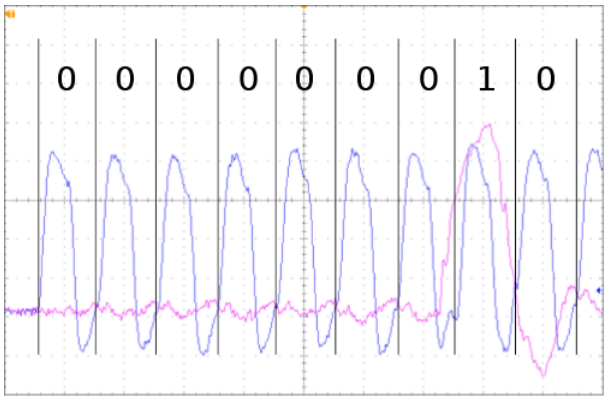
- frekvenco ure, ki določa hitrost prenosa podatkov,
- način prenosa (število uporabljenih linij).

Frekvenca ure prenosa

Tako QSPI flash pomnilnik kot SoC vsak zase definirata najvišjo možno frekvenco ure prenosa. Najmanjša izmed teh frekvenc pa določa najvišjo možno frekvenco ure prenosa med napravama.

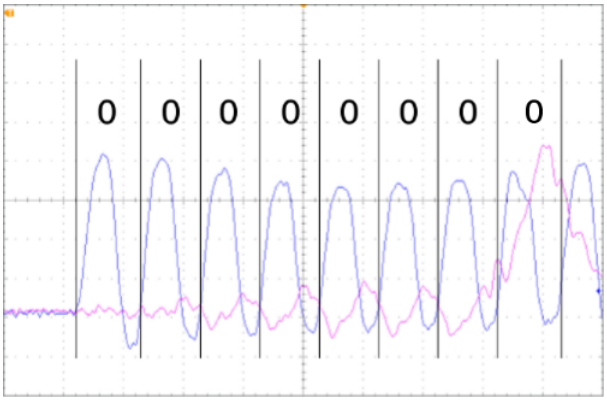
V primeru ZedBoard vgrajenega sistema s Xilinx Zynq XC7Z020 SoC [39] ter Spansion S25FL256S QSPI flash pomnilnikom [34] je najvišja možna frekvenca ure 100 MHz. Pri tako visoki frekvenci pa so se začele pojavljati težave na linijah.

Na sliki 5.1 vidimo posnetek odgovora pomnilnika na poizvedbo po identifikaciji pomnilnika (JEDEC ID). Posnetek je bil narejen z osciloskopom. Modri signal predstavlja urin signal, vijolični pa podatkovni signal. Frekvenca ure je v tem primeru 50 MHz in komunikacija deluje pravilno.



Slika 5.1: Posnetek signalov na QSPI linijah pri 50 MHz.

S slike lahko v grobem razpoznamo 7 zaporednih ničel na liniji ter eno enico, nato pa spet ničle. Na sliki 5.2 pa vidimo odgovor na poizvedbo po identifikaciji pomnilnika pri 100 MHz frekvenci urinega signala.



Slika 5.2: Posnetek signalov na QSPI linijah pri 100 MHz.

Pri 100 MHz je očitno vpliv elektromagnetnih motenj na podatkovni liniji bistveno večji. Povrh vsega pa osmi bit, ki bi ga morali zaznati kot logično 1, zaznamo kot logično 0, saj pride z zamikom. Tako procesor zazna pokvarjen JEDEC ID s strani pomnilnika in prekine komunikacijo, ker pomnilnika ne more identificirati.

Pri tako visokih frekvencah, kot je 100 MHz, imajo motnje še toliko večji vpliv na linijo. Tako se v primeru ZedBoard vgrajenega sistema zadovoljimo s 50 MHz urinim signalom, saj vmesnih frekvenc ne moremo generirati s strani procesorja oziroma bi morali krepko poseči v samo kodo U-boot nalagalnika. Frekvenco definiramo s konstanto CONFIG_SF_DEFAULT_SPEED pri gradnji U-boot nalagalnika [6].

Način prenosa (število uporabljenih linij)

Prenos podatkov po QSPI protokolu lahko poteka po eni, dveh ali štirih linijah. Potrebno je preveriti koliko linij pomnilnik in procesor sploh podpirata ter koliko linij je speljanih na tiskanem vezju med njima. Koliko linij se uporablja za prenos, določa ukaz, ki ga procesor pošlje pomnilniku. V tabeli 5.1 vidimo bralne ukaze glede na porabljeno število linij in imena njihovih nastavitev v U-boot konfiguraciji gradnje [6] [34].

Ukaz	Koda ukaza	Št. uporabljenih linij	U-boot nastavitev
READ	0x03	1	SPI_RX_SLOW
FAST_READ	0x0B	1	Privzeto
DOR	0x3B	2	SPI_RX_DUAL
QOR	0x6B	4	SPI_RX_QUAD

Tabela 5.1: Bralni ukazi za QSPI pomnilnik.

Če želimo uporabiti določen ukaz za prenos, ga definiramo v konstanti CONFIG_SF_DEFAULT_MODE, v tem primeru SPI_RX_QUAD (prenos po 4 linijah) [6].

5.1.4 U-boot v načinu Falcon

U-boot je razdeljen na dva dela: SPL ter polni U-boot (več o tem v poglavju 3.2). Razvijalci U-boot nalagalnika so nato prenesli funkcijo nalaganja GNU-/Linux OS iz polnega U-boot nalgalnika tudi v SPL (nalagalnik prve stopnje) in to funkcionalnost poimenovali način Falcon [8]. V tem načinu U-boot SPL direktno naloži gradnike GNU/Linux OS v delovni pomnilnik in požene jedro. Tako preskočimo nalagalnik druge stopnje.

Delovanje Falcon načina pri U-boot SPL nalagalniku

Če želimo Falcon način uporabljati, moramo biti najprej pri gradnji pozorni na naslednje nastavitve [8].

- CONFIG_CMD_SPL – omogoči ukaz za generiranje argumentov Linux jedra v U-boot ukazni vrstici.
- CONFIG_SPL_OS_BOOT – omogoči Falcon način v U-boot SPL.
- CONFIG_SYS_SPI_KERNEL_OFFS – lokacija Linux jedra v QSPI pomnilniku.
- CONFIG_SYS_SPI_ARGS_OFFS – lokacija argumentov Linux jedra v QSPI pomnilniku.
- CONFIG_SYS_SPI_ARGS_SIZE – velikost argumentov Linux jedra.
- CONFIG_SYS_SPL_ARGS_ADDR – naslov v delovnem pomnilniku, kamor so argumenti Linux jedra prenešeni.

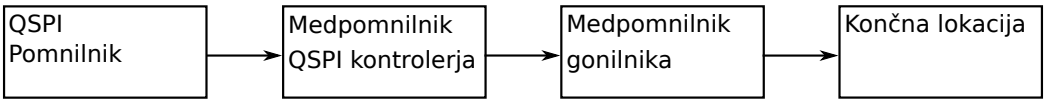
SPL dobi argumente jedra Linux na točno določeni lokaciji v QSPI flash pomnilniku, definirani z nastavitvami gradnje. Te argumente generiramo s pomočjo polnega U-boot nalagalnika z ukazom spl export. Zato ob vsaki spremembi argumentov še vedno potrebujemo polni U-boot nalagalnik. U-boot v svoji dokumentaciji za Falcon način zato zahteva, da se implementira mehanizem, preko katerega bo SPL nalagalnik ločil, ali naj v danem trenutku naloži GNU/Linux OS ali polni U-boot [8].

