Parcijalno ažuriranje programske podrške na AURIX TC397 platformi

Matijević, Barbara

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij

PARCIJALNO AŽURIRANJE PROGRAMSKE PODRŠKE NA AURIX TC397 PLATFORMI

Diplomski rad

Barbara Matijević

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 20.09.2020.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Barbara Matijević
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1002R, 24.09.2019.
OIB studenta:	12093627901
Mentor:	Izv. prof. dr. sc. Marijan Herceg
Sumentor:	
Sumentor iz tvrtke:	Ivan Mijić
Predsjednik Povjerenstva:	Izv.prof.dr.sc. Ratko Grbić
Član Povjerenstva 1:	Izv. prof. dr. sc. Marijan Herceg
Član Povjerenstva 2:	Dr.sc. Denis Vranješ
Naslov diplomskog rada:	Parcijalno ažuriranje programske podrške na AURIX TC397 platformi
Znanstvena grana rada:	**Telekomunikacije i informatika (zn. polje elektrotehnika)**
Zadatak diplomskog rada:	Parcijalno ažuriranje programske podrške trenutno je važna tema u svijetu automobilske industrije. Kako bi se uštedjelo vrijeme neophodno za testiranje cjelokupne programske podrške, a time skratilo vrijeme za isporuku krajnjim korisnicima, neophodno je razviti mehanizam parcijalnog ažuriranja programske podrške. Cilj ovog rada je napraviti dvije verzije programske podrške. Prva verzija bi bila osnovna i sa sobom bi uključila zadanu funkcionalnost. Druga verzija bi trebala sadržavati samo izmijene vezane za istu tu funkcionalnost, dok bi preostali dio flash memorije mikroupravljača trebao ostati nepromijenjen. Time bi se na praktičan način demonstrirala mogućnost
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	20.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 30.09.2020.

Ime i prezime studenta:	Barbara Matijević
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1002R, 24.09.2019.
Turnitin podudaranje [%]:	8

Ovom izjavom izjavljujem da je rad pod nazivom**: Parcijalno ažuriranje programske podrške na AURIX TC397 platformi**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Marijan Herceg

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Sadržaj:

1. UVOD

Ugradbeni računalni sustavi (engl. *embedded systems*) su sustavi koji predstavljaju kombinaciju sklopovlja i programske podrške, imaju mogućnost dodavanja novih komponenti (mehaničkih ili elektroničkih), a specijalizirani su za određenu namjenu te većinom predstavljaju dio nekog većeg uređaja/sustava. Pojednostavljeno gledano, ugradbeni sustavi sadrže mikroupravljač ili mikroprocesor (s dodatnom periferijom) koji je srce uređaja, nekoliko vrsta memorija te niz komunikacijskih sklopova [1]. Ono što ih karakterizira jest mala potrošnja energije, male dimenzije te obavljanje karakterističnih funkcionalnosti poput upravljanja automatskim sustavima za parkiranje vozila ili nadziranje rada motora. Upravo zahvaljujući ovim karakteristikama ovakvi sustavi su jednostavni za masovnu proizvodnju te se nalaze svuda oko nas. S druge strane, zbog manjka sklopovskih resursa, poput količine radne i programske memorije, programiranje programske podrške ugradbenog računalnog sustava je otežano. Kako bi se iskoristio puni potencijal ugradbenog računalnog sustava potrebno je puno inženjerskog iskustva.

Ovaj rad orijentiran je na razvoj metode koja omogućuje parcijalno ažuriranje programske podrške za ugradbene računalne sustave unutar automobila. Programska podrška, unutar automobila, razvija se prema AUTOSAR (engl. *AUTomotive Open Systems ARchitecture*) [2] standardu, koji je od 2003. godine osnovni standard za razvoj programske podrške u automobilskoj industriji. AUTOSAR standard strogo je definiran što znači da nema veliku fleksibilnost te nije podložan izmjenama ukoliko je sustav ugrađen unutar automobila. Kako bi se ti nedostatci nadomjestili, potrebno je uvesti parcijalno ažuriranje programske podrške koje otvara mogućnost da se strogo definiranom konceptu, kao što je AUTOSAR standard, poveća fleksibilnost te olakša prilagodljivost programske podrške na izmjene u sustavima koji se nalazi u automobilu. Osim toga, parcijalno ažuriranje programske podrške omogućava uštedu vremena koje je neophodno za testiranje cjelokupne programske podrške, a samim time skraćuje se i vrijeme za isporuku krajnjim korisnicima. Osim uštede vremena, omogućava uštedu računalnih resursa i dovodi do smanjenja opterećenja samog računala i raspodijeljenih procesa. Dakle, cilj parcijalnog ažuriranja u automobilskoj industriji predstavlja mogućnost izmjene željene funkcionalnosti koja je vezana za komponentu programske podrške. Pored same izmjene funkcionalnosti, koncept ažuriranja

omogućio bi također i dodavanje/uklanjanje funkcionalnosti unutar sustava, koje su vezane za fizičke komponente (npr. senzori) unutar automobila.

Zadatak ovog rada je napraviti dvije verzije programske podrške. Prva verzija programske podrške je osnovna verzija koja uključuje predefiniranu funkcionalnost. Također, prva verzija koristi cjelokupnu memoriju prilikom implementacije koda na ploču. Druga verzija programske podrške sadržava samo izmijene vezane za istu funkcionalnost i koristi memoriju mikroupravljača u čijem opsegu su unesene promijene. Time bi se na praktičan način demonstrirala mogućnost parcijalnog ažuriranja programske podrške. U okviru ovog rada za parcijalno ažuriranje programske podrške koristila se *MotionWise* [3] platforma na razvojnoj ploči s TC397 *Infineon* mikroupravljačem [4]. Ploča se povezuje putem UART serijske magistrale s osobnim računalom, kako bi se lakše pratili rezultati. Pored ove platforme koristit će se *MiniWiggler* [5] alat, koji služi za učitavanje izvršne datoteke na platformu, uz podršku *MemTool* [6] programskog alata na osobnom računalu.

Nastavak diplomskog rada strukturiran je na sljedeći način: u drugom poglavlju dan je pregled postojećih rješenja parcijalnog ažuriranja programske podrške. U trećem poglavlju su predstavljene tehnologije koje su korištene prilikom implementacije parcijalnog ažuriranja programske podrške, kao i sam postupak izrade parcijalnog ažuriranja programske podrške. U četvrtom poglavlju napravljeno je testiranje programske podrške te validacija dobivenih rezultata. Na kraju rada dan je zaključak.

