SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 3054

Izvršavanje firmwarea PLC-a korištenjem alata QEMU

Petar Kovač

Zagreb, lipanj 2022.

Zahvaljujem prijateljima na sreći koju su godinama pružali. Obitelji na podršci i razumijevanju. Mentoru doc. dr. sc. Stjepanu Grošu na savjetima i suradnji tijekom preddiplomskog i diplomskog studija.

SADRŽAJ

1. Uvod

Računala koja se koriste u industriji nisu slična korisničkim računalima. Ona upravljaju industrijskim procesom te moraju biti robusna i za razliku od korisničkih računala nikada ne smiju raditi greške. Takva industrijska računala se zovu programirljivi logički kontroleri (engl. *programmable logic controller*, PLC).

Cijena netočnog rada PLC-a može biti mala, primjerice gubitak kvalitete proizvoda. Ako se radi o netočnom radu više PLC-ova u isto vrijeme, cijena može biti veća, na razini proizvodne sposobnosti jedne tvornice ili čak u ekstremnim slučajevima dijela industrijskog sektora države. Posebno je zabrinjavajuće kada je netočan rad PLC-a uzrokovan zloupotrebljivanjem ranjivosti u njegovoj programskoj podršci. Zbog toga je važno testirati programsku podršku PLC-a i osigurati da nema ranjivosti. To je otežano činjenicom da za većinu PLC-ova nije javno dostupna niti programska podrška ni sklopovlje na kojoj se izvodi. Bez platforme pomoću koje istraživači mogu lagano testirati programsku potporu, PLC-ovi će ostati ranjivi.

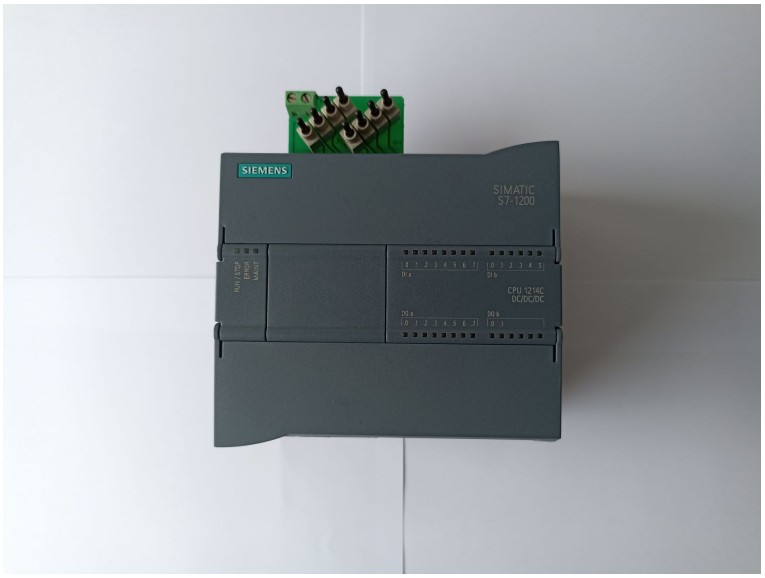
Cilj ovog rada je otkrivanje informacija o sklopovlju i programskoj podršci Siemensovog S7-1200 PLC-a tako da se omogući emuliranje uređaja korištenjem alata QEMU. Emulacija bi se u konačnici koristila kao platforma pomoću koje bi se moglo raznim alatima i metodama, primjerice neizrazitim testiranjem, otkriti ranjivosti u programskoj podršci PLC-a. Rad je strukturiran na sljedeći način. U drugom poglavlju se razmatraju mogućnosti nabave *firmwarea* PLC-a te analizira sklopovlje PLC-a. U trećem poglavlju se pokazuju alati koji se koriste, njihove mogućnosti te specifičnosti procesora koji su bitni za emulaciju i analizu *firmwarea*. U četvrtom poglavlju se otkriva općenita funkcionalnost cjelokupne programske potpore te se informacije dobivene tijekom analize koriste za pospješivanje emulacije. Dodatno se pokazuju rezultati rada. U zaključku će sažeto biti opisani cilj i doprinos rada te će biti komentirani rezultati.

2. Analiza *firmware* datoteke

Firmware S7-1200 PLC-a sadrži operacijski sustav koji omogućuje korisniku da pomoću PLC-a upravlja industrijskim procesom. Pošto je cilj emulacije što vjernije oponašanje rada *firmwarea* na stvarnom PLC-u, potrebno je što bolje se upoznati sa strukturom te datoteke i komponentama PLC-a.

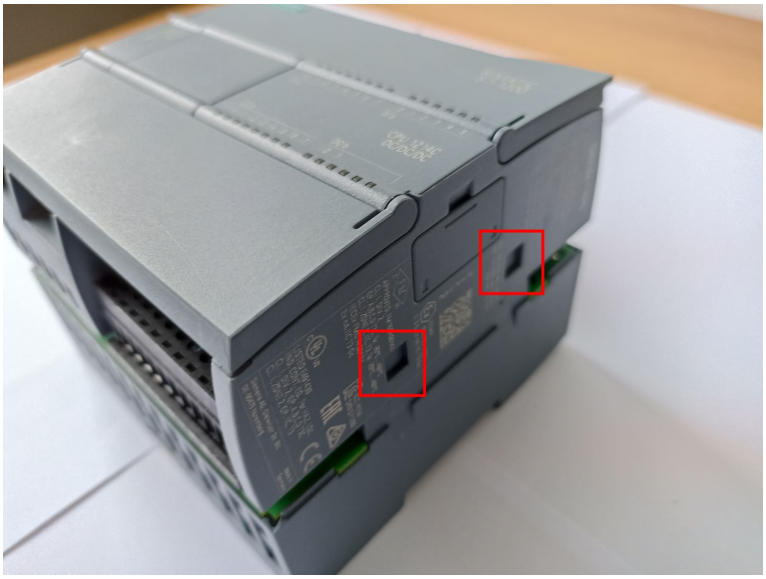
2.1. Sklopovlje PLC-a

Sklopovlje Siemensovog S7-1200 PLC-a verzije 4 se nalazi u plastičnom kućištu koje štiti PLC i pruža sučelja za spajanje na ulaze i izlaze PLC te za modularnu nadogradnju PLC-a. Na slikama 2.1, 2.2 se može vidjeti izgled PLC-a izvana. Verzija PLC-a kojeg se analizira u sklopu ovog rada je 6ES7214-1AG40-0XB0 [1]. Gornji dio kućišta se može lagano odvojiti od ostatka PLC-a pritiskom na utore označene na slici 2.3 te povlačenjem gornjeg dijela kućišta.



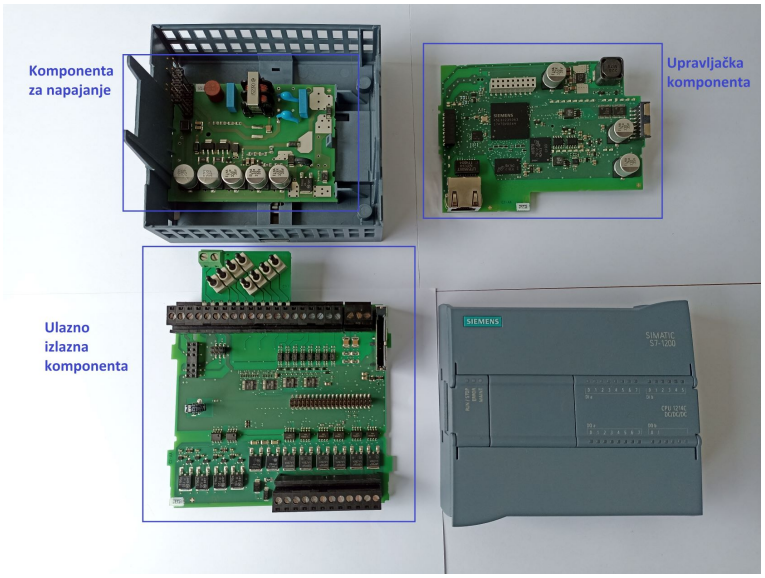
Slika 2.1: Gornja strana PLC-a

Slika 2.2: Prednja strana PLC-a



Slika 2.3: Utori na strani PLC-a

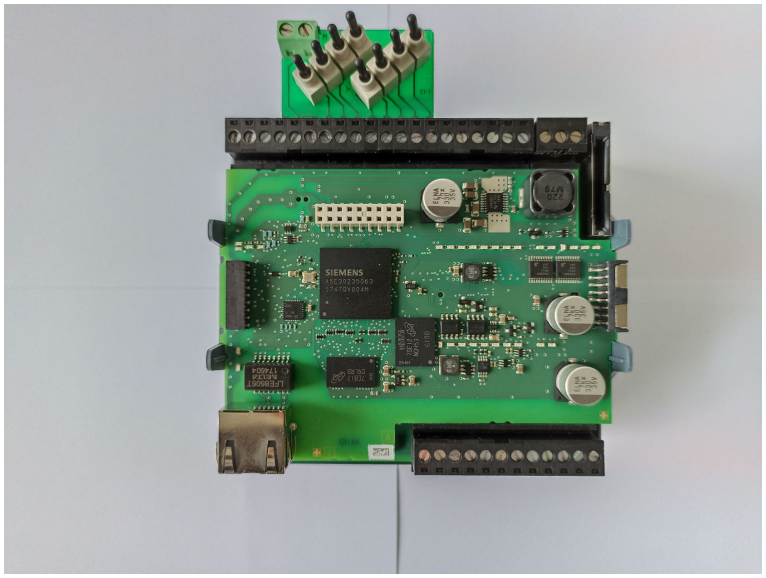
Kada se otvori kućište PLC-a, može se vidjeti da se PLC sastoji od tri sloja. Prvi je sloj s većinom upravljačkih elemenata. Na njemu se nalazi procesor, radna memorija, NAND čip, NOR čip te ostale komponente za komunikaciju. Sljedeći je sloj koji sadrži komponente za ulazno-izlaznu komunikaciju. Preko tih komponenti PLC dobiva informacije iz okoline i šalje podatke prema okolini. Na zadnjem sloju se nalaze komponente za napajanje. U radu se analizira upravljačka komponenta na vrhu zato što sadrži *firmware*. Na slici 2.4 su prikazane sve tri komponente s kućištem.



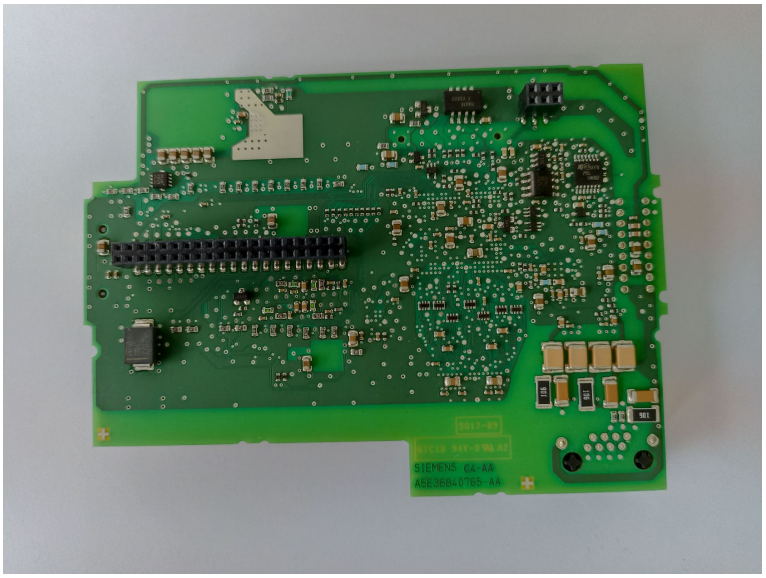
Slika 2.4: Sve komponente PLC-a

Slike 2.5, 2.6 prikazuju prvi *printed circuit board* (PCB) s gornje i donje strane. Informacije za većinu komponenti na PCB-u se mogu dobiti internetskim pretraživanjem broja na čipu. Jedina komponenta koja se ne može naći je *application-specific integrated circuit* (ASIC), pošto je napravljen isključivo za Siemens te ne postoji javno dostupna dokumentacija. Za nalaženje procesora koji se nalazi u ASIC-u, potrebno je naći *joint test action group* (JTAG) sučelje za testiranje te se preko njega spojiti na CoreSight sučelje za nalaženje grešaka koje ARM procesori nude. Tada se može poslati naredba za identifikaciju procesora i pronaći model koji se koristi. U slučaju S7-1200 v4 PLC-a, procesor je Cortex-R4 r1p3 [2] koji koristi ARMv7-R arhitekturu. Nalaženje JTAG sučelja nije lagano zato što je potrebna vrlo detaljna analiza izvoda na pločici da bi se mogli naći mogući kandidati. Za pločicu sa stotinama izvoda kao što je PCB na S7-1200 PLC-u, bez naprednih alata to je izrazito teško.