Mesto za implemetacijo takšnega mehanizma v U-boot kodi ni določeno. Najprimernejša se zdi funkcija spl_start_uboot, ki je del U-boot nalagalnika. Ta funkcija vrne 0, če mora SPL nalagalnik pognati Linux, in 1, če mora SPL pognati polni U-boot. V primeru ZedBoard vgrajenega sistema se na tem mestu preverja stanje enega izmed mostičkov.

Težave pri uporabi Falcon načina

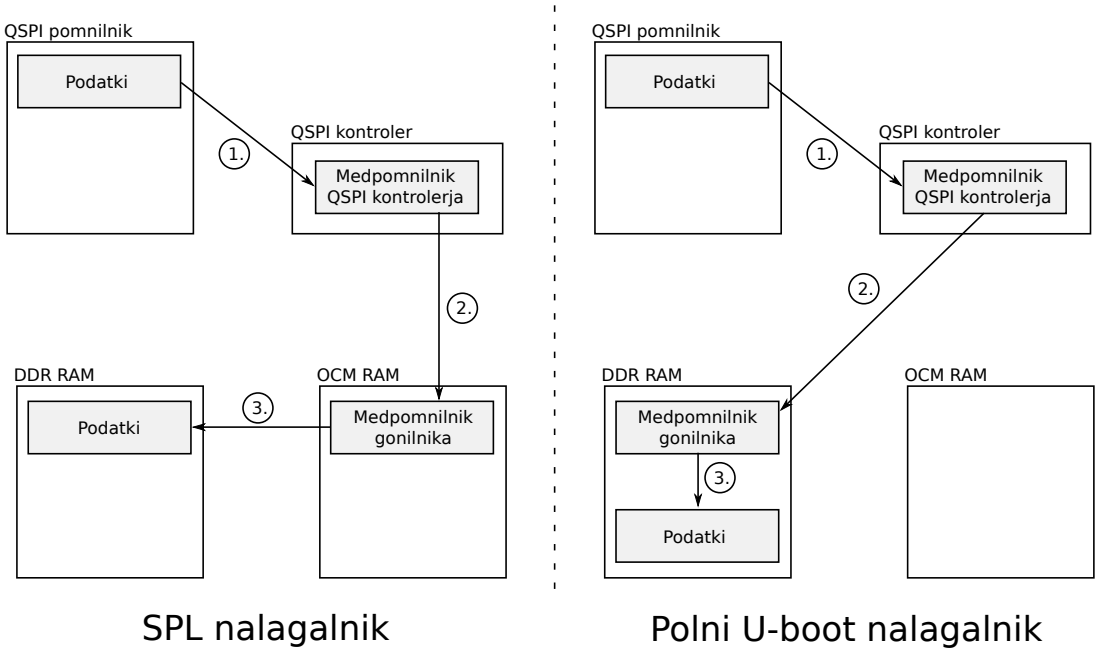
Izkaže se, da U-boot SPL porabi trikrat več časa za prenos GNU/Linux OS iz QSPI flash pomnilnika v DDR RAM pomnilnik kot pa polni U-boot nalagalnik. Zanimivo je to, da polni U-boot in SPL uporabljata isti gonilnik QSPI flash pomnilnika in isto konfiguracijo, zato bi pričakovali enako hitrost prenosa.

Težava se pojavi v gonilniku. Da to razložimo, si najprej poglejmo, kako deluje gonilnik. Delovanje gonilnika ponazorimo s sliko 5.3, kjer vidimo potek prenosa podatkov prek več pomnilnikov ter medpomnilnikov.



Slika 5.3: Delovanje QSPI gonilnika.

Natančneje si poglejmo medpomnilnik gonilnika, saj ostali trije niso zanimivi, ker so v obeh primerih (polni U-boot ter SPL) na istem mestu. Medpomnilnik gonilnika je za razliko od ostalih lokalna spremenljivka gonilnika in se zato nahaja v pomnilniku, kjer se gonilnik izvaja. V primeru polnega U-boot nalagalnika je to DDR RAM pomnilnik, v primeru SPL nalagalnika pa interni RAM pomnilnik na SoC (OCM RAM). Na sliki 5.4 vidimo pomnilniško shemo delovanja QSPI gonilnika v SPL in polnem U-boot nalagalniku.



Slika 5.4: Pomnilniška shema delovanja QSPI gonilnika.

Kot vidimo, sta kritični poti prenosa dve, in sicer druga in tretja. V tabeli 5.2 vidimo meritve časa prenosa štirih bajtov po vsaki od kritičnih poti.

	SPL (OCM RAM) [ns]	U-boot (DDR RAM) [ns]
Prenos 1	231	165
Prenos 2	441	93
Vsota	672	258

Tabela 5.2: Povprečen čas prenosa štirih bajtov iz QSPI flash pomnilnika.

Tako pojasnimo trikrat daljši čas prenosa podatkov iz QSPI flash pomnilnika v DDR RAM pomnilnik pri SPL nalagalniku. Ker nam zato Falcon način prinese slabše rezultate, te optimizacije ne upoštevamo v končnem produktu.

5.2 Optimizacije GNU/Linux OS

5.2.1 Čiščenje konfiguracije gradnje Linux jedra

Privzeta konfiguracija gradnje jedra, uporabljena za ZedBoard vgrajeni sistem, je globalna za ARMv7 arhitekturo [41]. To pomeni, da je zgrajene veliko navlake in nepotrebnih gonilnikov naprav, ki jih na našem vgrajenem sistemu ne potrebujemo ali pa sploh nimamo. Zato lahko s preprostim pregledom in odstranitvijo nepotrebnih stvari v konfiguraciji gradnje Linux jedra veliko pridobimo na velikosti jedra in tudi samem času zagona. Nekaj časa seveda posredno pridobimo zaradi manjše velikosti Linux jedra (manj časa porabimo za prenos jedra iz nalagalnega medija v delovni pomnilnik).

Stvari, ki jih lahko spremenimo, da prečistimo konfiguracijo gradnje:

- Odstranjevanje nepotrebnih gonilnikov naprav.
- Odstranjevanje podpore nekaterim perifernim napravam (tipkovnica, zaslon na dotik, računalniška miška ...), saj s terminalom operacijskega sistema komuniciramo preko serijske UART povezave.
- Odstranjevanje podpore vsem vgrajenim sistemom (SoC) razen ciljnemu.
- Odstranjevanje podpore virtualizaciji, saj je ne potrebujemo na vgrajenem sistemu.
- Odstranjevanje podpore ATAGS sistemu, saj v primeru da, uporabljamo drevo naprav, ne moremo uporabljati tudi ATAGS sistema. Več na to temo v poglavju 4.3.1.

5.2.2 Tihi zagon Linux jedra

Kot pri nalagalniku lahko precej na času zagona pridobimo, če izklopimo izpise jedra na terminalu med zagonom. To storimo za Linux jedro tako, da mu ob zagonu kot enega od argumentov damo tudi argument quiet [17]. S tem argumentom dosežemo to, da Linux jedro med zagonom na terminal izpiše le kritična sporočila (v primeru, da gre kaj narobe).

5.2.3 Odstranitev razhroščevalnih simbolov

Linux jedro je privzeto zgrajeno tako, da vsebuje razhroščevalne simbole. Tega za končni izdelek ne potrebujemo in predstavlja odvečno navlako. Nekaj nastavitev na to temo je opisanih v naslednjih podpoglavjih.

CONFIG_BUG

S to nastavitvijo omogočimo podporo za funkciji jedra BUG() ter WARN(). Funkcija BUG() se pokliče, ko gre v jedru nekaj narobe, in je zadolžena za to, da izpiše sled sklada (ang. stack trace) ter vsebino registrov procesorja. Poleg tega še terminira proces, ki je sprožil klic funkcije BUG() [19].