2. POSTOJEĆA RJEŠENJA PARCIJALNOG AŽURIRANJA PROGRAMSKE PODRŠKE

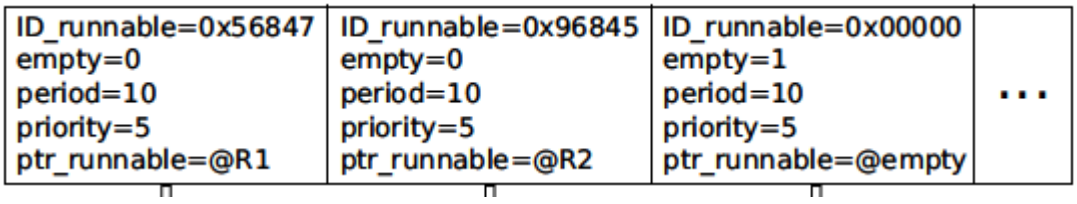
AUTOSAR je standard koji definira programsku arhitekturu za autonomne ugradbene sustave. Glavne prednosti AUTOSAR-a temelje se na: standardiziranim sučeljima, jednostavnom održavanju i integraciji komponenti programske podrške (engl. *software components*, SWC). Međutim, kao glavni nedostatak ovog standarda ističe se nedostatak fleksibilnosti. Kompletna konfiguracija sustava obično se izvršava prije postupka prevođenja (engl. *compile*), nakon čega nije moguća gotovo nikakva preinaka. Tradicionalan način ažuriranja ugradbene programske podrške predstavlja ponovnu implementaciju programske podrške na sve postojeće elektroničke upravljačke jedinice (engl. *Electronic Control Units*, ECUs). Suprotno tome, parcijalno ažuriranje programske podrške treba omogućiti postupak nadogradnje/izmjene dijela programske podrške, bez da se ažurira cijela programska podrška za elektroničku upravljačku jedinicu. Glavna ideja parcijalnog ažuriranja je modifikacija ili dodavanje malenih entiteta programske podrške unutar elektroničke upravljačke jedinice npr. modifikacija na postojećoj funkcionalnosti može dovesti do ažuriranja samo određene funkcije, a ne cijelog sustava.

U radu [7], kao rješenje predlaže se uvođenje spremnika (engl. *containers*) i pametnih pokazivača (engl. *smart pointers*). Spremnik predstavlja implementaciju memorijskog prostora unutar aplikacije. Spremnik je rezervirano memorijsko mjesto unutar kojeg se mogu dodavati izvršne funkcije (engl. *runnables*) koje će se kasnije ažurirati. Budući da su funkcije različitih veličina, potrebno je pripaziti na veličinu spremnika. Slika 2.1. prikazuje primjer programske podrške unutar kojeg je dodan spremnik, a unutar spremnika smještene su izvršne funkcije (iz različitih komponenti programske podrške) koje je potrebno ažurirati.

Izvršna_funkcija_1 (komponenta_programske _podrške_1)	Izvršna_funkcija_3 (komponenta_programske _podrške_1)	Izvršna_funkcija_5 (komponenta_programske _podrške_2)	Spremnik (slobodna mjesta)

Sl. 2.1. *Primjer programske podrške koja sadrži spremnik*

Osim spremnika, unutar programske podrške potrebno je dodati indirektnu tablicu (engl. *indirection table*). Slika 2.2. prikazuje sadržaj kojim je popunjena indirektna tablica.



Sl. 2.2. *Izgled indirektne tablice*

Indirektna tablica služi za pozivanje izvršnih funkcija i praznih spremnika. Tablica je popunjena pametnim pokazivačima koji preusmjeravaju do izvršnih funkcija koje se pozivaju i do spremnika. Dakle, pametni pokazivač sadrži referencu do izvršnih funkcija (ID izvršne funkcije) koje se pozivaju. Sljedeća stvar koja je vrlo važna za parcijalno ažuriranje je redoslijed izvršavanja zadataka. Kada se pokrene novo parcijalno ažuriranje, postupak ažuriranja razbija se na manje dijelove tj. zadatke. Unutar ovog rada, redoslijed izvršavanja definiran je pomoću prioriteta. Svaki zadatak ima definiran prioritet izvršavanja, na temelju prioriteta slaže se redoslijed. Važno je za napomenuti da zadatci moraju međusobno komunicirati i izmjenjivati podatke, kako bi redoslijed izvršavanja bio pravilno posložen. Postupak parcijalnog ažuriranja podijeljen je u tri koraka. Prvi korak je provjera kompatibilnosti spremnika s karakteristikama izvršnih funkcija, ukoliko su kompatibilni prelazi se na drugi korak. Drugi korak predstavlja kreiranje redoslijeda izvršavanja zadataka (kreira se matrica prioriteta). U zadnjem koraku provjerava se ima li ažuriranje negativan utjecaj na sigurnost ugradbenog sustava. Dodavanjem spremnika i pametnih pokazivača unutar elektroničkih upravljačkih jedinica, omogućen je postupak parcijalnog ažuriranja i nakon što je sustav implementiran u ugradbeni sklop. Ovakav postupak parcijalnog ažuriranja zaista je moguće primijeniti u realnom vremenu, no samo na klasičnom AUTOSAR standardu, a ne na adaptacijskom tj. onakvom kakav je on danas. Iako je ovakav postupak primjenjiv i dalje se pojavljuju određena ograničenja unutar AUTOSAR-a koja su vezana za autonomnu razinu ukupne sigurnosti (engl. *Automotive Safety Integrity Level, ASIL*). Ovakav tip ograničenja jedino je moguće riješiti kreiranjem nove arhitekture koja je bazirana na konvencionalnijem operacijskom sustavu.

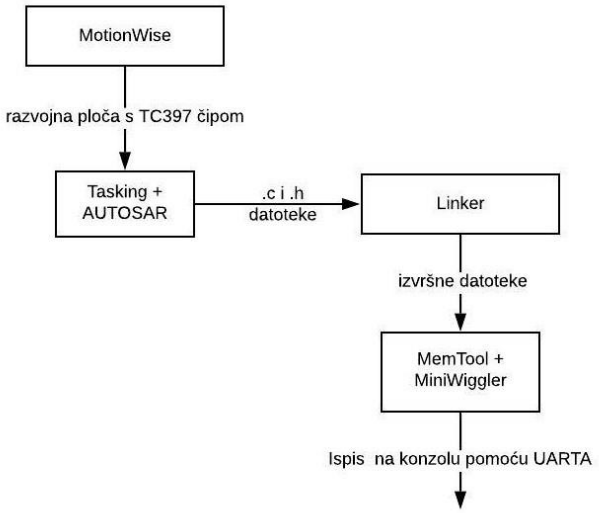
SystemDesk. [8] kao rješenje nudi alat koji omogućuje definiranje mogućih rekonfiguracija za zadani sustav. Ovaj alat zahtijeva model svih mogućih rekonfiguracija, a prednost mu je što se lako implementira u AUTOSAR sustav.