Od ostalih važnijih komponenti može se vidjeti radna memorija od 1 Gb [3]. Jedan od načina kako se može zaključiti da se radi o radnoj memoriji je taj da je povezana s ASIC-om putem krivuljastih linija, tzv. *serpentine routing* [4]. To je zato što je poželjno da sve linije budu jednake duljine tako da podatci dolaze istovremeno. Drugi razlog je taj što se time smanjuje interferencija između signala. Sljedeći je NAND čip MT29F2G16ABBEAHC-AIT:E na kojem se nalazi *firmware* koji sadrži 256 Mb memorije [5] te podržava *Open NAND Flash Interface* standard 1.0 za komunikaciju. S druge strane pločice se može vidjeti NOR čip IS25LQ040B s 4 Mb memorije [6] koji podržava *Serial Peripheral Interface* (SPI) standard za komunikaciju.



Slika 2.5: Upravljački PCB - gornja strana



Slika 2.6: Upravljački PCB - donja strana

2.2. Nabava *firmwarea*

Postoji više načina kako se može nabaviti *firmware* za neki PLC. Najjednostavniji način je, ako je to omogućeno, s proizvođačeve stranice [7] preuzeti datoteku za nadogradnju *firmwarea*. To je datoteka koja, između ostalog, sadrži izvršni kôd *firmwarea*. Ovaj pristup je jednostavan zato što nije potrebno imati primjerak PLC-a te zato što

je datoteka najčešće besplatno dostupna. Postoje poteškoće koje dolaze s ovim pristupom. Proizvođači programske potpore za PLC žele zaštititi razvijenu programsku potporu od analize - komprimiranjem ili kriptiranjem datoteka za nadogradnju ili obfuskacijom izvršnih datoteka koje se nalaze u njima. Obfuskacija je postupak kojim se otežava razumijevanje kôda tako da se kôd umjetno učini kompliciranijim. U slučaju Siemensovih datoteka za nadogradnju *firmwarea* verzije 4, algoritam koji se koristi za kompresiju je LZP3. To je relativno nepoznat algoritam. U vrijeme pisanja ovog rada pretraživanjem interneta se može naći tek nekoliko implementacija koje su isprobane i koje neuspješno dekomprimiraju Siemensovu datoteku za nadogradnju *firmwarea*. U [8] je uspješno dekomprimirana *firmware* datoteka te je objavljen i izvorni kôd [9]. U sklopu ovog istraživanja je isto tako napravljena implementacija LZP3 algoritma [10].

Drugi način je da se nabavi PLC te se identificira čip na PCB-u koji sadrži *firmware*. Potom se na čip spaja pomoću programatora, uređaja za čitanje i pisanje na razne programabilne integrirane sklopove. Sadržaj memorije nekog čipa se može preuzeti koristeći dostupnu programsku podršku za programator, primjerice *flashrom* [11]. Problemi kod ovog pristupa su brojni. Moguće je da je čip sklopovski zaštićen od čitanja te da je, pošto se radi o vrlo malim čipovima, teško spojiti se na čip bez da se ošteti PLC. Isto tako, zato što dokumentacija za sklopove na PLC-ovima često nije javno dostupna, potrebno je naći *datasheet* tog čipa tako da se čip točno spoji s programatorom. Programska podrška programatora mora podržavati programiranje čipa. Alternativno se može napraviti vlastita programska podrška za programiranje čipa koristeći *datasheet* za taj sklop. Konačno, ponekad je nemoguće spojiti se na čip bez oštećenja PLC-a. Tako se čip s *firmwareom* na Siemensovim S7-1200 v4 PLC-ovima ne može čitati bez da se odvoji od PCB-a te se posljedično PLC više ne može koristiti.

U [12] se ovaj problem riješio tako da se nije spajalo na čip s *firmwareom*, već na čip s *bootloaderom*. Njega nije potrebno odvojiti od PCB-a da bi se spojilo s programatorom te se može i mijenjati sadržaj samog čipa. Potom je u *bootloader* stavljen *gdb stub* koji je omogućio da se ispiše sadržaj radne memorije nakon prebacivanja *firmwarea* s čipa u radnu memoriju. U ovom radu se predlaže sličan postupak, ali u kojem se koristi *universal asynchronous receiver/transmitter* (UART) komunikacijski protokol i čitanje memorije s čipa umjesto iz radne memorije. Razlika između ova dva postupka je da se tijekom premještanja *firmwarea* u radnu memoriju mijenjaju dijelovi memorije te se čitanjem iz radne memorije ne dobiva isti sadržaj kao što je u čipu.

2.3. Nabava *firmwarea* preko UART-a

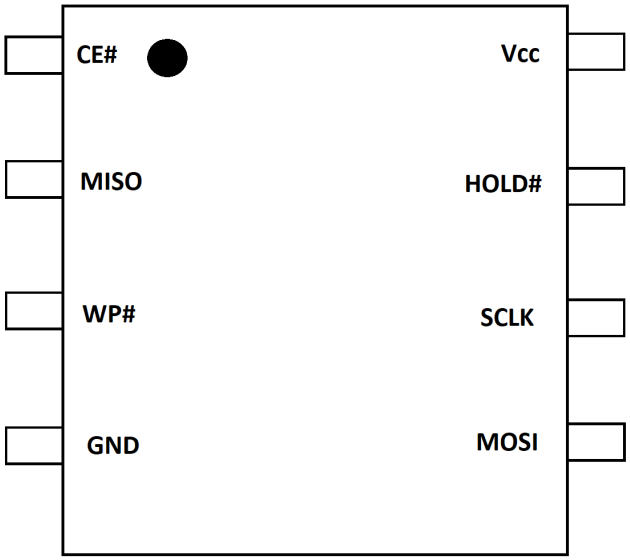
U ovom potpoglavlju se opisuje cjelokupni postupak nabave *firmwarea* s NAND čipa Siemensovog S7-1200 PLC-a. Na PLC je stavljena najnovija verzija *firmwarea* u vrijeme pisanja rada, verzija $4.5.2$.

Čip u kojem se nalazi *firmware* ima 256 Mb NAND memorije, a *bootloader* se nalazi na čipu s 4 Mb NOR memorije. Zbog ograničenja s resursima, odvajanje čipa od PCB-a nije bila opcija. Zato je odlučeno da se *firmware* memorija dobavi pomoću *bootloadera*. Za to je potrebno dobaviti *bootloader* s NOR čipa, izmijeniti njegov kôd tako da u njega stavimo funkciju u kojoj će se preko UART-a slati sadržaj *firmwarea*, zaobići mehanizme koji osiguravaju integritet izvršnog kôda te konačno staviti izmijenjeni kôd natrag na čip *bootloadera*.

Sučelje za komunikaciju koje NOR čip koristi je SPI. U njemu su definirana 4 logička signala:

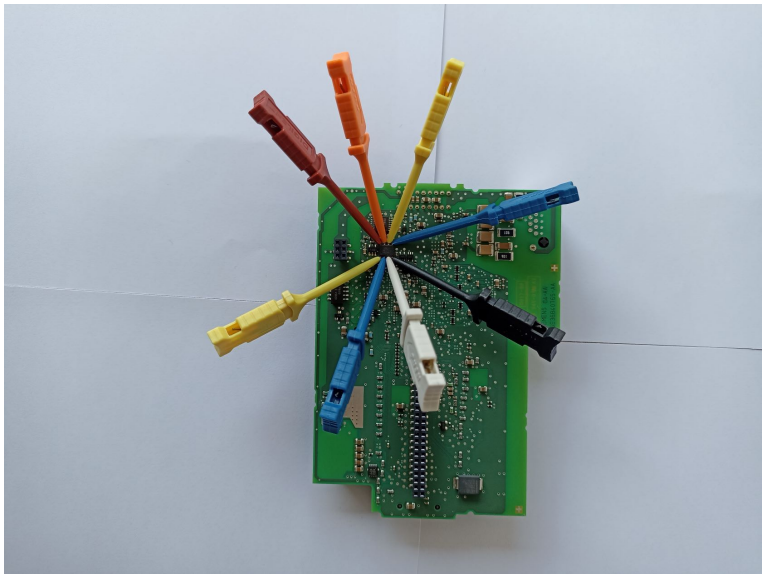
1. *Serial Clock* (SCLK),
2. *Master Out Slave In* (MOSI),
3. *Master In Slave Out* (MISO),
4. *Chip Select* (CS).

SCLK je signal takta, MOSI se koristi za slanje podataka prema čipu, MISO za dobivanje podataka od čipa te CS za odabir čipa. NOR čip na S7-1200 PLC-u ima sve ove signale te neke dodatne - signal Vcc za napajanje, GND za uzemljenje, HOLD za pauziranje komunikacije te konačno WP za zaštitu od pisanja. Na slici 2.7 je prikazan razmještaj ovih signala na čipu. Crna točka je prisutna na NOR čipu i služi identificiranju prvog signala CE#. Signali koji imaju ljestvice pored sebe su negirani signali, tj. signali koji su aktivni na logičkoj nuli.



Slika 2.7: Razmještaj signala na čipu IS25LQ040B

Kada znamo položaj signala na čipu, može ga se spojiti na programator. U sklopu ovog rada je korišten *bus pirate* v3.6 [13], koji podržava SPI komunikacijsko sučelje. NOR čip ima Small Outline Integrated Circuit (SOIC) paket, specifično SOIC-8. Broj 8 označava da čip ima 8 nožica. Za spajanje na takav čip nije preporučljivo koristiti žice zato što je razmak između kontakata na čipu iznimno malen te je jako velika mogućnost kratkog spoja. Bolja mogućnost je koristiti SOIC-8 hvataljke ili kukice za hvatanje na nožice. Potrebno je biti pažljiv kod odabira SOIC-8 hvataljki pošto se hvataljke lošije kvalitete neće dobro uhvatiti za čip te mogu i izgrebati čip do te mjere da se više ne hvataju ili čak da oštete sklop. Hvataljke koje su korištene u sklopu ovog rada i koje se preporučuju su Pomona 5250 hvataljke [14]. U slučaju korištenja kukica za hvatanje na nožice, potrebno je paziti da su kukice dovoljno malih dimenzija. Za razliku od SOIC-8 hvataljki, kukice neće lagano otpasti s čipa te neće oštetiti sklop. Na slici 2.8 su prikazane kukice koje su spojene na čip.



Slika 2.8: Spajanje na NOR čip s kukicama

Nakon spajanja na čip s programatorom, potrebno je pomoću programske potpore na programatoru preuzeti sadržaj memorije čipa. U slučaju programatora *bus pirate* je najpopularnija programska potpora alat *flashrom* koji podržava komunikaciju s velikim brojem čipova, ali ne i s čipom IS25LQ040B, tj. s NOR čipom. Pošto je *flashrom* alat otvorenog kôda, moguće je napisati novi modul i implementirati funkcionalnost za upravljanje čipom. Ako se nađe dovoljno sličan čip, moguće je i kopirati njegovu konfiguraciju u izvornom kodu alata *flashrom* te samo dodati identifikacijski broj čipa IS25LQ040B. Nakon tog koraka je moguće dobaviti memoriju na čipu. Jedina programska zaštita *bootloadera* je ciklički zaštitni kod (engl. *cyclic redundancy check*, CRC). To znači da je moguće izmijeniti kôd tako da sadrži funkciju koja sadržaj *firmwarea* šalje preko UART sučelja. Jednom kada je kôd izmijenjen, potrebno je izračunati novi CRC te se program konačno pomoću programatora može ponovo staviti na NOR čip. UART sučelje na S7-1200 PLC-u je dokumentirano [15] te se spajanjem na sučelje sada može pristupiti cijelom sadržaju NAND čipa bez da ga se ošteti. Zbog nedostatka vremena, ovaj postupak nije napravljen. Bez obzira na to, ovaj rad se može koristiti kao temelj za budući rad na ovom postupku.