Če to nastavitev onemogočimo, zmanjšamo velikost jedra in posledično čas zagona, izgubimo pa predvsem izpis funkcije BUG(). Ker vgrajeni sistemi običajno niso neprestano priklopljeni na serijski terminal, je smiselno to možnost onemogočiti, dokler ne vzpostavimo nekega sistema beleženja takšnih izpisov za kasnejšo analizo.

CONFIG_DEBUG_FS

S to nastavitvijo določimo, da se v jedru izgradi psevdo datotečni sistem debugfs. Ta datotečni sistem služi kot kanal med prostorom jedra (ang. kernel space) ter uporabniškim prostorom (ang. user space) za potrebe razhroščevanja. Viden je iz uporabniškega prostora ter prostora jedra [5]. Tako lahko razvijalci iz uporabniškega prostora dostopajo do razhroščevalnih informacij svojega modula. Tega v končnem izdelku ne potrebujemo, saj zaseda le dodaten prostor.

CONFIG_KALLSYMS

Ta nastavitev določa, ali so razhroščevalni simboli jedra vgrajeni v jedro [18]. Če to možnost onemogočimo, se znebimo prostora, ki ga zasedajo razhroščevalni simboli in tako zmanjšamo velikost jedra ter posledično pospešimo zagon vgrajenega sistema.

5.2.4 Večjedrni zagon Linux jedra

V primeru ZedBoard vgrajenega sistema imamo na voljo dve procesorski jedri, zato je koristno izmeriti, kakšen je čas zagona pri večjedrnem zagonu v primerjavi z enojedrnim zagonom Linux jedra.

Več jeder se v Linux jedru zbudi ob zagonu v primeru, da je Linux jedro zgrajeno z omogočeno nastavitvijo CONFIG_SMP [28]. Izkaže se, da zagon Linux jedra ni paraleliziran na več procesorskih jeder. Za inicializacijo sekundarnega procesorskega jedra na ZedBoard vgrajenem sistemu pa Linux jedro porabi približno 120 ms.

Potrebno je premisliti, ali smo pripravljeni žrtvovati 120 ms ob zagonu sistema in potem imeti na voljo za delo dve procesorski jedri, ki se jih Linux jedro zaveda. Kasneje je lahko namreč inicializacija ostalih procesorskih jeder težavna. Odvisno je od končne aplikacije, ki se bo izvajala na GNU/Linux OS. Če bo ta izkoriščala več jeder, je smiselno večjedrni zagon omogočiti. S stališča samega časa zagona Linux jedra pa je sicer bolje, da je ta možnost onemogočena.

5.2.5 Alokacijski algoritmi Slab

Pri Slab alokacijskih algoritmih govorimo o pomnilniški alokaciji objektov jedra. V osnovi poznamo tri algoritme, SLOB, SLAB ter SLUB [33].

SLOB

Je najstarejši in najbolj neučinkovit. Primeren je za vgrajene sisteme, ker sam algoritem ne zahteva veliko dodatnih resursov za delovanje. Deluje po preprostem principu že znanega first-fit algoritma, kjer argoritem postavi objekt za alokacijo v prvi dovolj velik prosti kos pomnilnika. Algoritem je močno nagnjen k fragmentaciji [33]. Čeprav je za sisteme z manjšim pomnilnikom primeren, ni dovolj dober za zmogljivejše vgrajene sisteme, kjer imamo dovolj pomnilnika, da si lahko privoščimo prostorsko dražji alokacijski algoritem.

SLAB

SLAB algoritem je eden od Slab alokacijskih algoritmov. Ker precej objektov Linux jedra zahteva enako velikost prostora, lahko rezerviramo nek del pomnilnika, ki je razdeljen na enako velike bloke. Algoritem za takšne bloke vodi evidenco, kje v pomnilniku je kateri izmed njih prost. Vsako procesorsko jedro ima vsaj eno vrsto. Zato, da se prečisti zastarele objekte iz vrst, vsaki 2 sekundi SLAB alokator pregleda vse vrste, kar pa je lahko precej potratna operacija [33] . Vseeno se izkaže, da precej pospešimo delovanje Slab algoritma, vendar pa vzame več prostora za samo delovanje.

SLUB

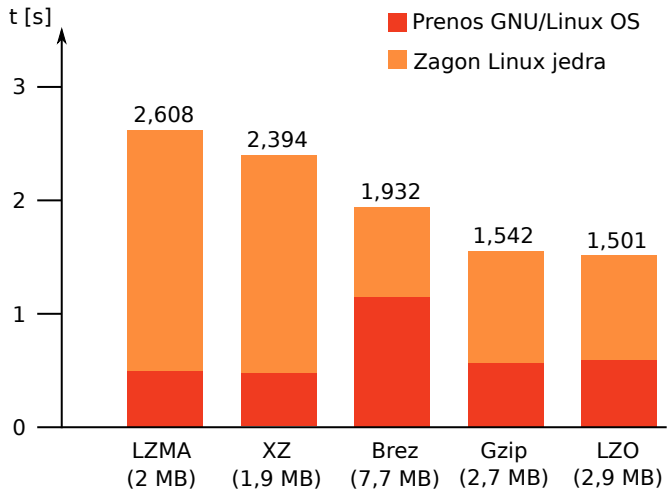
Gre predvsem za nadgradnjo in popravke SLAB algoritma. To se še posebej pozna pri sistemih z veliko procesorskimi jedri, saj se popolnoma znebimo sistema vrst in je za evidenco blokov, ki čakajo na alokacijo, uporabljen povezani seznam. Znebimo pa se tudi osveževanja seznamov s periodo dveh sekund [33].

Sklep

Glede časa zagona sistema je SLAB algoritem najboljša izbira (za odtenek boljša od SLUB algoritma). Prednosti SLUB algoritma ne prepoznamo, ker imamo samo eno aktivno procesorsko jedro in se nam na tej točki GNU/Linux požene že v manj kot dveh sekundah. Je pa vseeno smiseln izbor SLUB algoritma, saj porabi manj prostora za delovanje, manj režije in je novejši. Izguba pri času zagona sistema je v primerjavi s SLAB algoritmom nekaj deset milisekund.

5.2.6 Kompresija Linux jedra

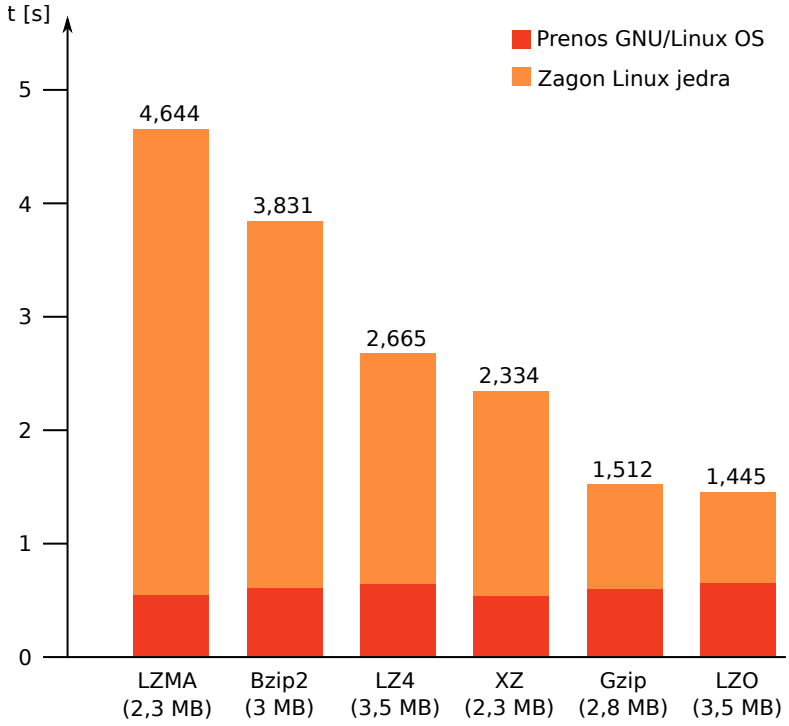
Preden ga poženemo, je Linux jedro lahko kompresirano in se potem razpihne po pomnilniškem prostoru. Običajno je to tudi privzeta konfiguracija. Izbrati moramo takšen kompresijski algoritem, ki ponuja najboljše razmerje med velikostjo kompresiranega jedra ter časom razpakiranja. Rezultati so predstavljeni na sliki 5.5.