Rad [9] predstavlja dosta općenitiji pristup i predlaže alat koji je zapravo rezultat povezivanja alata koji su zasnovani na UML-u (engl. *Unified Modeling Language*). Ovaj alat dizajniran je za kritične sustave koji trebaju raditi u stvarnome vremenu. Ovaj pristup je prilično zanimljiv, ali nije dizajniran posebno za autonomne sustave, a ni za sam AUTOSAR.

Binarna ažuriranja još su jedan od primjera rješenja parcijalnog ažuriranja za ugradbene sustave jer omogućavaju izmjenu dijela memorije bez potrebe za ponovnim učitavanjem kompletne programske podrške. Binarna ažuriranja temelje se na sustavu koji konstruira stablo razlike između dvije verzije binarnog koda. Kao primjer implementacije takvog sustava imamo OTA (engl. *Over-TheAir*) ažuriranja [10]. Ukoliko su na nekoj programskoj podršci unesene promijene, sustav se automatski ažurira.

3. IZRADA RJEŠENJA ZA PARCIJALNO AŽURIRANJE PROGRAMSKE PODRŠKE

Slika 3.1. prikazuje blok dijagram sustava [11] koji je korišten prilikom razvoja metode koja omogućuje parcijalno ažuriranje programske podrške.



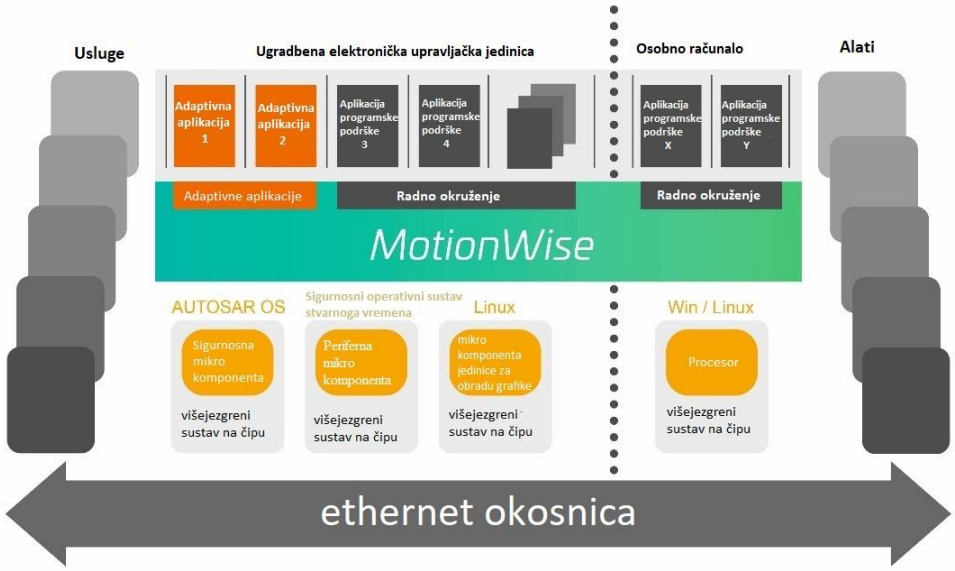
Sl. 3.1. *Blokovski prikaz korištenog sustava*

MotionWise je platforma koja sadrži sve programabilne komponente na razvojnoj ploči. Unutar ovog rada korištena je razvojna ploča s TC397 mikroupravljačem. Ploča je povezana s računalom pomoću univerzalnog asinkronog primopredajnika (engl. *Universal Asynchronous Receiver/Transmitter,* UART). Mikroupravljač se programira pomoću *Tasking*-a [12] unutar kojeg se koristi AUTOSAR standard, a koji određuje način na koji se programiraju komponente sustava. *Tasking* je korišten za stvaranje *.c* i *.h* datoteka, kojima se pridjeljuje određena funkcionalnost. Nakon što su stvorene sve potrebne datoteke, poziva se povezivač (engl. *linker*). Povezivač je komponenta koja je ugrađena unutar *Tasking*-a. Povezivač povezuje sve datoteke koje su nužne za proces stvaranja izvršnih datoteka. Nakon što su kreirane izvršne datoteke, pomoću *MemTool*-a odabiremo krajnji sklop (u ovom slučaju razvojna ploča), na koji se učitava izvršna datoteka. Kako bi učitavanje izvršne datoteke bilo uspješno, *MemTool* se koristi u kombinaciji s uređajem koji se

naziva *miniWiggler*. Postupak učitavanja izvršne datoteke na ploču i korištene tehnologije detaljnije su opisane u tekstu koji slijedi.

3.1. MotionWise platforma

Platforma koja je korištena u radu naziva se *MotionWise*. *MotionWise* je skalabilna platforma za razvoj programske podrške programske podrške za automatiziranu vožnju do razine autonomije 5. Na slici 3.2. [13] prikaza je struktura korištene platforme koja se sastoji od ugrađene programske podrške, prototipa sklopovlja i premium usluga koje se mogu pojedinačno kombinirati kako bi se platforma prilagodila specifičnim zahtjevima. Ovakva struktura platforme jamči jednostavan proces integracije i sigurnost programske podrške.



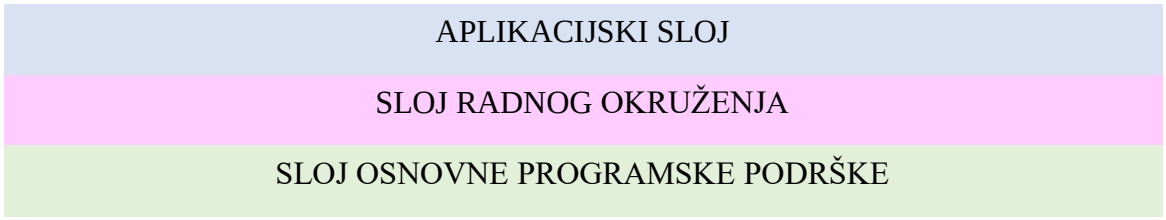
Sl. 3.2. *Prikaz Strukture MotionWise platforme*

Navedena platforma predstavlja skalabilno i visoko učinkovito rješenje za razvoj programske podrške koja ubrzava razvoj projekata koji su vezani uz autonomnu vožnju. *MotionWise* platforma nudi sveobuhvatan skup sistemskih i integracijskih usluga, pružajući homogenu platformu izvan sustava na mikroupravljaču (engl. *System on a Chip*, SoC).