2.4. Struktura *firmware* datoteke

Firmware datoteka se u obliku kakvom je spremljena na NAND čipu sastoji od dva glavna dijela, zaglavlja i izvršnog kôda. Izvršni kôd je dalje dodatno podijeljen na

segmente koje *bootloader* tijekom inicijalizacije sustava kopira u radnu memoriju. Da bi *firmware* bio funkcionalan, on se ne može jednostavno kopirati u radnu memoriju. Razmještaj *firmwarea* na NAND čipu i u radnoj memoriji nije isti i svaki segment ima svoju očekivanu početnu adresu. Osim toga, prvih 64 okteta, tj. zaglavlje *firmwarea* se ne kopira u radnu memoriju. Tablica 2.1 prikazuje ime svakog od njegovih polja.

Adresa	Ime polja	Kratak opis
`0x00`	Magični broj	`0x5D1B4153`, Označava da je ova datoteka *firmware* za S7-1200 PLC v4.
`0x04`	Ulazna točka	Adresa od koje će se *firmware* početi izvršavati nakon što mu *bootloader* preda kontrolu.
`0x08`	Početak *firmware* sadržaja	Početna adresa u NAND čipu od koje počinje dio *firmwarea* koji će biti u radnoj memoriji.
`0x0C`	Nepoznato	Nepoznato.
`0x10`	Virtualna adresa *firmware* sadržaja	Početna adresa u radnoj memoriji na koju će se *firmware* kopirati.
`0x14`	Veličina zaglavlja	Veličina ovog zaglavlja, uvijek je 64 okteta.
`0x18`	Kraj ITCM sadržaja	Krajnja adresa dijela *firmwarea* u NAND čipu koji ide u ITCM.
`0x1C`	IRAM početna adresa	Adresa na koju se mapira unutarnja radna memorija procesora.
`0x20`	Verzija *firmwarea*	Verzija *firmwarea*.
`0x24`	Nepoznato	Nepoznato, postavljeno na 0.
`0x28`	Nepoznato	Nepoznato, postavljeno na 0.
`0x2C`	Početak `.bss` odjeljka	Početna adresa neinicijalizirane memorije.
`0x30`	Adresa funkcije prve dretve	Adresa funkcije koju izvršava prva dretva nakon inicijalizacije operacijskog sustava.
`0x34`	Kraj *firmware* sadržaja	Konačna adresa memorije u NAND čipu koja se kopira u radnu memoriju.
`0x38`	Nepoznato	Nepoznato, postavljeno na 0.
`0x3C`	CRC zaglavlja	CRC koji se računa i provjerava za zaglavlje.

Tablica 2.1: Zaglavlje izvršne datoteke *firmwarea*

Neka od ovih polja je potrebno detaljnije objasniti. *Početak firmware sadržaja* i *virtualna adresa firmware sadržaja* su povezani na način da kada se *firmware* kopira s čipa u radnu memoriju, prvi oktet *firmwarea* od početka sadržaja će biti zapisan na

prvi oktet virtualne adrese, drugi na drugi te tako slijedno dok se ne dođe do adrese zapisane u polju *kraj firmware sadržaja* na čipu. Sadržaj *firmwarea* koji se stavlja u *instruction tightly coupled memory* (ITCM) je cijeli blok izvršnog kôda od kraja zaglavlja do adrese zapisane u polju *kraj ITCM sadržaja*.

Dio *firmwarea* koji sadrži izvršni kôd je na NAND čipu zapisan slijedno nakon zaglavlja. Izvršni kôd je podijeljen na memorijske segmente te je za svaki segment zapisana informacija o početnoj virtualnoj adresi, veličini segmenta u oktetima, ime segmenta i zastavica koja može označavati ako je segment inicijaliziran na početku rada PLC-a te je li dozvoljeno čitanje, pisanje i izvršavanje. Zastavica može označavati još jednu dodatnu dozvolu, ali njezino je značenje trenutno nepoznato. Na tablici 2.2 su prikazani svi segmenti s ovim informacijama. Oznakom / su označeni memorijski segmenti koji ne postoje na samom čipu, već se inicijaliziraju tijekom rada *firmwarea*. Slovima `R`, `W` i `X` su označene dozvole za čitanje, pisanje i izvršavanje, slovom `U` zastavica za neinicijaliziranost segmenta te konačno slovom ? nepoznata zastavica.

Ime segmenta	Adresa na čipu	Virtualna adresa	Veličina	Zastavice
.exec_in_lomem	0x00000040	0x00000000	0x00007624	- - - - X
.bitable	0x00008000	0x00040000	0x00000040	- - - - X
.sdramexec	0x00008040	0x00040040	0x000004F4	- - - - X
.syscall	0x00008540	0x00040540	0x00000008	- - - - X
.th_initial	0x00009040	0x00041040	0x00002310	? - - - X
.secinfo	0x0000B380	0x00043380	0x000003B8	? - R - -
.fixaddr	0x0000B740	0x00043740	0x00000000	- U - - -
.fixtype	0x0000B740	0x00043740	0x00000000	- U - - -
.text	0x0000B740	0x00043740	0x00F6C5C8	? - - - X
.rodata	0x00F77D40	0x00FAFD40	0x0042E40C	? - R - -
.data	0x013A6180	0x013DE180	0x00073FE4	? - R W -
.tls.cond.data	/	0x01452164	0x00000000	- - - - -
.bss	/	0x01E01040	0x00B3058C	- U - W -
.tls.cond.bss	/	0x029315CC	0x00000000	- U - - -
.uninitialized	/	0x03641040	0x03F8B904	- U - W -
CLSI_CACHED_MEM_POOL	/	0x075CC960	0x00000000	- U - - -
.cc_memory	/	0x075F0000	0x00600000	- U - W -
OPEN_BSD_MEM_POOL	/	0x07BF0000	0x00400000	- U - W X
.dram_uncache	/	0x07FF0000	0x00000000	- U - - -
MAP_MAC_MEM	/	0x07FF0000	0x00000494	- U - W -
.iram0	/	0x10030000	0x00007AA0	- U - W -
.iram1	/	0x10040000	0x0000C6A4	- U - W -
.crctable	/	0x1004F400	0x00000400	- U - W -
.softboot	/	0x1004F800	0x00000700	- U - W -
.bootinfo	/	0x1004FF00	0x0000001C	- U - W -
.dtcm	/	0x10010000	0x00002E00	- U - W -

Tablica 2.2: Memorijski segmenti S7-1200 *firmwarea* verzije 4.5.2

U tablici se mogu prepoznati uobičajeni memorijski segmenti kao `.text` za kôd, `.data` za inicijalizirane varijable, `.rodata` za konstante te `.bss` za neinicijalizirane varijable. Od drugih zanimljivijih memorijskih segmenata se mogu izdvojiti `.sdramexec`, u kojem se događa inicijalizacija operacijskog sustava, `.th_initial` u kojem je izvršni kôd kojeg izvršava prva dretva i `.uninitialized` u kojem se alocira dinamički alocirana memorija. Svaki od memorijskih segmenata se nalazi u radnoj memoriji,

osim `.exec_in_lomem` segmenta koji se nalazi u ITCM-u te `.dtcm` segmenta koji se nalazi u *data tightly coupled memoryju* (DTCM). ITCM i DTCM su memorije od nekoliko kB implementirane na Cortex-R4 r1p3 procesoru koje rade približno blizu brzini priručnog spremnika. Radi ovoga se uglavnom koriste kada je potrebna brza reakcija na događaje kao što su primjerice prekidi. Sa svim ovim informacijama bi se moglo pokušati izgraditi izvršnu ELF datoteku, ali zbog razloga koji će se kasnije spomenuti, to nije dovoljno za emulaciju.

Konačno, u *firmwareu* je definirano i memorijsko mapiranje svih perifernih uređaja koje je prikazano na tablici 2.3. S tim mapiranjem je moguće u svakom dijelu *firmwarea* odrediti s kojim uređajem se komunicira. Bez ove tablice bi bilo značajno teže zaključiti što se događa u *firmwareu* zato što bez nje ne bi postojao kontekst iz kojeg bi se moglo zaključiti što se događa u kôdu. Tako se primjerice u nedostatku tablice ne može bez dugotrajne analize znati što se događa kada se u *firmwareu* pristupa nekoj lokaciji, dok bi se s tablicom moglo zaključiti da se postavlja sat za nekakav prekid. U tablici su memorijska područja kojoj pripadaju istoj skupini uređaja pobojani drukčijom bojom od ostatka.

Ime segmenta	Početna adresa	Veličina
itcm	0x00000000	0x00008000
ddram	0x00008000	0x03FF8000
configured_dtcm	0x10010000	0x00004000
internal_ram0	0x10030000	0x00010000
internal_ram1	0x10040000	0x00010000
MAP3_PWRSTK	0xFFFB0000	0x0000003C
MAP3_SPI0	0xFFFB1000	0x00000018
MAP3_SPI1	0xFFFB2000	0x00000018
MAP3_I2C0	0xFFFB3000	0x0000006C
MAP3_I2C1	0xFFFB4000	0x0000006C
MAP3_I2C2	0xFFFB5000	0x0000006C
MAP3_ADC	0xFFFB6000	0x00000024
MAP3_UART0	0xFFFB7000	0x0000009C
MAP3_UART1	0xFFFB8000	0x0000009C
MAP3_HSC0	0xFFFB9100	0x00000080
MAP3_HSC1	0xFFFB9180	0x00000080
MAP3_HSC2	0xFFFB9200	0x00000080

Ime segmenta	Početna adresa	Veličina
MAP3_HSC3	0xFFFB9280	0x00000080
MAP3_HSC4	0xFFFB9300	0x00000080
MAP3_HSC5	0xFFFB9380	0x00000080
MAP3_INPUTS	0xFFFB9000	0x00000400
MAP3_PLS0	0xFFFBA080	0x00000080
MAP3_PLS1	0xFFFBA100	0x00000080
MAP3_PLS2	0xFFFBA180	0x00000080
MAP3_PLS3	0xFFFBA200	0x00000080
MAP3_OUTPUTS	0xFFFBA000	0x00000218
MAP3_ITIMER0	0xFFFBB010	0x00000010
MAP3_ITIMER1	0xFFFBB020	0x00000010
MAP3_ITIMER2	0xFFFBB030	0x00000010
MAP3_ITIMER3	0xFFFBB040	0x00000010
MAP3_ITIMER4	0xFFFBB050	0x00000010
MAP3_ITIMER5	0xFFFBB060	0x00000010
MAP3_ITIMER6	0xFFFBB070	0x00000010
MAP3_ITIMER7	0xFFFBB080	0x00000010
MAP3_ITIMER8	0xFFFBB090	0x00000010
MAP3_ITIMER9	0xFFFBB0A0	0x00000010
MAP3_ITIMER10	0xFFFBB0B0	0x00000010
MAP3_ITIMER11	0xFFFBB0C0	0x00000010
MAP3_ITIMER12	0xFFFBB0D0	0x00000010
MAP3_ITIMER13	0xFFFBB0E0	0x00000010
MAP3_TIMERS	0xFFFBB000	0x0000015C
MAP3_IOC	0xFFFBC000	0x0000002C
MAP3_FL_MEMCTL	0xFFFBD000	0x00010000
MAP3_VIC	0xFFFFFC00	0x00000200
MAP3_EMB0	0xFFF50000	0x00000048
MAP3_EMB1	0xFFF51000	0x0000008C
MAP3_DDR_MEMCTL	0xFFF52000	0x0000008C
MAP3_MMC	0xFFF60000	0x00000104
MAP3_LCD	0xFFF70000	0x00000FF8
MAP3_MAC	0xFFF90000	0x000000A4
MAP3_BOOL_HELPER	0xFFFA0000	0x00004000

Tablica 2.3: Memorijski segmenti za periferne uređaje

3. Emulacija

3.1. QEMU načini rada

Alat QEMU [16] nudi dva načina rada - *user mode* i *system* emulaciju. *User mode* emulacija omogućava emulaciju rada nekog procesora na računalu koje nema taj procesor. Ako je neki program preveden za jednu arhitekturu, pomoću *user mode* emulacije se taj program može pokrenuti na računalu koje ne mora imati istu arhitekturu. Alat QEMU prevodi naredbe programa u međukôd koji se zove *tiny code*. Potom se *tiny code* prevodi u instrukcije za računalo na kojem se program izvodi. QEMU podržava mnogo arhitektura: ARM, x86, SPARC, MIPS, MicroBlaze, PowerPC itd. Zbog ovoga se *user mode* emulacija u većini slučajeva može koristiti za emulaciju izvođenja programa. Jedini preduvjet je taj da program mora biti preveden u nekom od poznatih izvršnih formata kao što su *executable linkable format* (ELF) ili *portable executable* (PE). Zbog ovoga se *user mode* emulacija uglavnom ne može koristiti za emulaciju izvođenja *firmwarea* PLC-a.