Slika 5.5: Primerjava hitrosti zagona sistema pri različnih kompresijskih algoritmih Linux jedra.

5.2.7 Kompresija korenskega datotečnega sistema

Korenski datotečni sistem v obliki initrd (initial ramdisk) je pred uporabo kompresiran. Smiselno je primerjati podprte kompresijske algoritme, čeprav lahko pričakujemo podobne rezultate kot pri kompresiji Linux jedra. Na sliki 5.6 lahko vidimo čase zagona GNU/Linux OS pri podprtih kompresijskih metodah za korenski datotečni sistem v obliki initrd.



Slika 5.6: Primerjava hitrosti zagona sistema pri različnih kompresijskih algoritmih datotečnega sistema.

Rezultati so podobni kot pri jedru in vidimo lahko, da kompresijski algoritem LZO zopet dosega najboljše rezultate, čeprav ima najslabše kompresijsko razmerje.

Ko izberemo kompresijsko metodo datotečnega sistema, je smiselno podporo za ostale kompresijske algoritme izključiti iz gradnje jedra, saj tako prihranimo nekaj prostora.

5.2.8 Tip korenskega datotečnega sistema

Poznamo več različnih tipov datotečnega sistema. Na Windows sistemih srečamo NTFS ter FAT, v GNU/Linux svetu pa EXT2, EXT3, EXT4 ter mnoge druge. Smiselno je pregledati nekaj najpogostejših in izbrati najboljšega za naše potrebe.

EXT2

EXT2 je drugi iz družine Extended datotečnih sistemov, ki je že nekoliko star, ampak še vedno uporaben. Manjka mu nekaj naprednih funkcij, kot jih poznamo pri modernejših tipih, zato je smiselno premisliti, kaj na našemu sistemu sploh potrebujemo [25]. Primanjkuje mu predvsem dnevniška funkcija (ang. journaling).

EXT3

Kot lahko iz imena razpoznamo, gre za naslednjika EXT2 tipa datotečnega sistema. Njegova bistvena prednost je pridobitev dnevniške funkcije datotečnega sistema, poleg tega pa še nekaj manjših optimizacij [26].

EXT4

Gre za najnovejši tip datotečnega sistema iz družine EXT, ki doda še nekaj modernejših funkcionalnosti, kot sta boljše obravnavanje fragmentacije ter podpora velikim datotečnim sistemom (večjim od 16 TB) [27].

Sklep

Kar se tiče časa zagona, se najbolje odreže EXT2 datotečni sistem, kar je logično, saj je tudi najbolj preprost. Smiselna se zdi izbira med EXT2 ter EXT3 datotečnim sistemom, saj se dodatne možnosti EXT4 datotečnega sistema nekako ne zdijo primerne za svet vgrajenih sistemov. EXT3 datotečni sistem pa izberemo, če si v sistemu želimo imeti dnevniško funkcijo.

5.2.9 Standardna C knjižnica

Standardna C knjižnica implementira funkcije (npr. printf) ter podatkovne tipe (npr. string), določene po ISO C standardu [29]. Ta knjižnica mora biti prisotna v GNU/Linux OS, saj na njej sloni veliko funkcionalnosti samega operacijskega sistema (npr. ukazi v lupini).

Implementacij standardne C knjižnice je veliko, najbolj razširjena med njimi je GNU Glibc. Je izjemno bogata ter izpopolnjena, zato jo je smiselno primerjati z implementacijo, primernejšo za vgrajene sisteme. Primer takšne implementacije je Musl.

Musl implementacija standardne C knjižnice in BusyBox pa na žalost med seboj nista najbolj kompatibilna. Na uradni strani Musl knjižnice sicer predlagajo urejanje zaglavnih datotek Linux jedra za delovanje knjižnice v povezavi z BusyBox sistemom [30], a temu ni tako, če BusyBox sistem prevedemo statično in ne dinamično [31].

Primerjava med Musl standardno C knjižnico ter Glibc je predstavljena v tabeli 5.3.

	Velikost korenskega dat. sis. [MB]	Čas zagona [s]
GNU Glibc	3,527	1,607
Musl	2,883	1,531
Sprememba	-0,644	-0.076

Tabela 5.3: Primerjava GNU Glibc ter Musl standardne C knjižnice.

Manjšanje standardne C knjižnice

Če uporabljamo dinamične knjižnice na ciljnem sistemu, jih je smiselno z orodjem strip zmanjšati. To orodje odstrani vse odvečne stvari iz knjižnice, vključno z razhroščevalnimi simboli [35].

5.2.10 Začetna populacija /dev direktorija

Direkotrij */dev* v GNU/Linux OS vsebuje datoteke, preko katerih so dostopne vse naprave na sistemu. Za populacijo tega direktorija moramo na našem sistemu nekako poskrbeti. Rešitvi na tem področju sta dve: storitev *mdev* ter psevdo datotečni sistem *devtmpfs*.

Mdev

Mdev je okrnjena inačica veliko bolj poznane storitve udev, namenjena vgrajenim sistemom. Njene funkcionalnosti vključujejo tudi začetno populacijo */dev* direktorija po zagonu jedra. V končni fazi po zagonu jedra poišče vse trenutno obstoječe naprave v sistemu ter jih doda v */dev* direkotrij.

Devtmpfs

Devtmpfs datotečni sistem se med zagonom napolni z vsemi napravami, ki ustvarijo [11]. Linux jedro pa ga lahko kasneje med samim zagonom priklopi (ang. mount) na */dev* direktorij.

Sklep

Devtmpfs se izkaže za bistveno hitrejšo rešitev in z njo privarčujemo nekaj sto milisekund. Za aktivno osveževanje */dev* direkotrija med delovanjem pa poskrbi storitev mdev.

5.2.11 Zaglavja jedra, moduli ter strojno-programska oprema

Ponavadi ob izgradnji Linux jedra dodamo v datotečni sistem še zaglavja jedra (ang. kernel headers), izgrajene module ter strojno-programsko opremo (ang. firmware). Ti konstrukti zasedajo kar precejšen del prostora v korenskem datotečnem sistemu, zato je potrebno premisliti in raziskati, kaj sploh ponujajo.

Zaglavja jedra

Zaglavlja jedra predstavljajo vmesnike med komponentami Linux jedra ter vmesnike med uporabniškim prostorom ter prostorom jedra [22]. Pomembna so samo, če želimo na ciljnem sistemu prevesti neko kodo, ki te vmesnike uporablja, kar pa v našem primeru ne velja. Za ta primer zato zaglavij jedra ni potrebno vnašati v korenski datotečni sistem.

Moduli jedra

V to kategorijo spada vse, kar je bilo pri gradnji jedra zgrajeno kot samostojni modul. Ker so v tem primeru vsi gonilniki in uporabljeni moduli statično prevedeni v Linux jedro, nam samostoječih modulov prav tako ni potrebno vnašati v korenski datotečni sistem.