3.2. AUTOSAR Standard

Standard koji je korišten za razvoj ovog rada je AUTOSAR. AUTOSAR predstavlja standard za razvoj programske podrške za ugradbene računalne sustave u automobilskoj industriji. AUTOSAR je nastao s ciljem uspostavljanja otvorene i standardizirane programske arhitekture za programiranje elektroničke upravljačke jedinice u automobilima.

AUTOSAR standard predstavlja se pomoću slojevite arhitekture koja se naziva komunikacijski stog (engl. *Communication Stack, ComStack*). Izgled stoga prikazan je na slici 3.3.



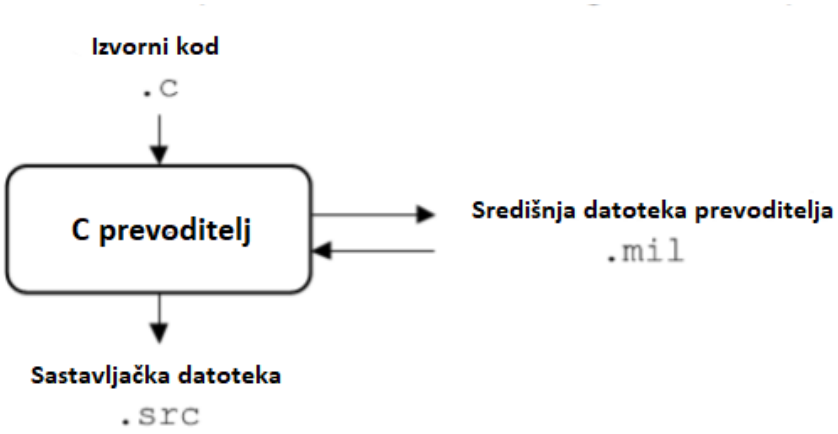
Sl. 3.3. *AUTOSAR komunikacijski stog*

Komunikacijski stog sastoji se od: aplikacijskog sloja (engl. *Application Layer*, AL), sloja okruženja izvođenja (engl. *RunTime Environment*, RTE) i sloja osnovne programske podrške (engl. *basic software*, BSW). Aplikacijski sloj sastoji se od komponenti programske podrške (engl. *software components*) koje predstavljaju algoritme ili aplikativne komponente koje se nalaze na samom automobilu. U okviru sloja osnovne programske podrške nalaze se izvršne funkcije (engl. *runnables*) koje se pokreću ciklički ili kao rezultat određene pobude (ulazni podatak za izvršnu funkciju).

3.3. Programska podrška za izradu parcijalnog ažuriranja

Tasking v6.2r2.

Kako bi postupak izgradnje izvršne datoteke bio potpun potrebno je prevesti izvorni kod. Prevoditelj (engl. *compiler*) je program koji kao ulaz prima kod koji je napisan u izvornom jeziku (u ovom slučaju C), te ga prevodi i kao izlaz daje *.src* datoteku (datoteka koja sadrži izvorni kod, a služi za kreiranje izvršne datoteke), što je i prikazano slikom 3.4. [14]



Sl. 3.4. *Ulaz i izlaz prevoditelja*

Prevoditelj koji se koristi u ovom radu naziva se *Tasking v6.2r2*. Prilikom izrade postupka parcijalnog ažuriranja, *Tasking* je korišten i kao sučelje za pisanje aplikativnog koda na *MotionWise* platformi.

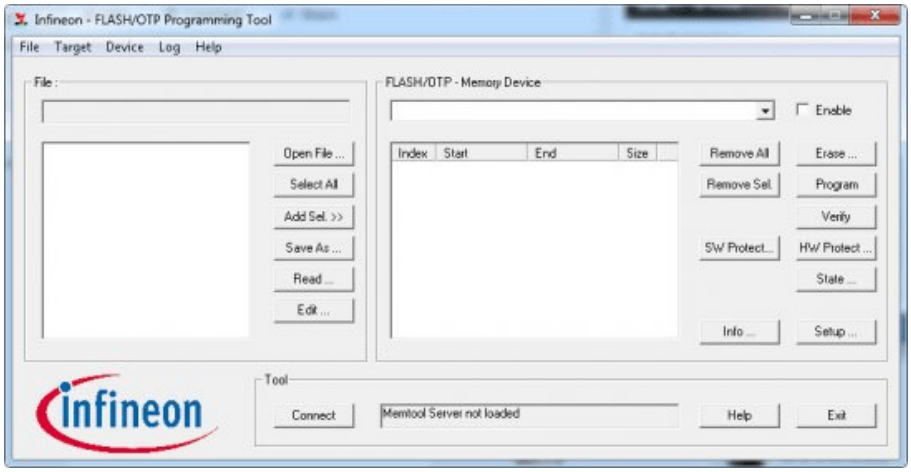
Infineon Memtool

MemTool je besplatna programska podrška koja je dizajnirana za programiranje mikroupravljačkih sklopovskih sustava kao što su: *AURIX*, *TriCore*, Cortex, *ARM*, *C166/ST10*, *XE166/XC2000*. Unutar tablice 3.1. navedene su samo neke od mogućnosti koje pruža *MemTool* alat.

Tablica 3.1. *Mogućnosti MemTool alata*

1.	Brisanje cijelog memorijskog modula
2.	Brisanje odabranih sekcija memorijskog modula
3.	Učitavanje Intel HEX datoteka
4.	Programiranje svih ili odabranih dijelova datoteke u memorijski modul
5.	Usporedba svih ili odabranih dijelova datoteke s trenutnim sadržajem memorijskog modula
6.	Postavljanje i resetiranje zaštite čipa / sektora (samo on-chip)
7.	Konfiguracija indeks pokretanja (engl. *Boot Mode Index*, BMI)
8.	Podržava univerzalni asinkroni primopredajnik

MemTool sučelje prikazano je na slici 3.5. [15], a ono omogućava povezivanje programske podrške s ciljanim sklopovljem tj. pločom.