Firmware PLC-a često nije preveden u neki od poznatih izvršnih formata, već je spremljen kao binarna datoteka koju procesor PLC-a može izvršavati u trenutku kada se upali, tj. dovoljna je inicijalizacija koja se događa na sklopovskoj razini pri paljenju PLC-a da se postigne stabilno stanje. Alternativno, *firmware* PLC-a može biti spremljen u izvršnom formatu kojeg je napravio proizvođač PLC-a, ali nije javno dostupan. Ovo je još jedan od načina na koji proizvođači PLC-a štite svoj kôd. Za takve izvršne programe je prigodna *system* emulacija koja ne zahtijeva da izvršna datoteka ima neki poznati format. Putem *system* emulacije se emulira ne samo izvođenje programa na nekom drugom procesoru, već cijeli sustav na kojem se program izvodi. Ovo znači da ako već u QEMU-u ne postoji implementacija pločice koju je potrebno emulirati, potrebno ju je samostalno implementirati što zahtijeva jako veliku količinu truda. Čak i da postoji implementacija neke pločice koja ima prikladan procesor, vjerojatnost da će ostatak sustava biti dovoljno sličan da omogući točno izvršavanje ciljnog programa je vrlo malena.

Sljedeći razlog zašto je potrebna *system* emulacija je taj što *firmware* memorijski mapira periferne uređaje. To znači da za svaki periferni uređaj postoji memorijska regija te se komunikacija s tim uređajem svodi na pisanje i čitanje memorijskih lokacija koje su u memorijskoj regiji tog uređaja. Tako se za neki uređaj može provjeriti ako je spreman za pisanje podataka čitanjem prikladne memorijske lokacije u njegovoj memorijskoj regiji te se nakon toga pisanjem na drugu memorijsku lokaciju mogu i poslati podatci. Kada bi *firmware* nekako i bio uspješno pokrenut putem *user mode* emulacije, to ne bi bilo dovoljno da se emulira ponašanje PLC-a zato što *user mode* emulacija ne dozvoljava emulaciju perifernih uređaja. Pomoću *system* emulacije može se odrediti što se točno treba dogoditi kada se pristupa određenoj memorijskoj lokaciji.

Cilj emulacije je da se ostvari ponašanje istovjetno ponašanju kada bi se program izvršavao na samom PLC-u. Prvo je potrebno implementirati ponašanje procesora u QEMU-ovom kôdu. Moguće je da je takav ili sličan procesor već implementiran pa nije potrebno to napraviti. Ako se radi o *system* emulaciji, procesor će biti implementiran u sklopu neke pločice koja ima svoje periferne uređaje. Tada je potrebno prilagoditi kôd te pločice za potrebe emulacije PLC-a. U sklopu ovog rada je korištena Xilinx pločica [17] koja koristi ARM Cortex-R5 procesor koji ima istu arhitekturu kao Cortex-R4 procesor te se mogao koristiti za emulaciju Siemensove pločice. Najveće razlike između ta dva procesora su da Cortex-R5 ima dvije jezgre i neke dodatne funkcionalnosti koje se mogu isključiti dok Cortex-R4 ima samo jednu jezgru i manji skup funkcionalnosti.

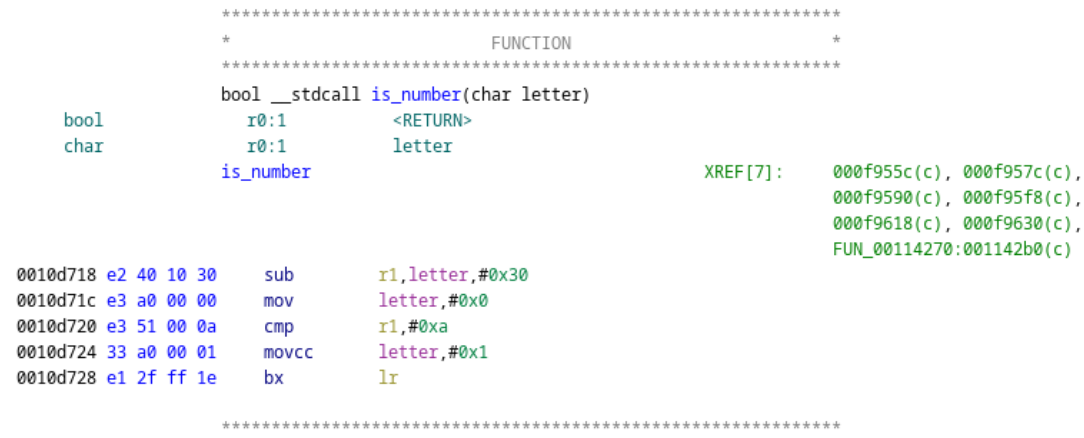
Dalje, pošto PLC ima mnogo perifernih uređaja, uspješna emulacija mora simulirati i ponašanje tih perifernih uređaja. To je moguće napraviti putem potpune implementacije funkcionalnosti ili putem *mockova*. QEMU nam omogućava implementiranje ponašanja perifernih uređaja u *system* emulaciji. Može se za svaku memorijsku lokaciju odrediti koja se funkcija poziva kada se na tu lokaciju čita ili piše. U trenutku pisanja ili čitanja na tu memorijsku lokaciju, QEMU predaje kontrolu implementiranoj funkciji koja će u ovisnosti o stanju sustava napraviti neku aktivnost ili vratiti neku vrijednost. Da bi bilo moguće implementirati ove funkcije, potrebno je što detaljnije poznavati kako periferni uređaj radi. Bez specifikacije samog sustava koja nije javno dostupna, perifernim uređajima se može jedino pristupati kao prema crnim kutijama. Moguće je jedino analizirati kako kôd PLC-a interagira s perifernim uređajima.

Za analiziranje ponašanja ovih perifernih uređaja nam mogu pomoći alati za statičku analizu kôda. Bilo bi poželjno koristiti i alate za dinamičku analizu, kada bi bilo moguće pokrenuti sam *firmware* na računalu, ali pošto je cilj rada upravo pokretanje *firmwarea* na računalu, to je ipak nemoguće. Popularni alati koji se koriste za statičku

analizu kôda su Ghidra [18] i IDA Pro [19]. Oba alata olakšavaju analiziranje izvršnih datoteka, ali je u sklopu ovog rada bio korišten alat Ghidra zato što je besplatan, iako ima sličan skup funkcionalnosti kao IDA Pro.

3.2. Alat Ghidra

Ghidra je alat otvorenog kôda za statičku analizu izvršnog kôda. Podržava mnogo arhitektura uključujući ARMv7-R arhitekturu. Sposoban je prepoznati naredbe i funkcije te korisniku pruža niz funkcionalnosti koje olakšavaju analizu. Od najkorisnijih su komentari, pretraživanje nizova znakova, prikaz povratnih referenci, stablo pozivanja funkcija, izrada memorijske mape, dodavanje oznaka i dodavanje korisnički definiranih struktura podataka. Na slici 3.1 je prikazan izgled glavnog ekrana za analizu izvršnog kôda u alatu Ghidra.

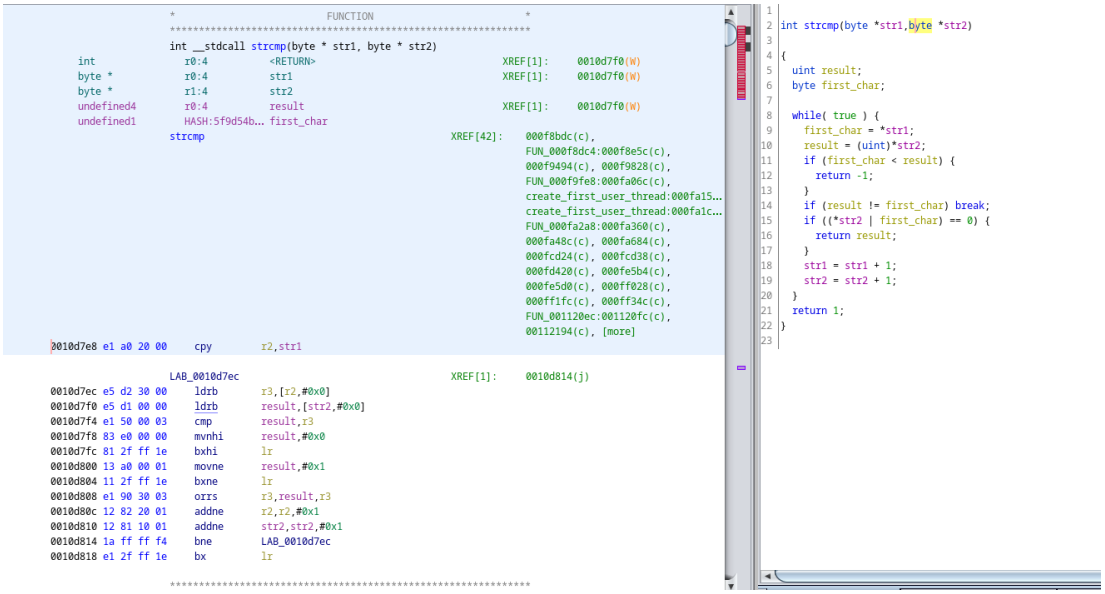


Slika 3.1: Izgled glavnog ekrana u alatu Ghidra

Na vrhu prikazane slike je komentar koji označava da se na ovom mjestu nalazi početak funkcije. Definicija funkcije zajedno s povratnim argumentom, načinom pozivanja, imenom i argumentima su ispod komentara. Potom su u zelenoj boji zapisani povratna vrijednost funkcije i argumenti funkcije. Informacije koje su dane uz povratne vrijednosti i argumente su tip podataka, registar u kojem su spremljeni, širina u oktetima te ime argumenta. Tako se za povratnu vrijednost može iščitati da se radi o tipu podatka `bool` koji je velik 1 oktet te da će povratna vrijednost biti spremljena u registru `R0`. Potom su u 4 stupca dani podatci o pronađenim naredbama. U prvom stupcu u crnoj boji je ispisana trenutna adresa u izvršnoj datoteci, u drugom stupcu vrijednost memorije na toj lokaciji, u trećem prepoznata naredba te u četvrtom argumenti te

naredbe. Na desnoj strani u zelenoj boji se nalazi popis povratnih referenci. One pokazuju koja mjesta u izvršnoj datoteci na neki način referenciraju lokaciju `0x0010d718`, tj. početnu adresu funkcije. U ovom slučaju povratne reference daju informaciju koje funkcije pozivaju funkciju `is_number`.