Strojno-programska oprema (ang. firmware)

Sem spadajo naprave, ki potrebujejo poleg gonilnika še neko dodatno kodo na nižjem nivoju. Tej kodi rečemo strojno-programska oprema [21]. Takšnih naprav na našem sistemu ni, zato strojno-programske opreme ni smiselno vnašati v korenski datotečni sistem.

5.2.12 Datotečni sistem na persistenčnem pomnilniku

Do sedaj smo vedno imeli datotečni sistem zapakiran v obliki initrd. Problema z initrd datotečnim sistemom sta dva:

- Potrebno ga je od nekod prenesti v delovni pomnilnik.
- Potrebno ga je razpakirati.

Zato je vredno izmeriti čas zagona, če je datotečni sistem že razpakiran na nekem persistenčnem mediju.

Na samem ZedBoard vgrajenem sistemu imamo dve možnosti, in sicer QSPI flash pomnilnik ter SD kartico. Pri QSPI flash pomnilniku se zalomi, saj je s strani Linux jedra zelo slabo podprt in trenutni gonilnik ne deluje po pričakovanjih. Zato izberemo SD kartico.

V tabeli 5.4 vidimo primerjavo v času zagona med korenskim datotečnim sistemom na SD kartici ter v obliki initrd. V obeh primerih je bil uporabljen EXT2 tip datotečnega sistema.

	Prenos GNU/Linux OS [ms]	Zagon Linux jedra [ms]	Skupni čas [ms]
initrd	575	518	1093
SD kartica	323	866	1189

Tabela 5.4: Čas zagona pri datotečnem sistemu na SD kartici ter v initrd obliki.

Ob pogledu na rezultate lahko vidimo, je performanca sistema s korenskim datotečnim sistemom na SD kartici nekoliko slabša v smislu časa zagona sistema. Zdi se, da čas, ki ga pridobimo s tem, da nam korenskega datotečnega sistema ni treba prenašati in razpakirati, izgubimo med zagonom jedra. Razlog za dodatni čas je seveda dejstvo, da je dostop do SD kartice počasnejši od dostopa do delovnega pomnilnika.

Zanimivo je še to, da je čas zagona bistveno bolj nepredvidljiv, če je korenski datotečni sistem na SD kartici. Meritve v povprečju odstopajo kar 70 ms od povprečnega časa zagona.

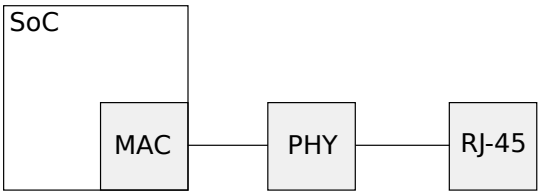
Uporabljena je bila SDHS kartica hitrostnega razreda 10. Po SD standardu to pomeni, da je minimalna hitrost sekvenčnega pisanja 10 MB/s [32]. To je tudi najhitrejša SD kartica, ki jo naš sistem še podpira [39].

5.2.13 Konfiguracija mreže

Za konfiguracijo mreže moramo poskrbeti po samem zagonu jedra, običajno v zagonskih skriptah sistema. Za dinamično konfiguracijo mreže prek DHCP servisa nam BusyBox ponuja uhcpc DHCP odjemalca. Izkaže se, da je konfiguracija mrežnih naprav ter pridobivanje IP naslova prek DHCP strežnika precej potratna operacija. Traja približno 6 sekund. Človek, ki že ima nekaj znanja o mreži in njenih protokolih, bi rekel, da je ta čas absolutno predolg, zato je potrebno raziskati, kje se zatakne.

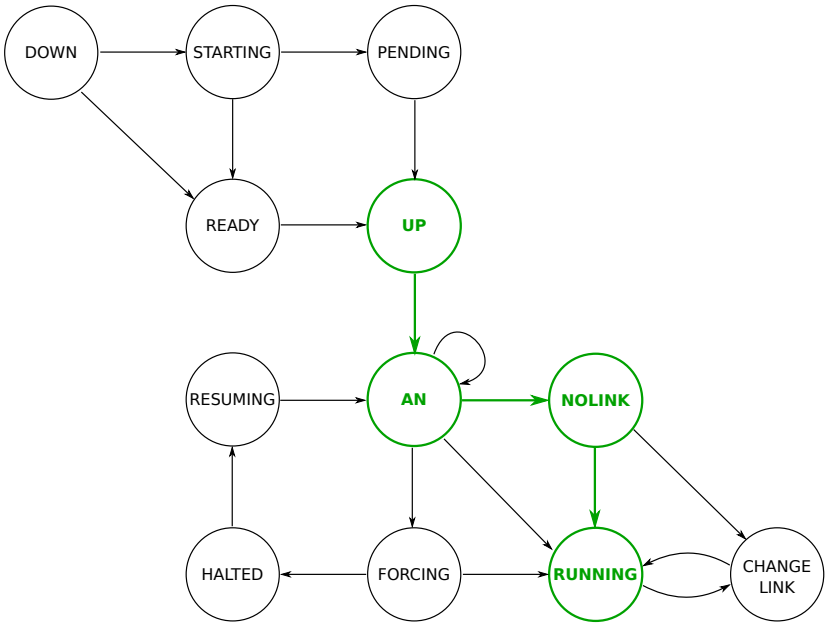
Za primerjavo si najprej poglejmo statično konfiguracijo mreže. Tako identificiramo, ali je težava v pridobivanju IP naslova prek DHCP strežnika ali kje drugje. Izkaže se, da statična konfiguracija mreže porabi približno 3 sekunde, kar je že absolutno preveč. Poleg tega pa dodatne tri sekunde pridejo od internih zakasnitev udhcpc odjemalca, ki pošilja DHCP poizvedbe vsakih nekaj sekund, in ne konstantno.

Tako vidimo, da problem tiči v sami inicializaciji mrežnih komponent. V svetu vgrajenih sistemov v grobem poznamo dve napravi (komponenti), preko katerih je procesor povezan z dejansko fizično plastjo ISO/OSI modela. Shemo komponent vidimo na sliki 5.7.



Slika 5.7: Shema mrežnih komponent.

Prva, v našem primeru ena od perifernih naprav na SoC, je Ethernet kontroler (MAC) druga pa je kontroler na fizičnem nivoju (PHY), ki naredi 'most' med fizičnim konektorjem (RJ-45) ter Ethernet kotrolerjem na čipu. Izkaže se, da je problem v inicializaciji teh naprav. Za boljšo predstavo si poglejmo sliko 5.8, ki predstavlja avtomat stanj gonilnika na fizičnem nivoju ISO/OSI modela v Linux jedru.



Slika 5.8: Avtomat stanj gonilnika na fizičnem nivoju.

V tabeli 5.5 vidimo prehode med stanji, ko želimo mrežo konfigurirati. Do stanja UP pridemo že med samim zagonom jedra.

Časovni žig	Stanje
1	UP
2	AN
3	NOLINK
4	NOLINK
5	RUNNING

Tabela 5.5: Prehod stanj fizičnega nivoja v primeru ZedBoard sistema.

Prvo stanje je AN (Autonegotiation). V tem stanju se kontrolerja, ki sta med seboj povezana z Ethernet vodilom, na fizičnem nivoju dogovorita glede pravil komunikacije na najnižjem nivoju (hitrost, način ...) [38]. Ko se kontrolerja na fizičnem nivoju dogovorita za pravila komunikacije, se na podlagi tega, ali že imamo delujočo povezavo (ang. link up), premaknemo bodisi v stanje RUNNING ali NOLINK. Če se premaknemo v stanje NOLINK, pomeni, da povezava na fizičnem nivoju še ni vzpostavljena, če gremo v stanje RUNNING, pa pomeni, da je. Zakaj traja 3 sekunde, da pridobimo povezavo na fizičnem nivoju, ne vemo.