Sl. 3.5. *Prikaz MemTool sučelja*

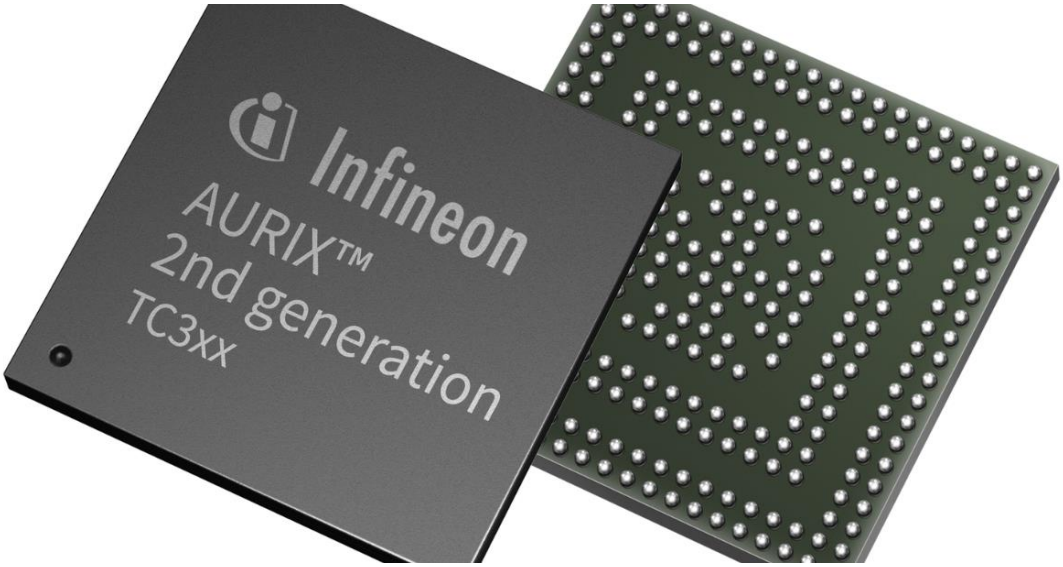
Unutar tablice 3.2. opisane su funkcionalnosti komponenti koje se nalaze unutar *MemTool* sučelja.

Tablica 3.2. *Funkcionalnosti komponenti*

Gumb „Otvori datoteku“ (engl. *Open File*)	Omogućava učitavanje HEX datoteke u MemTool, kasnije će ista datoteka biti učitana na krajnji uređaj (sklop)
Gumb „Spoji se s krajnjim uređajem“ (engl. *Connect target*)	Omogućava povezivanje programske podrške, tj. računala sa krajnjim uređajem
Dijalog „Memorija uređaja“ (engl. *Memory device*)	Pruža informacije o konfiguraciji memorije krajnjeg uređaja
Gumb „Odaberi sve..“ (engl. Select All...)	Omogućava odabir učitavanja podataka na cjelokupnu memoriju krajnjeg uređaja
Gumb „Dodaj odabrano..“ (engl. Add Sel...)	Pritiskom na ovaj gumb potvrđuju se memorijske adrese koje će se koristiti prilikom pokretanja izvršne datoteke

3.4. Infineon TC397 sklopovska platforma

U diplomskom radu se kao sklopovska platforma koristila razvojna ploča s TC397 *Infineon* mikroupravljačem. Izgled mikroupravljača, koji je programiran prilikom izrade parcijalnog ažuriranja, prikazan je slici 3.6. [16]



Sl. 3.6. *Infineon AURIX mikroupravljač*

Značenje oznake TC397 *Infineon* mikroupravljača definirano je u tablici 3.3.

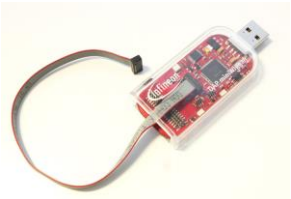
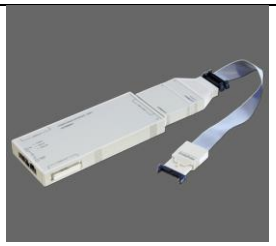
Tablica 3.3. *Oznake koje su korištene prilikom imenovanja mikroupravljača*

TC	Predstavlja oznaku za vrstu mikroupravljača	Mikroupravljač s trojezgrenim procesorom (engl. *TriCore*, TC)
3	Ove tri oznake prikazuju informacije o performansama uređaja	Arhitektura uređaja
9		Predstavlja verziju uređaja
7		PIN-a koji se koristi na razvojnoj ploči za pristup mikroupravljaču

Mikroupravljač je dizajniran da udovolji visokim i zahtjevnim potrebama automobilske industrije u pogledu performansi i sigurnosti, a rad zasniva na tri neovisna trojezgrena procesora. Osim što je jednostavan za korištenje, mikroupravljač omogućava visok stupanj kompatibilnosti i skalabilnosti.

Razvojna ploča pruža mogućnost spajanja s velikim brojem različitih perifernih uređaja. Prilikom implementacije postupka parcijalnog ažuriranja programske podrške, na razvojnu ploču spojen je uređaj koji se naziva *miniWiggler* [17]. *MiniWiggler* se koristi u kombinaciji s *MemTool*-om, kako bi se izvršilo učitavanje izvršne datoteke na krajnji uređaj. Osim što omogućava povezivanje programske podrške sa sklopovljem, *miniWiggler* pruža mogućnost detekcije i otklanjanja pogrešaka (engl. *debugging*). *Lauterbach* [18] je uređaj koji ima istu funkcionalnost kao i *miniWiggler*, a razlike među spomenutim uređajima navede su u tablici 3.4.

Tablica 3.4. *Usporedba uređaja za detekciju pogrešaka*

	miniWiggler	*Lauterbach*
Primjena	Oba uređaja koriste se za implementaciju izvršne datoteke na ploču, kao i za sam proces detekcije/otklanjanja pogrešaka	
Cijena	100 €	3000 €
Opseg korištenja	Samo za Infineon Chipove	Za gotovo sve ARM Chipove
Radno okruženje	DAS	TRACE32
Izgled uređaja		

Tijekom izrade diplomskog rada korišten je *miniWiggler,* jer je dostupniji i cijena uređaja je manja.

3.5. Predloženo rješenje za parcijalno ažuriranje programske podrške

Zadatak ovog rada je na praktičan način demonstrirati mogućnost parcijalnog ažuriranja programske podrške. Parcijalno ažuriranje realizirano je pomoću dvije verzije programske podrške. Prva verzija programske podrške je osnovna, unutar koje su implementirane dvije funkcije: *FunA* i *FunB*. Druga verzija programske podrške sadrži samo ključne izmjene funkcionalnosti koje su usko vezane za *FunB*, dok je preostali dio *flash* memorije (gdje je pohranjen programski kod) mikroupravljača ostao nepromijenjen. Unutar tablice 3.5. navedeni su ključni koraci postupka izrade parcijalnog ažuriranja. Svaki korak detaljno je objašnjen u tekstu koji slijedi nakon tablice.