Funkcionalnost koja Ghidru ističe od drugih alata je integrirani dekompajler koji izvršni kôd pretvara u kôd jezika C. Ghidra izvršni kôd prevodi u svoj međukôd zvan *P-code*. Potom se *P-code* koristi za izgradnju apstraktnog sintaksnog stabla koje se koristi za određivanje pripadnog C kôda. Sposoban je prepoznati koje se lokacije ili registri koriste kao varijable te odrediti kontrolu toka programa. Sve to omogućuje puno lakše prepoznavanje funkcionalnosti neke funkcije. Na slici 3.2 je prikazana jedna funkcija i njezin pripadni dekompajlirani kôd. Važno je spomenuti da Ghidra sama ne generira vrlo smislena imena varijabli već korisnik sam mora promijeniti imena varijabli, a nekada stvoriti i nove varijable ili promijeniti tipove podataka. Alat će napravljene promjene propagirati na ostatak izvršne datoteke. Tako će primjerice varijable koje su u nekoj funkciji dane kao argument funkcije `strcmp` isto imati tip podatka `byte *`.



Slika 3.2: Ghidrin dekompajler

Konačno, Ghidra omogućuje vrlo laganu navigaciju po programu koristeći bilješke, imena funkcija ili memorijske lokacije. Čak i kada se radi s vrlo velikim datotekama, jednom zabilježenu lokaciju poslije nije teško naći.

3.3. ARM Cortex-R4 r1p3

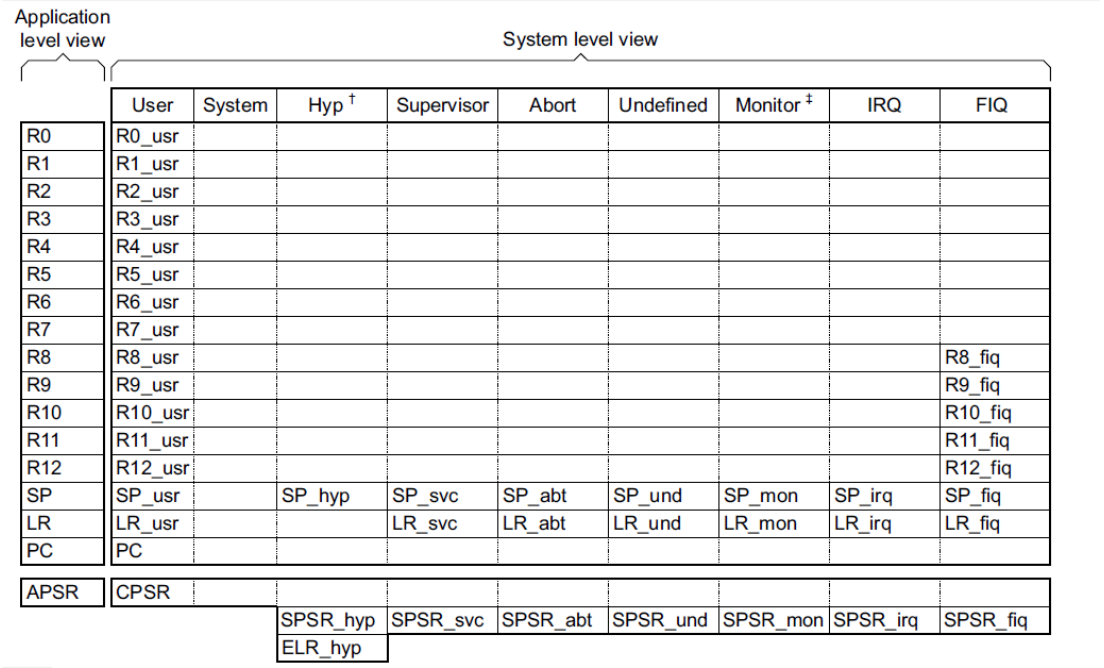
Još jedna bitna stvar za emulaciju je poznavanje ponašanja i strukture procesora kojeg se koristi. Ovo može biti izrazito bitno kada koristimo pločicu koju nismo sami razvili. Ako nešto pođe po krivu tijekom emuliranja izvršavanja programa i neka naredba se ne uspije izvršiti, lakše je odrediti zašto se to dogodilo, ako je primjerice poznato da se pokušavalo pristupiti nekoj funkcionalnosti koja nije implementirana na emuliranom procesoru. Dodatna prednost je da će se tako olakšati i analiza izvršnog kôda. ARM nudi 3 profila procesora [20]:

1. Profil A, procesori namijenjeni za visoku učinkovitost, koriste se u računalima i mobitelima,
2. profil R, procesori namijenjeni za rad u stvarnom vremenu,
3. profil M, procesori namijenjeni za ugradbene sustave.

Arhitektura Cortex-R4 r1p3 procesora je ARMv7-R [21]. To je *reduced instruction set computer* (RISC) arhitektura u kojoj postoje dva načina rada za naredbe. ARM način rada gdje su dostupne sve naredbe te je svaka naredba široka 4 okteta i *thumb* način rada kod kojeg je dostupan ograničen skup naredbi te je svaka naredba široka 2 okteta. *Thumb* način rada se uglavnom koristi u slučaju da je bitno da naredbe ne zauzimaju puno memorije.

U ARMv7-R arhitekturi [22] postoji 17 registara označenih s `R0-R15` i `APSR` koji su dostupni u svim procesorskim načinima rada. Registri `R0-R12` su registri za općenitu uporabu. Registar `R13` se koristi kao pokazivač na stog. Registar `R14` se koristi kao takozvani *link register* te se u njega stavlja adresa na koju se treba vratiti nakon kraja trenutne funkcije. Moguće ga je koristiti kao registar za općenitu uporabu, ako se u njega prije kraja funkcije vrati povratna adresa. Registar `R15` je programsko brojilo. Konačno, registar `APSR` je statusni registar. Tijekom poziva funkcije, prva 4 argumenata se šalju putem registara `R0-R3`. Ako je potrebno poslati još argumenata, oni se šalju putem stoga.

Postoje dodatni registri koji su dostupni ovisno o procesorskom načinu rada. Na slici 3.3 su prikazani svi procesorski načini rada te njihovi pripadni registri. Kada je neko od polja u tablici prazno, to označava da se koristi isti registar kao u *user* načinu rada. Ako polje nije prazno, registar koji se koristi se mijenja na pripadni registar u tom načinu rada. Neke arhitekture ne implementiraju sve procesorske načine rada.



Slika 3.3: Registri u ARM procesoru [23]

Svaki od načina rada je povezan uz neku iznimku u sustavu. To su:

- *Reset*, iznimka koja označuje paljenje sustava, procesor se postavlja u *Supervisor* način rada,
- prekidi, programski prekidi postavljaju procesor u *Interrupt Request* (IRQ), a sklopovski prekidi u *Fast Interrupt Request* (FIQ) način rada,
- neuspješno dohvaćanje podataka, u *Data Abort* način rada,
- neuspješno izvršavanje instrukcije, u *Prefetch Abort* način rada,
- izvršavanje nepostojeće naredbe, u *Undefined Instruction* način rada,
- *Supervisor* poziv (SVC), *Secure Monitor* poziv (SMC) i *Hypervisor* poziv (HVC) u pripadne načine rada.

Svi načini rada imaju svoj pokazivač stoga, a većina ima i svoj *link register*. Zanimljivo je primijetiti da FIQ osim registara `SP` i `LR` ima i svoje registre `R8-R12`. Razlog tome je da je taj način rada namijenjen za brze obrade prekida te se takvom implementacijom treba puno manje vremena provoditi na promjenu konteksta kada se dogodi prekid.

Važno je da se na početku emulacije emulirani procesor ponaša isto kao i stvarni procesor tijekom paljenja. To se naravno mora konfigurirati u izvornom kôdu alata QEMU. Većina konfiguracije na ARMv7-R arhitekturi se postavlja u *System Control*

registru (SCTLR). Promjenom bitova u `SCTRL` registru se uključuju i mijenjaju funkcionalnost. Neke od stvari koje se mogu konfigurirati su:

- Prihvaćaju li se iznimke u ARM ili *thumb* načinu rada,
- mogu li se maskirati FIQ-ovi,
- bit `EE`, endianness iznimaka,
- bit `VE`, koristi li se vektorska tablica za prekide ili implementacijski definirane vrijednosti,
- bit `V`, hoće li početna adresa za prihvaćanje prekida biti `0x00000000` ili `0xFFFF0000`,
- bit `M`, je li uključen *memory management unit* (MMU).

Procesor Cortex-R4 r1p3 na S7-1200 PLC-u prihvaća iznimke u ARM načinu rada i ne maskira FIQ-ove. Iznimke se prihvaćaju u *big-endian* načinu rada i koristi se vektorska tablica koja počinje od `0x00000000` za prihvaćanje iznimaka. Procesor nema MMU već *memory protection unit* (MPU). Putem MPU-a se određuju memorijske regije u adresnom prostoru te način na koji se može pristupati prema njima. Ovo uključuje izvršavanje, čitanje i pisanje na memorijske regije u korisničkom i privilegiranom načinu rada te neke dodatne privilegije.

4. Rezultati analize

U ovom poglavlju se detaljnije objašnjava ponašanje *bootloadera* i *firmwarea*. Navedena opažanja su rezultat analize pomoću alata Ghidra. Pošto je vrlo zahtjevno analizirati velike izvršne programe bez dodatnih informacija o dijelovima programa, u poglavlju se isto tako diskutira na koji je način nađena svrha velikih dijelova izvršne datoteke. Zatim se obrađuju pronađena sučelja prema perifernim uređajima i načini na koje se zaključilo kakvo je ponašanje uređaja. Na kraju se prikazuju rezultati implementacije tih sučelja u emulaciji.

4.1. Ponašanje *bootloadera*

Izvršavanje instrukcija *bootloadera* je potrebno pratiti od adrese `0x00000000` zato što se na tu adresu postavlja programsko brojilo pri ponovnom paljenju procesora. Pošto je poznato memorijsko mapiranje perifernih uređaja, može se vidjeti s kojim uređajima *bootloader* komunicira. U početku se inicijalizira *input output controller* (IOC) i radna memorija. Potom se postavljaju pokazivači stoga za svaki od načina rada. *Vectored interrupt controller* (VIC), sklop za prihvaćanje prekida, se postavlja tako da se onemogućuju svi prekidi osim jednog, a jedini prekid koji je omogućen ne radi ništa. *Bootloader* između ostalog postavlja i konfiguraciju na vrlo niskoj razini kao što je gašenje nekih optimizacija na procesoru. Pošto se pomoću *bootloadera* vrši učitavanje *firmwarea*, u njemu se inicijalizira i CRC tablica za CRC algoritam kojim se provjerava je li *firmware* na neki način modificiran. CRC algoritam prikazan na ispisu 4.1 koji se koristi u *bootloaderu* i *firmwareu* nije standardan CRC algoritam.

```
unsigned int calc_crc(unsigned char *start, unsigned
   char *end) {
    unsigned int crc_checksum = 0xffffffff;
    unsigned char curr_byte;
    unsigned char *current;

    for (current = start; current <= end; current++) {
        curr_byte = *current;
        crc_checksum = *(unsigned int *)
           (&crc_table[curr_byte ^ crc_checksum >>
           0x18]) ^ crc_checksum << 8;
    }

    return crc_checksum;
}
```

Ispis 4.1: CRC algoritam

U verzijama *bootloadera* starijim od 4.2.1 postoji ranjivost koja omogućuje udaljeno izvršavanje kôda [12]. Naime, *bootloader* bi u jednom trenutku izvršavanja $0,5$ sekundi čekao niz znakova `MFGT1` s UART sučelja. Ako ne primi te znakove, *bootloader* će provesti ostatak inicijalizacije te učitati *firmware* u radnu memoriju. Ako se primi taj niz znakova, *bootloader* preko UART-a šalje niz znakova `CPU-` i omogućava posebne funkcionalnosti. To su privilegirane funkcionalnosti koje su vjerojatno bile namijenjene za Siemensove zaposlenike te dopuštaju provjeru integriteta izvršnog kôda, rada perifernih uređaja, nadogradnju *firmwarea* itd. Jedna od funkcionalnosti dopušta pisanje funkcije u memoriju PLC-a te kasnije i pozivanje te funkcije. Time je omogućeno izvršavanje proizvoljnog kôda. Ove privilegirane funkcionalnosti su maknute u novijim verzijama *bootloadera*.