Dinamično konfiguracijo lahko pospešimo tako, da najprej konfiguriramo mrežo statično z nekim IP naslovom. Kadar je fizična povezava že vzpostavljena, pokličemo udhcpc servis. Tako se znebimo zakasnitve s strani udhcpc odjemalca.

V tabeli 5.6 vidimo natančne meritve konfiguracije mreže na ZedBoard vgrajenem sistemu. Za primerjavo pa si poglejmo čas konfiguracije mreže še na trenutno najnovejšem razhroščevalniku podjetja iSYSTEM Labs d.o.o, iC6000.

Konfiguracija	Trajanje [s]
Statična	3,109
Dinamična	5,815
Hitra dinamična	3,125
iC6000	1,200

Tabela 5.6: Čas konfiguracije mreže.

5.2.14 Končno čiščenje konfiguracije gradnje Linux jedra

Na koncu projekta, ko tudi bolje poznamo samo Linux jedro in to, kar naš sistem ponuja, je smiselno še enkrat prečesati celotno konfiguracijo in poiskati še kaj, česar ne potrebujemo in je vključeno v gradnjo. Pozorni moramo biti predvsem na sledeče:

- Konstrukte Linux jedra, ki so v jedru le za podporo starejšim sistemom (ang. legacy support).
- Razhroščevalne simbole kostruktov Linux jedra.
- Gonilnike, ki jih ne potrebujemo.
- Podporo datotečnim ter psevdo datotečnim sistemom, ki jih ne potrebujemo.

5.3 Rezultati

V tem poglavju je še pregled končnih rezultatov na ZedBoard vgrajenem sistemu.

5.3.1 Obstoječi GNU/Linux proti izgrajenemu

Najprej si poglejmo primerjavo časa zagona ter velikosti že obstoječega Arch ARM Linux operacijskega sistema ter zgrajenega GNU/Linux operacijskega sistema s privzeto konfiguracijo za ARMv7 arhitekturo.

	Velikost [MB]	Čas zagona [s]
Arch ARM Linux	727,127	35
Privzete nastavitve	25,544	8,387
Sprememba	-701,583	-26,613

Tabela 5.7: Primerjava že obstoječega Arch ARM Linux OS z izgrajenim.

Kot vidimo, se lahko že s čisto gradnjo operacijskega sistema GNU/Linux iz izvornih datotek rešimo precej nepotrebnih stvari in pospešimo zagon. Poleg tega pa imamo več manevrskega prostora za optimizacije, saj manipuliramo s konfiguracijo gradnje Linux jedra.

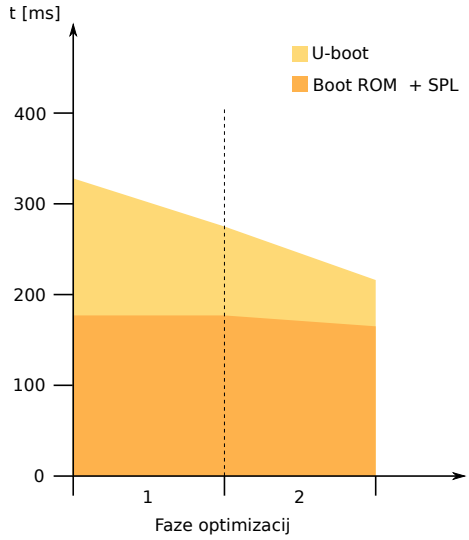
Ob zagonu OS Arch ARM Linux se požene več programov ter servisov, kot pa ob zagonu izgrajenega GNU/Linux OS. Med njimi najdemo:

- servis systemd,
- gonilnik za PCI,
- gonilnik za NFS datotečni sistem.

5.3.2 Optimizacije nalagalnika

Granularnost meritev se tukaj poveča, saj vpeljemo natančnejši sistem merjenja časa zagona. Uvedli smo tri optimizacije nalagalnika, a ena od teh je prinesla rezultate le pri zagonu GNU/Linux OS. Zato si poglejmo le naslednji dve (na grafu na sliki 5.9 od leve proti desni):

1. Premestitev nalagalnika v pomnilniku (več o tem v poglavju 5.1.1)
2. Tihi zagon nalagalnika (več o tem v poglavju 5.1.2)



Slika 5.9: Čas izvajanja U-boot nalagalnika.

Kot je razvidno iz grafa na sliki 5.9, smo relativno precej zmanjšali čas delovanja polnega U-boot nalagalnika, SPL nalagalnika pa niti ne.

U-boot je odličen nalagalnik za vgrajene sisteme, saj jih podpira veliko in večinoma dela brez težav. Vendar pa široka podpora pomeni veliko nivojev abstrakcije v kodi, kar upočasni njegovo delovanje. To se lepo vidi pri preprosti izvedbi ukaza za branje iz QSPI flash pomnilnika, kjer se kliče 10 funkcij do prve, ki komunicira s strojno opremo. Zato predlagam, da se za časovno kritične sisteme razvije nalagalnik, ki bo prirejen za ciljni sistem.

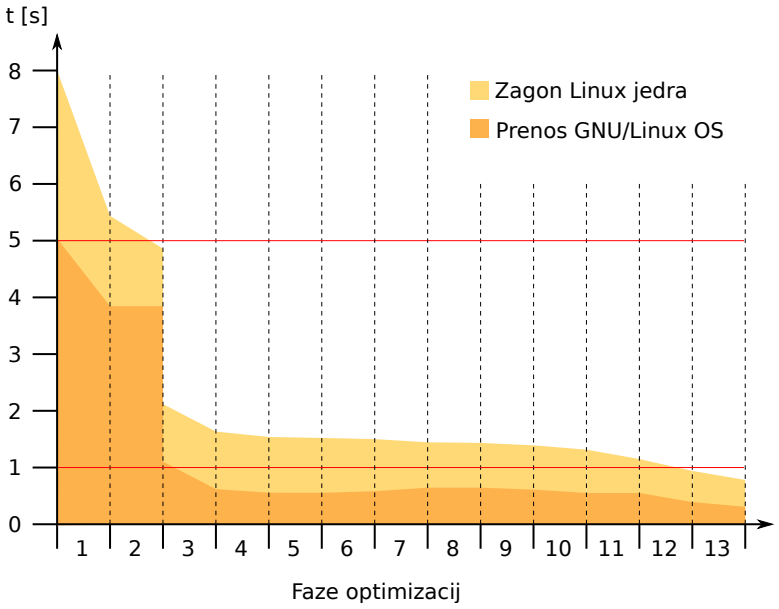
5.3.3 Optimizacije GNU/Linux OS

V spodnji tabeli vidimo faze pospešitev, njihov opis ter referenco na poglavje, kjer se nahaja več informacij glede faze optimizacije.

Faza	Opis	Poglavje
1	Čiščenje konfiguracije gradnje Linux jedra	5.2.1
2	Tihi zagon Linux jedra	5.2.2
3	Optimizacija prenosa iz QSPI flash pomnilnika	5.1.3
4	Odstranitev razhroščevalnih simbolov	5.2.3
5	Izbira alokacijskega algoritma Slab	5.2.5
6	Kompresija Linux jedra	5.2.6
7	Kompresija korenskega datotečnega sistema	5.2.7
8	Tip korenskega datotečnega sistema (initrd)	5.2.8
9	Odstranjevanje podpore kompresijskim algoritmom initrd	5.2.7
10	Standardna C knjižnica	5.2.9
11	Populacija /dev direktorija	5.2.10
12	Odstranitev odvečnih elementov Linux jedra v dat. sistemu	5.2.11
13	Končno čiščenje konfiguracije gradnje Linux jedra	5.2.14

Tabela 5.8: Faze optimizacije GNU/Linux OS.