Tablica 3.5. *Ključni koraci izrade parcijalnog ažuriranja*

1.	Dodavanje praznih datoteka	Prva verzija
2.	Povezivanje datoteka	
3.	Pokretanje inicijalne izgradnje izvršne datoteke (engl. *build)* i prvo učitavanje (engl. *flash)* na ploču	
4.	Dodjeljivanje funkcionalnosti novokreiranim datotekama	
5.	Ispis poruke svake dvije sekunde putem UART-a	
6.	Unošenje promjena u odnosnu na osnovnu verziju	Druga verzija
7.	Kreiranje probne particije	
8.	Popunjavanje probne particije i njeno testiranje	
9.	Kreiranje parcijalne povezivač i *.cmd* skripte	

Početni korak, prilikom izrade rješenja za parcijalno ažuriranje programske podrške je kreiranje datoteka. Kreirane su 4 datoteke: *FunctionA.c* i *FunctionB.c*, *FunctionA.h* i *FunctionB.h*. Kreirane datoteke su u početku prazne te će im naknadno biti dodijeljena funkcionalnost. Izabran je ovakav postupak, jer se želilo osigurati da su datoteke dodane na pravilnom mjestu i provjeriti jesu li dobro definirani podatci za implementaciju novokreiranih datoteka. Ukoliko datoteke nisu pravilno implementirane, postupak izgradnje izvršne datoteke bit će neuspješan. Dakle, sljedeći

korak u rješavanju problema je pravilno povezivanje *.c* i *.h* datoteka, na način da se u *.ini* datoteci kreiraju varijable koje predstavljaju put do novokreiranih datoteka. Povezivanje datoteka izvršava se unutar *.ini* datoteke koja sadrži popis svih putanja do ključnih direktorija i komponenti. Ukoliko je putanja točno definirana postupak izgradnje izvršne datoteke će se uspješno izvršiti, u suprotnom postupak izgradnje izvršne datoteke pada, jer komponente sustava ne mogu pronaći potrebne datoteke unutar sustava. U tekstu koji slijedi, detaljno je opisan postupak pokretanja inicijalne izgradnje izvršne datoteke i postupak učitavanja koda na ploču.

Proces izgradnje izvršne datoteke započinje pozicioniranjem u direktorij unutar kojeg se nalazi datoteka koja definira sam postupak izgradnje. Nakon pravilnog pozicioniranja, otvara se *Command Prompt* unutar kojeg se pokreće sljedeća naredba: *build.cmd System_SH00 -j8*. Unutar tablice 3.6. objašnjeni su ključni pojmovi naredbe koja pokreće postupak izgradnje izvršne datoteke.

Tablica 3.6. *Ključni pojmovi naredbe koja pokreće proces izgradnje izvršne datoteke*

<table>
<tr><th></th><th>Ključna riječ</th><th colspan="2">Opis funkcionalnosti</th></tr>
<tr><td>1.</td><td>build.cmd</td><td colspan="2">Skripta koja definira pozive funkcija/komponenti koje su potrebne za kreiranje izvršne datoteke</td></tr>
<tr><td rowspan="2">2.</td><td rowspan="2">System_SH00</td><td>System</td><td>Vrsta izgradnje izvršne datoteke</td></tr>
<tr><td>SH00</td><td>safety host 00</td></tr>
<tr><td>3.</td><td>-j8</td><td colspan="2">Oznaka kojom se uključuje rad 8 jezgara računala kako bi se skratilo vrijeme trajanja postupka izgradnje izvršne datoteke</td></tr>
</table>

Prilikom pokretanja *build.cmd* skripte, kontrolni program poziva: *C* prevoditelj, sastavljač (engl. *assembler)* i povezivač (unutar tablice 3.7. detaljno su objašnjene funkcionalnosti navedenih komponenti sustava).

Tablica 3.7. *Funkcionalnosti komponenti koje se pozivaju prilikom izgradnje izvršne datoteke*

<table>
<tr><th></th><th>Naziv komponente</th><th colspan="2">Opis funkcionalnosti</th></tr>
<tr><td>1.</td><td>C prevoditelj</td><td colspan="2">Za ulaz uzima .c datoteke, koje obrađuje i stvara .src datoteke (datoteka koja sadrži izvorni kod, služi za kreiranje izvršne datoteke)</td></tr>
<tr><td>2.</td><td>Sastavljač</td><td colspan="2">Sastavljač pretvara kodove/programe (ručno napisane ili generirane od strane prevoditelja) u programe koji su prilagođeni strojnom jeziku. Stvara datoteku ELF formata.</td></tr>
<tr><td rowspan="3">3.</td><td rowspan="3">Povezivač</td><td colspan="2">Povezivač je komponenta koja izvršava proces povezivanja (engl. linking). Proces povezivanja sastoji se od dvije faze: povezivanje (engl. link) i lociranje (engl. locate). Povezivač omogućava objektnoj datoteci učitavanje (zauzimanje memorije) na krajnji uređaj (engl. target).</td></tr>
<tr><td>povezivanje</td><td>Povezivač povezuje isporučene datoteke i biblioteke u jednu prenosivu objektnu datoteku (.o)</td></tr>
<tr><td>lociranje</td><td>U fazi lociranja, povezivač dodjeljuje apsolutne adrese objektnom kodu, smještajući svaki odjeljak (memorije) unutar određenog dijela fizičke memorije na ploči</td></tr>
</table>