Konačno, *bootloader* premješta većinu *firmwarea* u radnu memoriju, a dio *firmwarea* koji spada u `.exec_in_lomem` odjeljak stavlja u ITCM te predaje kontrolu *firmwareu*.

4.2. Ponašanje *firmwarea*

Firmware, slično kao i *bootloader*, inicijalizira vrijednosti pokazivača stoga, ali ovaj put s drukčijim vrijednostima. Pošto se *bootloader* mijenja rjeđe nego *firmware*, moguće je da su na njima radili različiti timovi pa zbog toga nisu iste vrijednosti pokazivača stoga. Potom se postavlja MPU i inicijalizira se `.bss` odjeljak. Time završava osnovna inicijalizacija u *firmwareu* i poziva se funkcija za pokretanje operacijskog sustava na S7-1200 PLC-u koji se zove ADONIS. Osim mnoštva podsustava koji se inicijaliziraju u ovoj funkciji, još jedna smjernica da se radi o početnoj funkciji operacijskog sustava su nizovi znakova koji se nalaze na različitim dijelovima ove funkcije. U slučaju greške u registar `R0` se stavlja adresa niza znakova prikazanog na ispisu 4.2 i poziva se funkcija koja ne radi ništa, a u slučaju uspješne inicijalizacije operacijskog sustava se upisuje adresa niza znakova prikazan na ispisu 4.3. Beskorisna funkcija koja se poziva se vjerojatno koristi tijekom razvoja sustava za brzo određivanje gdje je došlo do greške stavljanjem prekidne točke na tu funkciju.

```
debug_func("Error during system initialization.\n");
```

Ispis 4.2: Poziv funkcije kod neuspješne inicijalizacije ADONIS-a

```
debug_func("ADONIS boot successful, starting first user
    thread...\n");
```

Ispis 4.3: Poziv funkcije kod uspješne inicijalizacije ADONIS-a

ADONIS je operacijski sustav koji je do sada samo viđen na S7-1200 *firmwareu*. Analizom izvršne datoteke pronađeno je da podržava dretve, monitor, prekide, upravljanje satom, zapisivanje događaja, datotečni sustav, dinamičko upravljanje memorijom, mrežnu komunikaciju te brojne druge mehanizme. Generalna funkcionalnost početne ADONIS funkcije je prikazana na ispisu 4.4.

```
void adonis_startup() {
    int error;

    error = basic_testing();
    error = call_initialization_functions();
    error = create_first_user_thread();

    if (error) {
        log_error();
    } else {
        start_first_thread();
    }

    while (1);
}
```

Ispis 4.4: ADONIS početna funkcija

Prva zanimljiva funkcija je funkcija `call_initialization_functions`. U njoj se poziva niz funkcija koje inicijaliziraju podsustave ADONIS operacijskog sustava. U to su uključeni datotečni podsustav, sustav za dinamičko upravljanje memorijom jezgre i aplikacija, raspoređivač dretvi, prekidni podsustav, dnevnik događaja te drugi.

U drugoj funkciji, `create_first_user_thread`, se stvara prva dretva koja se zove `ADN_SYS_USR_INIT`. Funkcija koju ova dretva izvršava je `th_initial`. Adresa te funkcije se isto tako može naći u tablici 2.2 gdje je prikazano memorijsko mapiranje *firmwarea*. Jedna zanimljivost koja se može uočiti u ovom trenutku je da se u ADONIS-u korisne informacije često zapisuju u obliku heksadekadskih konstanti. Tako primjerice polje koje određuje je li novostvorena dretva neovisna ili ponekad mora čekati druge dretve da završe s izvođenjem prije nego što se može nastaviti izvoditi može biti jednako konstanti `0xDE1A3CED` ili `1019AB1E`, tj. *detached* ili *joinable*.

U slučaju da se dogodila greška tijekom izvođenja ovih funkcija, informacija o grešci će se zabilježiti u dnevniku događaja. Inače će se pozvati raspoređivač koji će izabrati jednu od dretvi za izvođenje što će u početnom slučaju biti dretva koja izvodi funkciju `th_initial`, tj. *initial thread*. U njoj se većinom dovršava inicijalizacija nekih podsustava i nekoliko se dretvi stavlja u raspoređivač, primjerice sustavska pozadinska dretva i pozadinska dretva za ulazno izlazne operacije. Analiza funkcije

th_initial nije do kraja napravljena zbog njezine veličine, koja broji oko 2200 ARM naredbi. Može se pretpostaviti da bi završetkom ove funkcije PLC trebao biti u stabilnom stanju i spreman za korištenje.

4.3. Dobivanje konteksta

Prije analize izvršnog programa vrlo je bitno saznati što više informacija koje mogu pomoći pri analizi. Takve informacije u izvršnim programima najčešće dolaze u obliku nizova znakova koji otkrivaju nešto o stanju u kojem se sustav nalazi. Već pokazani primjer ovakvog niza znakova su poruke koje se mogu naći u funkciji adonis_startup. Osim ovakvog primjera gdje se u registar direktno stavlja adresa nekog niza znakova, postoje i drugi načini na koje se može saznati nešto o svrsi dijela programa kojeg se analizira.

4.3.1. Dnevnik događaja

U prijašnjem potpoglavlju je spomenuto da ADONIS sadrži dnevnik događaja. Kada se dogodi značajniji događaj u funkcioniranju PLC-a, ovisno o određenim uvjetima, to se zapiše u dnevniku. Kôd za zapisivanje u dnevnik je prikazan na ispisu 4.5. Za svaki pokušaj zapisivanja u dnevnik, provjerava se je li razina za zapisivanje u dnevnik spremljena u varijabli CURRENT_LOG_LEVEL dovoljno visoka, tj. viša od konstante LOG_URGENCY_LEVEL. Varijabla CURRENT_LOG_LEVEL je jednaka za cijeli jedan podsustav. U slučaju da je dovoljno visoka, u dnevnik se zapisuje identifikacijski broj podsustava u kojem se dogodio događaj, važnost poruke te linija izvornog kôda kod kojeg se dogodio događaj. Linija izvornog kôda koja se daje kao argument kod poziva funkcije zapisa u dnevnik događaja naravno nije jednaka liniji izvršnog kôda.

```
if (LOG_URGENCY_LEVEL < CURRENT_LOG_LEVEL) {
    log_event(subsystem_id, log_level, line_of_code);
}
```

Ispis 4.5: ADONIS zapisivanje u dnevnik

Dnevnik operacijskog sustava ADONIS može otkriti puno informacija o funkciji koja se analizira, ali isto tako i o cijelom sustavu. Prvo, govori koliko je važan dio kôda kojeg se analizira. Što je LOG_URGENCY_LEVEL bliži 0, to je ta funkcionalnost

važnija cijelom operacijskom sustavu. Ako je primjerice LOG_URGENCY_LEVEL jednak 0 može se zaključiti da je taj dio kôda nužan za točno funkcioniranje sustava, pošto se u tom slučaju događaj zapisuje bez obzira na razinu podsustava. Ako se s druge strane samo propuštaju događaji niže važnosti, primjerice 6 ili 7, može se zaključiti da se radi o jednostavnoj obavijesti ili nekoj aktivnosti koja nije jako bitna. Varijabla CURRENT_LOG_LEVEL je koristan pokazivač koliko je neki podsustav bitan cijelom operacijskom sustavu. Nažalost, u ADONIS-u, samo nekoliko jezgrenih sustava ima razinu dnevnika postavljenu na 1, dok svi drugi podsustavi imaju razinu 4 tako da varijabla CURRENT_LOG_LEVEL ne pruža puno novih informacija. Treća i najvažnija informacija koja se može dobiti od zapisivanja u dnevnik je ime podsustava u kojem se zapisivanje događa. U *firmwareu* se nalazi niz koji sadrži imena svih podsustava, njihove identifikacijske brojeve te njihove razine dnevnika. Varijabla subsystem_id je jedan od identifikacijskih brojeva u nizu te se preko njega može otkriti o kojem se podsustavu radi. Preko te se varijable mogu jednostavno naći dijelovi operacijskog sustava ADONIS koji mogu biti zanimljivi. Primjerice, ako se traži izvršni kôd koji funkciji log_event daje argument 0x66 za subsystem_id, može se pronaći svaka funkcija koja se bavi nadogradnjom *firmwarea*, zato što identifikacijski broj 0x66 pripada podsustavu koji se zove FW_UPDATE. Ovo naknadno znači da se veliki dio *firmwarea* može razdvojiti na međusobno disjunktne skupine funkcija od kojih svaka predstavlja cjelokupnost jednog podsustava unutar ADONIS-a. Konačno, pošto je moguće prepoznati podsustav kojeg se analizira, isto tako je moguće zaobići funkcije koje pripadaju podsustavima koji nisu bitni za emulaciju. Tako primjerice podsustav ADONIS_MUTEX sigurno ne komunicira s perifernim uređajima te zbog toga nije potrebno napraviti analizu funkcija koje njemu pripadaju. Na ispisu 4.6 je prikazan poziv funkcije za zapisivanje u dnevnik koji je jednak ispisu 4.5, ali koji sada pokazuje koji podsustav se analizira te koliko je važan događaj koji se zapisuje.

```
if (LOG_LEVEL_FATAL < ADONIS_SYSTEM_LOG_LEVEL) {
    log_event([ADONIS_SYSTEM], LOG_LEVEL_ERROR, 0xbeef);
}
```

Ispis 4.6: ADONIS zapisivanje u dnevnik sa stvarnim vrijednostima

4.3.2. Dretve

Dretve su još jedan izvor informacija o funkcioniranju sustava. Slično kao i kod dnevnika događaja, za dretve isto postoji polje u *firmwareu* koje sadrži informacije o dretvama. Svaki element polja sadrži ime dretve, identifikacijski broj dretve te prioritet dretve. Za razliku od polja kod dnevnika događaja, gdje je većina podsustava imala istu razinu dnevnika te je informacija o važnosti podsustava bila uvelike skrivena, prioritet dretve za svaku dretvu mora biti postavljen u skladu s važnosti podsustava zato što u protivnom raspoređivač neće dobro raditi. Pri stvaranju dretve se u njezin pripadni objekt stavlja ime dretve, identifikacijski broj i funkcija koju dretva obavlja. To znači da se za svaku funkciju koju obavlja neka dretva može znati koja je njezina općenita svrha preko imena dretve te koliko je ta funkcija važna za sam operacijski sustav preko prioriteta dretve.

4.4. Otkrivena sučelja

Rezultat analize izvršnog kôda su specifikacije sučelja koje će se potom implementirati u alatu QEMU. Iako u tablici 2.3 ima puno perifernih uređaja, veliki broj njih se može *mockati*, tj. implementirati na način da uvijek javljaju da je neka operacija prošla uspješno ili da vraća točno one informacije koje *firmware* traži. Dodatno, neki od tih uređaja su samo memorijski prostori kod kojih se može čitati i pisati te za njih ne postoji posebno sučelje za komunikaciju. Takvi uređaji nisu nužni za osnovni rad emuliranog PLC-a. Jednom kada se emulacija nalazi u stabilnom stanju i nema neočekivanih grešaka, mogu se implementirati upravljački programi za preostale uređaje. Puno su važnija sučelja bez kojih se PLC ne može inicijalizirati kao što je primjerice sat. U ovom potpoglavlju slijedi niz specifikacija za pronađene uređaje.