Graf na sliki 5.10 predstavlja izboljšavo v smislu časa zagona na vsaki izmed faz optimizacije. Takoj lahko vidimo, katere faze optimizacije so se najbolj izplačale.

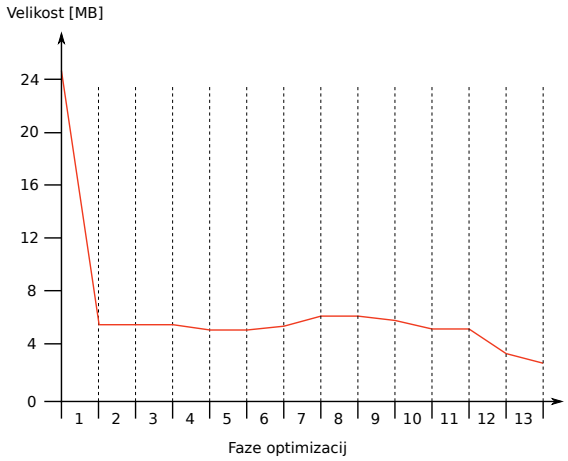


Slika 5.10: Čas zagona GNU/Linux OS.

Pomembno je poudariti, da smo v drugi fazi preklopili iz prenašanja GNU-/Linux OS elementov prek mreže (TFTP protokol) na prenos iz QSPI flash pomnilnika, zato v tej fazi pride do takšnega skoka.

Kot je razvidno iz grafa, se zelo izplača čiščenje konfiguracije gradnje Linux jedra (faza 1), kar je seveda mogoče samo, če Linux jedro sami gradimo. Tukaj vidimo, da se ta dodaten aspekt kontrole izplača, čeprav nam od začetka mogoče povzroča težave, saj gradnja Linux jedra ni trivialna. K hitrejšemu zagonu Linux jedra bistveno pripomoreta tudi tihi zagon Linux jedra (faza 2) ter pospešitev prenosa GNU/Linux OS iz nalagalnega medija (faza 3).

Zanimivi sta še fazi kompresije Linux jedra (faza 6) ter korenskega datotečnega sistema (faza 7), saj dokazujeta, da manjše ni vedno hitrejše. Pomembno je najti ravnovesje med dekompresijskim časom algoritma in časom, ki ga potrebujemo za prenos GNU/Linux OS v delovni pomnilnik, ta pa je direktno povezan z velikostjo sistema. Lepše se to vidi z grafa na sliki 5.11, ki predstavlja spreminjanje velikosti sistema pri fazah optimizacij.



Slika 5.11: Velikost GNU/Linux OS.

5.3.4 Končno stanje na ZedBoard vgrajenem sistemu

Pomemben pa ni le dosežen čas zagona sistema ampak tudi to, kaj sistem po tem času ponuja. Predvsem v smislu perifernih naprav. V našem primeru so bistvene štiri periferne naprave, ki jih želimo imeti na voljo v GNU/Linux sistemu.

- **Serijska komunikacija UART** – komunikacija z operacijskim sistemom prek terminala.
- **USB komunikacija** – komunikacija z napravami, povezanimi prek USB protokola s sistemom.
- **Mrežna komunikacija** – komunikacija z mrežo prek Ethernet protokola.
- **Dostop do persistenčnega pomn. medija** – dostop do nekega pomnilniškega medija, kjer lahko persistenčno shranimo podatke.

Delovanje teh štirih naprav je bilo tudi testirano. Kar se tiče konfiguracije mrežne komunikacije pa imamo dve možnosti:

1. **Dostop do terminala je mogoč, ko je mogoč dostop do mreže.**

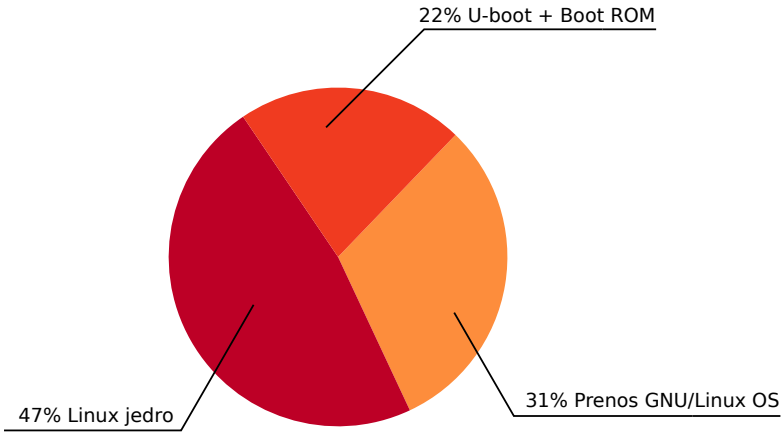
   Tukaj je konfiguracija mreže blokirajoča. Posledično je dostop do terminala mogoč po približno 4,1 sekundah od vklopa sistema.

2. **Dostop do terminala je mogoč takoj.**

   V tem primeru konfiguracijo mreže poženemo v ozadju kot še en proces. Posledica tega je, da je dostop do terminala mogoč po približno 1 sekundi od vklopa sistema, mreža pa postane dostopna po dodatnih 3,1 sekundah.

Katero od možnosti izberemo, je stvar zahteve končne aplikacije vgrajenega sistema.

Zdi se, da je večina pridobljenega časa povezana z velikostjo GNU/Linux OS. Manjši kot je sistem, hitreje ga prenesemo iz nalagalnega medija v delovni pomnilnik in hitreje ga razpakiramo. To je še vedno druga najbolj potratna faza zagona vgrajenega sistema z GNU/Linux OS, kot to vidimo s tortnega diagrama na sliki 5.12.



Slika 5.12: Procentualna poraba časa posameznih faz zagona.

Tudi nalagalnik pa, čeprav na prvi pogled najmanjši in najbolj trivialen del zagona, pri takšnih časih zagona ni več nedolžen. Prispeva slabo četrtino (22%) vsega časa k zagonu sistema.

Natančne meritve časa zagona na posameznih fazah so na voljo tudi v dodatku A.

Poglavje 6

Zaključek

Korektna obdelava problema diplomske naloge je zahtevala vzpostavitev okolja za gradnjo GNU/Linux OS ter implementacijo zanesljive metode merjenja časa zagona. S takšnim pristopom dodobra spoznamo GNU/Linux operacijski sistem, nalagalnike ter vgrajene sisteme.

Takšen projekt zahteva analizo in reševanje raznovrstnih problemov, od električnih, kot je integriteta signalov pri QSPI prenosu, do visokonivojskih, kot je izbor standardne C knjižnice na nivoju operacijskega sistema. S tem načinom dela dobimo občutek, kako deluje vgrajeni sistem, pa tudi splošen računalniški sistem kot združena celota strojne ter programske opreme. Tako se spoznamo tudi s tem, kar je dandanes potrebno za zagon in delovanje splošnega operacijskega sistema.

Delo podaja splošne smernice za optimizacijo zagona GNU/Linux OS za praktično katerikoli vgrajen sistem, kar je tudi eden od ciljev diplomske naloge. Nekaj optimizacij lahko tudi apliciramo na katerikoli računalniški sistem. Seveda se da še kaj postoriti na tem področju, saj zagon sistema še ni na minimalnem možnem času. Z optimizacijami, ki so predstavljene v tej diplomski nalogi, dosežemo padec s 35 sekund dolgega zagona sistema na 1 sekundo. Menim, da je tak čas že primeren za nekatere vgrajene sisteme.

Literatura

[1] ARM. *Cortex^{TM}-A9 MPCore® Technical Reference Manual, Revision: r3p0*, 2011.