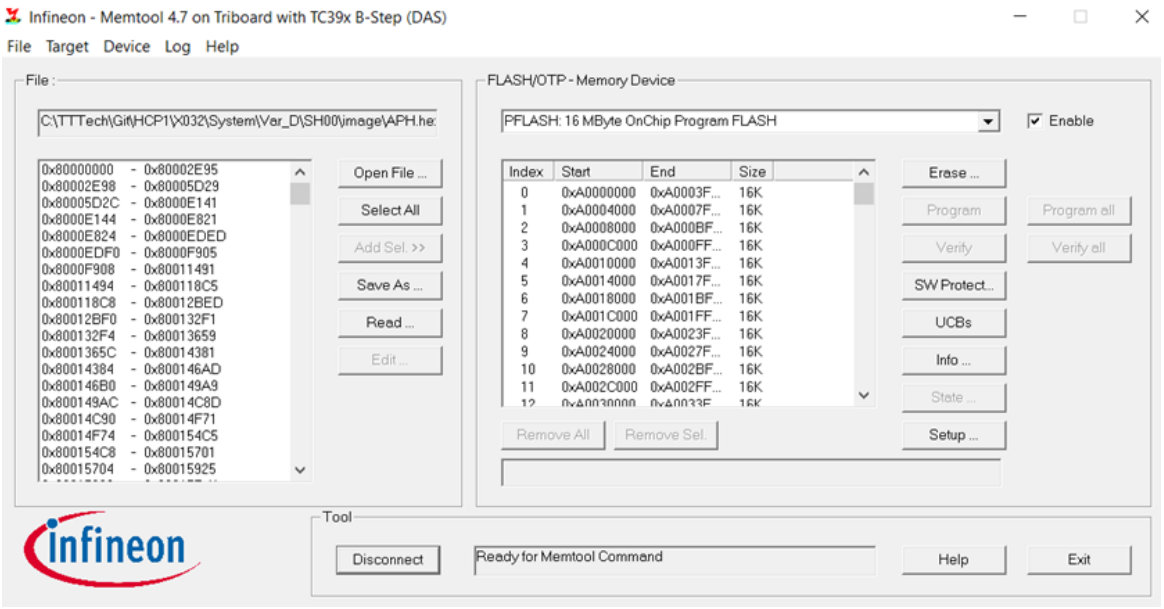
Ukoliko je postupak izgradnje izvršne datoteke uspješno izvršen (*build_status SUCCESS*, slika 3.7.) stvara se datoteka HEX formata (kreirane su i druge datoteke, ali ova je važna za daljnji tekst) koja sadrži sliku (engl. *image*) izvršnog koda i svih ključnih podataka. Iz slike 3.7. vidljivo je da je trajanje postupka inicijalne izgradnje izvršne datoteke otprilike šezdesetak minuta, a razlog tome jest što se prilikom prvog prevođenja poziva jako velika količina biblioteka i komponenti koje se trebaju povezati.

```
ltc I401: activating locate phase
ltc I419: applying LSL file (all tasks)
ltc I421: preparing copy table (task1)
ltc I424: optimizing copy table (task1)
ltc I427: locating items (all tasks)
ltc I428: binding locator symbols (task1)
ltc I429: evaluating expressions (task1)
ltc I430: patching relocation information (task1)
ltc I431: running post locate actions (task1)
ltc I432: finalize locating (task1)
ltc I402: activating file producing phase
ltc I433: emitting object files (task1)
ltc I434: emitting report files (task1)
============================================================
build_status                 SUCCESS
scons: done building targets.

--------------------------------------------------------------------------------
Build started : /20/- 0-Mo  8:17:29
Build finished: /20/- 0-Mo  9:14:35
--------------------------------------------------------------------------------
```

Sl. 3.7. *Poruka kojom se potvrđuje uspješnost inicijalne izgradnje izvršne datoteke*

Nakon stvaranja izvršne datoteke slijedi njeno učitavanje na ploču. Slika 3.8. prikazuje sučelje *MemTool*-a tj. programa pomoću kojeg se slika učitava na ploču, a u daljnjem tekstu navedeni su koraci koje je potrebno obaviti kako bi izvršna datoteka bila uspješno učitana.



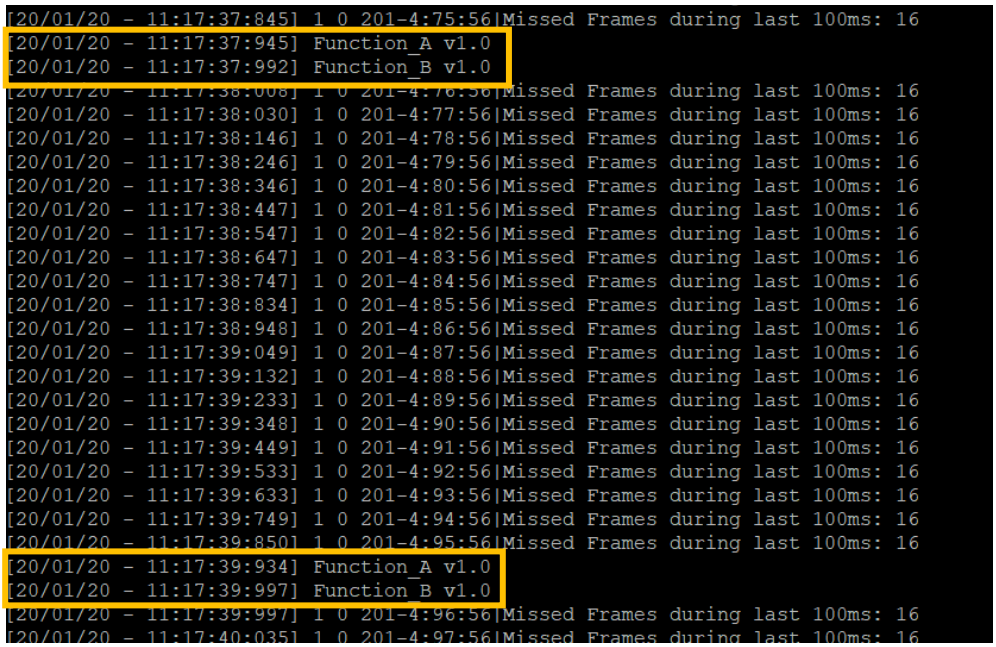
Sl. 3.8. *Prikaz programskog sučelja MemTool-a*

Redoslijed koraka za učitavanje izvršne datoteke na ploču:

1. Učitavanje HEX datoteke unutar polja „*File*“
2. Klikom na „*Select All*“ odabran je raspon adresa za postupak učitavanja
3. Klikom na „*Add Sel.* >>“ ploča je spremna za postupak učitavanja
4. Klikom na „*Program all*“ pokrenut je postupak učitavanja izvršne datoteke