4.4.1. Sučelje za sat

Unutar ADONIS-a se može komunicirati s 14 satova čija su memorijska područja označena s `MAP_ITIMER0` - `MAP_ITIMER13`. Memorijsko područje svakog sata je veliko `0x10` okteta, a svi satovi se nalaze u većem memorijskom području `MAP3_TIMERS` koje je veliko `0x15C` okteta. U memorijskom području pojedinačnog sata se može podesiti rad tog sata, dok se u području svih satova može dodatno podesiti rad cijelog podsustava. U području svih satova se isto tako nalazi nadzorni alarm (engl. *watchdog timer*). To je sat koji se neprestano iznova postavlja i u pravilnom izvođenju programa nikad ne

odbroji sve otkucaje prije sljedećeg postavljanja. Kod sustava za rad u stvarnom vremenu kao što je PLC, ako se neki zadatak izvodi dulje nego što je očekivano, nastavak izvođenja programa može biti štetnije od zaustavljanja. Zato nadzorni alarm generira prekid kada odbroji sve otkucaje te će PLC nakon toga zaustaviti rad ili poduzeti neku drugu prikladnu akciju.

Sučelje za upravljanje pojedinačnim satom ima 4 polja s kojima se može komunicirati. Ako se u Ghidri analizira na koji način se pristupa tim poljima, može se vidjeti da se s prvog polja isključivo čita, na drugo i treće polje se isključivo piše te se na četvrto polje i čita i piše. Način na koji sat radi je pronađen analizirajući ponašanje raspoređivača, što je i očekivano pošto raspoređivač za sustave za rad u stvarnom vremenu mora zadovoljiti vremenska ograničenja poslova koji se obavljaju. Pretpostavlja se da raspoređivač ima sljedeće mogućnosti:

1. Postavljanje broja otkucaja u satu tako da se postavi vrijeme do sljedećeg prekida za raspoređivanje,

2. čitanje ili računanje broja otkucaja koji su prošli od zadnjeg raspoređivanja, ako je raspoređivač pozvan zbog primjerice prijevremenog završetka dretve,

3. pokretanje i zaustavljanje sata.

Način na koji raspoređivač komunicira sa sučeljem sata je prikazan na ispisu 4.7. Ukratko, raspoređivač dobavlja vrijednost od sata koju sprema u varijabli `timer_val`. Ako je četvrto polje sata uključeno, ta varijabla se postavlja na `0`. Potom se koristi tijekom raspoređivanja u funkciji `schedule_timer` koja vraća vrijednost koja se zapisuje u sučelje sata.

```
void scheduler() {
    unsigned long timer_val, fun_return;
    adonis_timer *sched_timer = (adonis_timer *)
       0xfffbb0a0;

    timer_val = sched_timer->field_1;
    if (sched_timer->field_4) {
        timer_val = 0;
    }
    sched_timer->field_3 &= 0xfffffffe;
    sched_timer->field_4 = 0;


    fun_return = schedule_threads(timer_val);


    sched_timer->field_2 = fun_return;
    sched_timer->field_3 =0x1d;
}
```

Ispis 4.7: ADONIS raspoređivač

Pošto je ovim kôdom prikazana sveukupna komunikacija koju raspoređivač postavlja sa satom, moguće je zaključiti nekoliko stvari. Jedina vrijednost koja se koristi u sklopu raspoređivanja je vrijednost polja `field_1` te to mora biti broj otkucaja koje je sat već napravio ili otkucaja koji su ostali. Na polje `field_2` se zapisuje neka varijabilna vrijednost, tj. izlaz funkcije `schedule_threads`. Vrlo je vjerojatno da se radi o postavljanju otkucaja koje sat treba napraviti do sljedećeg poziva raspoređivača za taj posao. Polja `field_3` i `field_4` oba mogu biti polja za pokretanje sata. No, pošto se čini da se polje `field_4` postavlja na vrijednost `0` prije nego što se u sat postavi broj otkucaja, vjerojatnije je da se radi o nekakvom stanju sata. Polje `field_3` bi u tom slučaju bilo polje za pokretanje odnosno zaustavljanje sata. S ovim pretpostavkama daljnjom je analizom potvrđeno da je `field_1` broj otkucaja koje sat još treba napraviti. Ispis 4.8 opisuje kako izgleda predloženo sučelje za komunikaciju sa satom, a ispis 4.9 kako prijašnji kôd izgleda s predloženim sučeljem.

```
struct adonis_timer {
        uint ticks_left;
        uint ticks_setter;
        uint mode;
        uint is_finished;
};
```

Ispis 4.8: ADONIS sučelje za sat

```
void scheduler() {
    unsigned long ticks_left, next_ticks_val;
    adonis_timer *sched_timer = (adonis_timer *)
       0xfffbb0a0;

    ticks_left = sched_timer->ticks_left;
    if (adonis_timer->is_finished) {
        ticks_left = 0;
    }
    adonis_timer->mode &= TIMER_PAUSE; // Zaustavljanje
       sata
    adonis_timer->is_finished = 0;


    next_ticks_val = schedule_threads(ticks_left);


    sched_timer->ticks_setter = next_ticks_val;
    adonis_timer->mode = TIMER_START;
}
```

Ispis 4.9: ADONIS raspoređivač s predloženim sučeljem

4.4.2. Sučelje za UART komunikaciju

Sučelje za UART komunikaciju nije nužno za rad emuliranog PLC-a, no može se lako saznati kako PLC komunicira s UART sučeljem analizom kôda. Programski, UART

komunikacija nije kompliciranа. Može se očekivati kôd sličan onome na ispisu 4.10. Prvo se provjerava stanje uređaja i čeka se da uređaj bude spreman za čitanje ili pisanje što je predstavljeno varijablom `uart_status`. U slučaju da je uređaj spreman, s njega se može čitati ili pisati tako što se pristupa pripadnoj memorijskoj lokaciji, predstavljenom varijablom `uart_data`.

```
// Citanje
if (*uart_status & READ_RDY) {
    next_character = *uart_data;
}

// Pisanje
if (*uart_status & WRITE_RDY) {
    *uart_data = next_character;
}
```

Ispis 4.10: Generalni izgled komunikacije s UART sučeljem

U *bootloaderu* postoji dobro dokumentiran primjer UART komunikacije. Radi se o posebnim funkcionalnostima opisanim u poglavlju s opisom ponašanja *bootloadera* gdje se preko UART sučelja čeka niz znakova `MFGT1`. Na ispisu 4.11 je prikazano kako UART komunikacija izgleda u toj funkciji.

```
// Citanje
if ((*0xfffb8018 & 0x10) == 0) {
    next_character = (char) *(0xfffb8000);
    status = *(0xfffb8004) & 0xb;
} else {
    status = 0x1000;
}

// Pisanje
do {
} while ((*0xfffb8018 & 0x20) != 0);
*(0xfffb8000) = next_character;
```

Ispis 4.11: UART čitanje i pisanje u *bootloaderu*

Kao što je očekivano, komunikacija na ispisu 4.11 je slična ispisu 4.10. Može se zaključiti da je lokacija `0xfffb8018` istovjetna varijabli `uart_status`, a lokacija `0xfffb80000` varijabli `uart_data`. Osim toga, konstante `READ_RDY` i `WRITE_RDY` su redom `0x10` i `0x20`. Komunikacija s memorijskom lokacijom `0xfffb8004` nije predviđena, ali pretpostavlja se da se radi o statusu uspješnosti operacije pisanja ili čitanja. Pošto se u *bootloaderu* s UART-a čita u petlji čekalici, taj mehanizam je potreban da se zna je li čitanje bilo uspješno ili ne. Predloženo programsko sučelje za UART komunikaciju je prikazano na ispisu 4.12. Na polja `uart_config_x` koja se nalaze između lokacije `0xfffb8004` i `0xfffb8018` se samo zapisuje tijekom incijalizacije te se pretpostavlja da se radi o nekoj vrsti konfiguracije uređaja.

```
struct uart_com {
        unsigned int data;
        unsigned int op_status;
        unsigned int uart_config_1;
        unsigned int uart_config_2;
        unsigned int uart_config_3;
        unsigned int uart_config_4;
        unsigned int state;
};
```

Ispis 4.12: ADONIS UART sučelje

4.4.3. Sučelje za komunikaciju s NAND *flashom*

Memorijsko mapiranje za NAND *flash* ne postoji u tablici 2.3. Bez obzira na to, i *firmware* i *bootloader* komuniciraju s tim uređajem. Komunikacija s NAND *flashom* se ne izvodi na kontinuiranom memorijskom području, kao što je slučaj s drugim uređajima, već se dijelovi sučelja nalaze na vrlo različitim memorijskim lokacijama počevši od adrese `0x80000000` na više.

Sučelje za NAND *flash* se može otkriti analizirajući dio kôda u *bootloaderu* koji je zadužen za premještanje *firmwarea* u radnu memoriju i ITCM. Pošto kôd mora prvo parsirati *firmware* zaglavlje, ta se funkcija može lagano naći tako da se u *firmwareu* pretražuje naredba koja u gornji dio bilo kojeg registra stavlja konstantu `0x5D1B` te u donji konstantu `0x4153`. Tada će u registru biti konstanta `0x5D1B4153` što je magični

broj *firmware* zaglavlja i prva stvar koja se traži u zaglavlju. Na ispisu 4.13 je prikazan pojednostavljen izgled komunikacije s NAND sučeljem gdje su izdvojeni samo relevantni dijelovi.

```
int copy_firmware(int var1, int var2) {
    int var3 = var2 | var1 << 6;
    unsigned int var4, counter = 0;
    *0x80918000 = (var3 << 0x10) >> 0x18 | (var3 <<
       0x18) >> 0x10;
    *0x80000380 = 0;

    while ((*0x80280000 & 0x4000) != 0x4000) {};

    if ((*0x80280000 & 0x100) == 0x100)
        return -1;

    do {
        var4 = *0x80080000;
        copy_to_mem(var4);
        counter++;
    } while (counter < 512);
}
```

Ispis 4.13: ADONIS kopiranje *firmwarea*

Pošto se sadržaj varijable `var4`, tj, adrese `0x80080000` na kraju stavlja u memoriju, lagano je zaključiti da se u njoj nalaze podatci koji se čitaju iz NAND *flasha*. Zanimljivo je primijetiti da se u `do-while` petlji čitanje ponavlja 512 puta. Pregledom dokumentacije za taj čip [5], može se zaključiti da se radi o veličini jedne stranice (engl. *page*) od 2 KiB u memoriji. Svaki poziv ove funkcije predstavlja čitanje jedne stranice. Dodatnom analizom dokumentacije, može se pretpostaviti da su varijable `var1` i `var2` vjerojatno indeks bloka na čipu te indeks stranice u bloku. Za adresu `0x80280000` se može pretpostaviti da je lokacija gdje se nalazi status operacije te `0x4000` predstavlja završetak operacije učitavanja stranice, dok `0x100` označava nekakvu grešku. Adresa `0x80918000` je najvjerojatnije redni broj stranice koju se želi učitati. Konačno, adresa

`0x80000380` bi mogla biti signal da je željena adresa učitana te da operacija može započeti. Taj signal je, čini se, aktivan kada je postavljen na vrijednost `0`. Ako se isti kôd prepiše s predloženim oznakama, dobije se ispis 4.14. Pronađeno je i sučelje za pisanje na NAND čip. Koristi se na isti način kao i kada se čita s čipa, samo se umjesto lokacije `0x80080000` koristi adresa `0x80088000`.

```
int copy_firmware(int block_no, int page_in_block) {
    int index = page_in_block | block_no << 6;
    unsigned int data, counter = 0;
    *NAND_ADDRESS = (index << 0x10) >> 0x18 | (index <<
       0x18) >> 0x10;
    *NAND_START_OP = 0;

    while ((*NAND_STATUS & READ_RDY) != READ_RDY);

    if ((*NAND_STATUS & READ_ERR) == READ_ERR)
        return -1;

    do {
        data = *NAND_DATA_GET;
        copy_to_mem(data);
        counter++;
    } while (counter < PAGE_SIZE);
}
```