[2] Avnet. *ZedBoard Hardware User's Guide, verzija 2.2*, Januar 2014.

[3] Building a root filesystem. Dosegljivo: `http://www.tldp.org/HOWTO/Bootdisk-HOWTO/buildroot.html`. [Dostopano: 9. 9. 2017].

[4] BusyBox. Dosegljivo: `https://busybox.net/about.html`. [Dostopano: 9. 9. 2017].

[5] Jonathan Corbet. *Debugfs*. Linux Kernel Organization, 2009. [Del dokumentacije Linux jedra].

[6] Wolfgang Denk. *README*. DENX Software Engineering, 2013.

[7] DENX Software Engineering. *README.arm-relocation*. [Del dokumentacije U-boota].

[8] DENX Software Engineering. *README.falcon*. [Del dokumentacije U-boota].

[9] DENX Software Engineering. *README.silent*. [Del dokumentacije U-boota].

[10] Device tree for Dummies. Dosegljivo: `http://free-electrons.com/pub/conferences/2013/elce/petazzoni-device-tree-`

dummies/petazzoni-device-tree-dummies.pdf. [Dostopano: 9. 9. 2017].

[11] driver-core: devtmpfs - driver core maintained /dev tmpfs [LWN.net]. Dosegljivo: https://lwn.net/Articles/330985/. [Dostopano: 10. 9. 2017].

[12] Everything is a File. Dosegljivo: http://ibgwww.colorado.edu/~lessem/psyc5112/usail/concepts/filesystems/everything-is-a-file.html. [Dostopano: 9. 9. 2017].

[13] Overview of the GNU System - GNU Project - Free Software Foundation. Dosegljivo: https://www.gnu.org/gnu/gnu-history.en.html. [Dostopano: 9. 9. 2017].

[14] Make - GNU Project - Free Software Foundation. Dosegljivo: https://www.gnu.org/software/make/. [Dostopano: 9. 9. 2017].

[15] iSYSTEM. *iC5000 On-Chip Analyzer Hardware reference, verzija 6.16*, Junij 2017.

[16] iSYSTEM - winIDEA. Dosegljivo: http://www.isystem.com/products/software/winidea. [Dostopano: 9. 9. 2017].

[17] The kernel's command-line parameters – The Linux Kernel documentation. Dosegljivo: https://www.kernel.org/doc/html/v4.10/admin-guide/kernel-parameters.html. [Dostopano: 10. 9. 2017].

[18] 4.1 Debugging Support in the Kernel. Dosegljivo: http://www.makelinux.net/ldd3/chp-4-sect-1. [Dostopano: 10. 9. 2017].

[19] FAQ/BUG - Linux Kernel Newbies. Dosegljivo: https://kernelnewbies.org/FAQ/BUG. [Dostopano: 10. 9. 2017].

[20] Kernel Configuration. Dosegljivo: http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html. [Dostopano: 9. 9. 2017].

[21] Kernel/Firmware - Ubuntu Wiki. Dosegljivo: https://wiki.ubuntu.com/Kernel/Firmware. [Dostopano: 10. 9. 2017].

[22] KernelHeaders - Linux Kernel Newbies. Dosegljivo: https://kernelnewbies.org/KernelHeaders. [Dostopano: 10. 9. 2017].

[23] Russel King. *Booting ARM Linux*. Linux Kernel Organization, Maj 2012.

[24] Inside the Linux boot process. Dosegljivo: https://www.ibm.com/developerworks/library/l-linuxboot/index.html. [Dostopano: 9. 9. 2017].

[25] Linux Kernel Organization. *ext2*. [Del dokumentacije Linux jedra].

[26] Linux Kernel Organization. *ext3*. [Del dokumentacije Linux jedra].

[27] Linux Kernel Organization. *ext4*. [Del dokumentacije Linux jedra].

[28] Linux and symmetric multiprocessing. Dosegljivo: https://www.ibm.com/developerworks/library/l-linux-smp/. [Dostopano: 10. 9. 2017].

[29] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual version 2.26*, 2017.

[30] Building BusyBox - musl libc wiki. Dosegljivo: http://wiki.musl-libc.org/wiki/Building_Busybox. [Dostopano: 10. 9. 2017].

[31] musl - Compiling latest busybox with latest musl. Dosegljivo: http://www.openwall.com/lists/musl/2014/08/08/13. [Dostopano: 10. 9. 2017].

[32] Speed Class - SD Association. Dosegljivo: https://www.sdcard.org/developers/overview/speed_class/. [Dostopano: 10. 9. 2017].

[33] Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB. Dosegljivo: `https://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf`. [Dostopano: 10. 9. 2017].

[34] Spansion. *S25FL128S and S25FL256S Data Sheet, Revision 09*, Maj 2015.

[35] strip(1): Discard symbols from object files - Linux man page. Dosegljivo: `https://linux.die.net/man/1/strip`. [Dostopano: 10. 9. 2017].

[36] TaskAddArmRelocation < U-boot < DENX. Dosegljivo: `http://www.denx.de/wiki/U-Boot/TaskAddArmRelocation`. [Dostopano: 10. 9. 2017].

[37] Denys Vlasenko. *MDEV Primer*. [Del dokumentacije BusyBoxa].

[38] Autonegotiation - Wikipedia. Dosegljivo: `https://en.wikipedia.org/wiki/Autonegotiation`. [Dostopano: 10. 9. 2017].

[39] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual, version 1.6.1*, September 2013.

[40] Xilinx. *Zynq-7000 All Programmable SoC Data Sheet: Overview, version 1.11*, Junij 2017.

[41] Xilinx Wiki - Mainline Linux on Zynq. Dosegljivo: `http://www.wiki.xilinx.com/Mainline+Linux+on+Zynq`. [Dostopano: 10. 9. 2017].

Dodatek A

Meritve časa zagona v posameznih fazah

Faza	Opis
1	Premestitev nalagalnika v pomnilniku
2	Tihi zagon nalagalnika
3	Čiščenje konfiguracije gradnje Linux jedra
4	Tihi zagon Linux jedra
5	Optimizacija prenosa iz QSPI flash pomnilnika
6	Odstranitev razhroščevalnih simbolov
7	Izbira alokacijskega algoritma Slab
8	Kompresija Linux jedra
9	Kompresija korenskega datotečnega sistema
10	Tip korenskega datotečnega sistema (initrd)
11	Odstranjevanje podpore kompresijskim algoritmom initrd
12	Standardna C knjižnica
13	Populacija /dev direktorija
14	Odstranitev odvečnih elementov Linux jedra v dat. sistemu
15	Končno čiščenje konfiguracije gradnje Linux jedra

Tabela A.1: Opis vseh faz optimizacije.

Faza	Boot ROM + SPL	U-boot	Prenos GNU/Linux OS	Zagon OS	Vsota
0[1]	177	151	5060	2999	8387
1	177	98	5060	2999	8334
2	165	51	5060	2999	8275
3	165	51	3844	1593	5653
4	165	51	3844	1015	5075
4[2]	165	51	1103	1015	2334
5	165	51	619	1015	1850
6	165	51	556	983	1755
7	165	51	556	965	1737
8	165	51	584	917	1717
9	165	51	645	800	1661
10	165	51	645	789	1650
11	165	51	610	781	1607
12	165	51	551	764	1531
13	165	51	551	488	1255
14	165	51	391	549[3]	1155
15	165	51	308	473	997

Tabela A.2: Meritve časa zagona v milisekundah.

[1]Začetna faza.

[2]Preskok iz TFTP nalagalnega medija na QSPI flash pomnilnik.

[3]Dodajanje gonilnikov za USB ter SD kartico.