Ispis 4.14: ADONIS kopiranje *firmwarea* s predloženim oznakama

4.4.4. *Mockovi*

Velik broj sučelja prikazanih na tablici 2.3 nije potrebno implementirati tako da se ponašaju kao pravi uređaji, već kao *mockovi*. *Mockovi* su objekti koji oponašaju rad stvarnog objekta na kontroliran način. To znači da je poznato koje se operacije s *mock* uređajem očekuju te što se minimalno očekuje od *mocka* nakon tih operacija. Često se u kôdu provjerava je li neki uređaj dostupan, uključen, spreman i slično. Pošto nije očekivano da će emulirani uređaj imati periferne uređaje koji ne rade, za takve je slučajeve dovoljno implementirati *mock* koji uvijek vraća očekivani rezultat. Primjer

jednog takvog sučelja se nalazi na početku *bootloadera* gdje se provjerava je li inicijaliziran *input-output controller* (IOC). Pripadni kôd je prikazan na ispisu 4.15. Za nastavak rada emuliranog uređaja je dovoljno implementirati *mock* koji vraća `0x100` kada se čita s adrese `0xfffbc000`. Primjeri sučelja s tablice 2.3 kod kojih se ovakva implementacija može koristiti su `MAP3_DDR_MEMCTL`, `MAP3_FL_MEMCTL`, `MAP3_INPUTS`, `MAP3_OUTPUTS`, `MAP3_PWRSTK` te bilo koja druga sučelja gdje se može osigurati normalan rad vraćanjem vrijednosti koju kôd *firmwarea* očekuje.

```
do {} while ((*0xfffbc000 & 0x100U) == 0);
```

Ispis 4.15: ADONIS IOC provjera

4.5. Rezultati emuliranja

Pošto je za točnu inicijalizaciju sistema potrebno prvo izvršiti *bootloader* pa tek onda *firmware*, prvi je cilj emulirati izvršavanje *bootloadera* do trenutka kada se u njemu predaje kontrola *firmwareu*. Za emulaciju je korištena Xilinxova pločica koja koristi procesor Cortex-R5 s ARMv7-R arhitekturom. Prva poteškoća je da su naredbe u izvršnim datotekama *bootloadera* i *firmwarea* zapisane u *big-endian* poretku, dok Xilinxova emulirana pločica podržava samo *little-endian* poredak [24]. Iako ARMv7-R arhitektura dopušta da se poredak konfigurira, nije nađen način da se to napravi programski. Ovaj problem je zaobiđen tako da je promijenjen poredak u *bootloader* i *firmware* datoteci tako da same datoteku budu u *little-endian* poretku. Nakon toga je emulirani uređaj točno izvršavao instrukcije sve dok nije bilo potrebno komunicirati s perifernim uređajem. Naime, očekivalo bi se da se na nekoj adresi nalazi određena vrijednost koju bi postavio neki periferni uređaj, no pošto u emulaciji periferni uređaj nije implementiran, na toj adresi se nalazila vrijednost `0`. Nakon implementacije jednostavnih uređaja, *bootloader* bi se nastavio izvoditi. Još jedan problem koji se pojavio zbog korištenja emulacije procesora Cortex-R5 umjesto procesora Cortex-R4 je taj da neke naredbe za konfiguraciju rada procesora na niskoj razini nisu implementirane na jednak način. Naime, ARMv7-R arhitektura dopušta da neke funkcionalnosti ovisno o implementaciji budu konfigurirane potpuno drukčijim naredbama. Zato, kada bi se u *bootloaderu* naredbom konfigurirao rad Cortex-R4 procesora da se omogući neka funkcionalnost, na Cortex-R5 procesoru bi to bila nepostojeća naredba. Te naredbe su preskočene na način da se postave na operaciju `nop` naredbu koja ne radi ništa.

Ako se zanemare *mockovi* uređaja, jedino sučelje koje je implementirano je sučelje sata. Za implementaciju se koristio QEMU-ov integrirani sat te programski model temeljen na prijašnje predloženom sučelju. Kada se sat pokrene, zabilježi se trenutni broj otkucaja QEMU-ovog sata. Kada se u kôdu provjerava ako je sat završio s otkucavanjem, pokreće se funkcija u kojoj se računa razlika između trenutne i zabilježene vrijednosti QEMU-ovog sata. Ako je razlika veća od postavljene vrijednosti od koje sat odbrojava, onda će zastavica `is_finished` biti jednaka `1`, a inače će biti `0`.

UART sučelje i NAND sučelje nije implementirano u trenutku pisanja rada. UART sučelje nije niti bilo potrebno implementirati zato što za normalno izvršavanje kôda nije potrebna UART komunikacija. S druge strane, NAND sučelje je potrebno implementirati, ali je zbog nedostatka vremena odabrana jednostavnija opcija mijenjanja izvršnog kôda tako da se ne komunicira s NAND sučeljem i premješta *firmware*, već se *firmware* ručno postavlja na očekivanu lokaciju. S ovim izmjenama, emulirani uređaj uspijeva izvršiti cijeli kôd *bootloadera* i dio kôda *firmwarea*. Ovo ukazuje da se s relativno jednostavnom emulacijom rada uređaja može emulirati izvršavanje značajnog dijela kôda. Naravno, točna emulacija bi zahtijevala da se i najjednostavnija sučelja implementiraju tako da se ponašaju čim više kao stvarni uređaj, ali su i ovakvi rezultati obećavajući.

5. Zaključak

U ovom radu su pokazani načini na koje se može dobaviti *firmware* datoteka Siemensovog S7-1200 PLC-a. Identificirano je sklopovlje koje se nalazi na upravljačkom PCB-u i objašnjen je format *firmware* datoteke. Nakon toga je pomoću alata Ghidra istražena funkcionalnost *bootloadera* i *firmwarea* te su nađeni načini na koje PLC komunicira s perifernim uređajima što je omogućilo djelomičnu emulaciju PLC-a na postojećoj implementaciji pločice. Emulacija PLC-a omogućuje dinamičku analizu rada programa što će olakšati daljnje istraživanje rada PLC-a.

Postoji puno mogućnosti za poboljšanje te je potrebno puno rada da emulacija radi isto kao i PLC. U budućnosti bi se trebalo fokusirati na implementaciji svih perifernih sučelja te dovođenje emulacije u stanje gdje se više ne događaju prekidi u izvođenju programa. Osim toga je poželjno integrirati alate za testiranje programske potpore s emulacijom.

6. LITERATURA

[1] Siemens. 6es7214-1ag40-0xb0, 2022. URL https://mall.industry.siemens.com/mall/en/WW/Catalog/Product/6ES7214-1AG40-0XB0, pristupljeno 15.6.2022.

[2] Thomas Weber. Reverse engineering architecture and pinout of custom asics, 2019. URL https://sec-consult.com/blog/detail/reverse-engineering-architecture-pinout-plc/, pristupljeno 8.6.2022.

[3] Micron Technology. Mobile low-power ddr sdram, 2018. URL https://www.micron.com/products/dram/lpdram/part-catalog/mt46h32m32lfb5-5-it, pristupljeno 8.6.2022.

[4] Serpentine routing, 2021. URL https://jp.nextpcb.com/blog/routing-strategy-in-pcb-layout, pristupljeno 10.6.2022.

[5] Micron Technology. Automotive nand flash memory, 2014. URL https://hr.mouser.com/ProductDetail/Micron/MT29F2G16ABBEAHC-AITE?qs=rrS6PyfT74fSbxwJToeF3w%3D%3D, pristupljeno 8.6.2022.

[6] Integrated Silicon Solution Inc. Is25lq040b datasheet, 2014. URL https://pdf1.alldatasheet.com/datasheet-pdf/view/812868/ISSI/IS25LQ040B.html, pristupljeno 2.6.2022.

[7] Siemens. Release s7-1200 cpu firmware version v4.5, 2021. URL https://support.industry.siemens.com/cs/document/109793280/release-s7-1200-cpu-firmware-version-v4-5?dti=0&lc=en-US, pristupljeno 2.6.2022.

[8] Jean-Baptise Bédrune, Alexandre Gazet, and Florent Monjalent. Supervising the supervisor: Reversing proprietary scada tech. 05 2015. URL https://conference.hitb.org/hitbsecconf2015ams/wp-content/

uploads/2015/02/WHITEPAPER-Reversing-Proprietary-SCADA-Tech.pdf, pristupljeno 15.6.2022.

[9] Jean-Baptise Bédrune. Lzp3 algorithm implementation, 2022. URL https://github.com/jibeee/s7unpack, pristupljeno 15.6.2022.

[10] Lovro Grgurić-Mileusnić. Lzp3 algorithm implementation, 2022. URL https://gitlab.com/lgrguric/siemens_lzp3, pristupljeno 10.6.2022.

[11] flashrom. URL https://www.flashrom.org/Flashrom, pristupljeno 15.6.2022.

[12] Doors of durin, 2019. URL https://i.blackhat.com/eu-19/Wednesday/eu-19-Abbasi-Doors-Of-Durin-The-Veiled-Gate-To-Siemens-S7-Silicon.pdf, pristupljeno 10.6.2022.

[13] Bus pirate. URL http://dangerousprototypes.com/docs/Bus_Pirate, pristupljeno 15.6.2022.

[14] Pomona 5250. URL https://www.digikey.com/en/products/detail/pomona-electronics/5250/745102, pristupljeno 15.6.2022.

[15] Lucian Cojocar, Kaveh Razavi, and Herbert Bos. Off-the-shelf embedded devices as platforms for security research. pages 1–6, 04 2017.

[16] Qemu documentation, 2022. URL https://www.qemu.org/docs/master/, pristupljeno 15.6.2022.

[17] Xilinx. Qemu microblaze github. URL https://github.com/qemu/qemu/tree/master/hw/microblaze, pristupljeno 9.6.2022.

[18] Ghidra, 2022. URL https://ghidra-sre.org/, pristupljeno 15.6.2022.

[19] Ida pro, 2022. URL https://hex-rays.com/IDA-pro/, pristupljeno 15.6.2022.

[20] ARM. Arm architecture profiles. URL https://developer.arm.com/documentation/dui0471/m/key-features-of-arm-architecture-versions/arm-architecture-profiles, pristupljeno 15.6.2022.

[21] ARM. Cortex-r4 and cortex-r4f technical reference manual r1p3. URL https://developer.arm.com/documentation/ddi0363/e/introduction/about-the-architecture?lang=en, pristupljeno 15.6.2022.

[22] ARM. Arm architecture reference manual armv7-a and armv7-r edition. URL https://developer.arm.com/documentation/ddi0406/cd/, pristupljeno 15.6.2022.

[23] Saravana Pandian Annamalai. Arm architecture – registers and exception model, 2015. URL https://www.embien.com/blog/arm-architecture-registers-exception-model/, pristupljeno 15.6.2022.

[24] Xilinx. Xilinx quick emulator user guide, 2020. URL https://xilinx-wiki.atlassian.net/wiki/download/attachments/821395464/QEMU%20User%20Documentation.pdf?version=2&modificationDate=1614274853910&cacheVersion=1&api=v2, pristupljeno 15.6.2022.

Izvršavanje firmwarea PLC-a korištenjem alata QEMU

Sažetak

Ranjivosti u PLC-ovima mogu dovesti do ogromnih materijalnih šteta, ako se zloupotrijebe. Potrebno je testirati programsku potporu PLC-ova tako da se ovakve ranjivosti mogu pronaći. Jedan način kako se to može napraviti je emuliranjem sklopovlja PLC-a te izvršavanjem programa na emulaciji, što može služiti kao baza za buduću integraciju s alatima za testiranje. U tu svrhu su razmatrani načini nabave *firmware* datoteke S7-1200 PLC-a koja se zatim analizirala pomoću alata Ghidra. Dobivene informacije, koje uključuju način na koji PLC komunicira s perifernim uređajima, su iskorištene za djelomičnu emulaciju PLC-a pomoću alata QEMU.

Ključne riječi: PLC, S7-1200, emulacija, statička analiza, QEMU, Ghidra

PLC firmware execution using QEMU

Abstract

PLC software vulnerabilities can lead to enormous material losses if they are exploited. It is necessary to prevent this by testing PLC software. A possible approach is using an emulation tool like QEMU to emulate software execution, which could serve as a basis for future integration with testing tools. For that purpose methods for acquiring the firmware file for the S7-1200 PLC are discussed. The file is then analyzed using Ghidra. Using the obtained information, which includes IO behavior, a partial PLC emulation is implemented.

Keywords: PLC, S7-1200, emulation, static analysis, QEMU, Ghidra