Nakon prvog pokretanja izvršne datoteke slijedi popunjavanje praznih datoteka tj. dodjeljivanje određene funkcionalnosti datotekama koje su kreirane u prvom koraku. Unutar praznih *.h* datoteka implementirane su potrebne biblioteke i napisana je definicija funkcije za ispis. Unutar praznih *.c* datoteka (*FunctionA.c* i *FunctionB.c*) pozvane su neophodne *.h* datoteke, također pozvana je „zapakirana“ funkcija (engl. *wrapper function*) za ispis na serijskom sučelju, funkciji je potrebno predati neophodne parametre. Ukoliko je funkcija pravilno pozvana i ukoliko su joj proslijeđeni validni parametri, funkcija za ispis će pomoću modula za serijsku komunikaciju na terminal ispisati poruku (brzina serijske komunikacije postavljena je na 115200 bps). Funkcija *FunA* će ispisati „Function_A v1.0“, dok će *FunB* ispisati „Function_B v1.0“, na slici 3.9. prikazana je poruka koja se ispisuje. Nakon dodavanja koda izvornim datotekama ponovno je pokrenut postupak izgradnje izvršne datoteke, kako bi se provjerila uspješnost rada novokreiranih funkcija. Ponovno pokretanje postupka izgradnje izvršne datoteke nešto je kraće i traje otprilike 45 minuta (biblioteke su ranije implementirane i komponente su već povezane s izvornim kodom). Rezultat pokretanja izvršne datoteke na ploči je ispis ranije definiranih poruka svakih 10 ms, uzrok tome je što se funkciju za ispis poziva unutar komponente koja ima predefinirano vrijeme izvršavanja svakih 10 ms. Sljedeći problem je podesiti poziv funkcije za ispis svake dvije sekunde. Kako bi se riješio navedeni problem, unutar koda uvodi se varijabla koja predstavlja brojač. Vrijednost brojača će se povećati za jedan, svaki puta prilikom izvršavanja cikličke funkcije, tj. svakih 10 ms. Dakle, kada se funkcija izvrši 200 puta, proteklo vrijeme iznosi dvije sekunde tj. 2000 ms. Kada brojač dosegne vrijednost koja je jednaka 200, tada se poziva funkcija za ispis, a vrijednost brojača izjednačava se s nulom. *ExtraPutty* je korisnički program koji pruža mogućnost ispisa vremenske oznake (engl. *time stamp*), na konzoli, u formatu [%d/%m/%y - %H:/%M:/%S:_MIL]. Na slici 3.9. vidljivo je da se na ovaj način može provjeriti poziva li se funkcija za ispis zaista svake dvije sekunde. Dovršetkom

ovog koraka kreirana je osnovna (prva) verzija programske podrške koja sadržava implementaciju funkcija *FunA* i *FunB*.

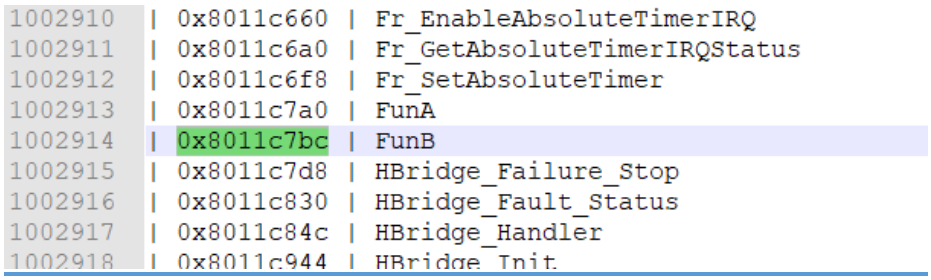


Sl. 3.9. *Ispis poruke na UART pomoću klijentskog programa ExtraPutty*

U kod je unesena izmjena funkcije *FunB* (funkcija koja se ažurira) koja se očituje u izmjeni njezine verzije, dakle verzija početne funkcije *FunB* bila je 1.0, a nakon promjene njezina verzija je 1.1 (verzija je jedan od podataka koji se šalje putem serijske sabirnice). Funkcija *FunA* ostaje nepromijenjena u toku cijelog procesa ažuriranja, dakle njezina verzija je 1.0.. Promjena koja je unesena u kod nije velika, ali je dovoljna za ponovno pokretanje izgradnje izvršne datoteke (kako bi funkcija mogla ispisati novu poruku tj. verziju). Unošenjem promjena u kod, ulazi se u proces izgradnje druge verzije programske podrške. Druga verzija treba omogućiti izgradnju izvršne datoteke koja će u obzir uzimati samo dijelove koda koji se razlikuju u odnosu na osnovnu verziju (dakle, samo ono što je promijenjeno) i komponente sustava u kojima se nalazi izmijenjen kod. Skripta koja će definirati proces izrade izvršne datoteke, manipulirati će samo odabranim dijelovima koda i komponentama (na kojima je izvršena promjena) te će dovesti do smanjivanja vremena koje je potrebno za izgradnju izvršne datoteke.

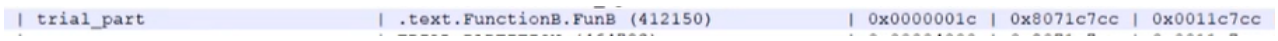
U nastojanju da se skrati vrijeme kreiranja izvršne datoteke, koristit će se samo jedan dio memorije, a ne sva memoriju koja je dostupna. Korištenje samo dijela memorije moguće je postići kreiranjem particije. Particija predstavlja određeni „komad“ memorije čija se veličina alocira na

način da se definira raspon memorijskih adresa. Kako bi particija bila definira na traženom rasponu adresa, potrebno je pronaći na kojoj su adresi upisane funkcije *FunA* i *FunB*. Informaciju o memorijskim adresama povlačimo iz *.map* datoteke, koja se kreira nakon pokretanja izvršne datoteke što je i prikazano na slici 3.10. *Map* datoteka omogućava prikaz raspodijeljene memorije, veličinu memorije koju zauzimaju funkcije, broj particija koje su kreirane, raspon adresa na kojoj je particija kreirana te prikaz samog sadržaja particije.



Sl. 3.10. *Prikaz adresa na kojima se nalaze novokreirane funkcije*

Detekcijom adresa na kojima su upisane ključne funkcije, prikupljeni su svi potrebni podatci za kreiranje particije. Ovim podatcima dobivamo informaciju o količini memorije koju zauzimaju funkcije. Kako bi se funkcija *FunB* uspješno smjestila u rezerviranu memoriju, potrebno je rezervirati dovoljno memorije. U konkretnom slučaju to je 16 kB programske memorije. Kreirana je nova particija pod nazivom „*trial_part*“, a njena veličina postavljenja je na 16 kB. Nakon prebacivanja funkcije *FunB* u rezerviranu memoriju, memorijski prostor koji je prethodno zauzimala funkcija *FunB* se oslobađa i ostaje prazan. U dio memorije koji je rezerviran za probnu particiju smještena je samo funkcija *FunB*, jer se u obzir želi uzeti samo onaj komad memorije na kojem su unesene promijene u kodu. Nakon ponovnog kreiranja izvršne datoteke, kao što je



Sl. 3.11. *Prikaz novokreirane particije u .map datoteci*

prikazano na slici 3.11., unutar *.map* datoteke pojavljuje se novokreirana particija s osnovnim informacija.

Sve daljnje operacije koje su potrebne za izgradnju izvršne datoteke, koristit će samo zadani raspon tj. rezerviranu memoriju (ukoliko kontrolni program troši manje memorije, brže će se kreirati izvršna datoteka i smanjiti ukupno vrijeme procesa). Nakon što je definirana probna particija, potrebno je napraviti prilagođene skripte, koje će za proces kreiranja izvršne datoteke uzimati samo one dijelove koji su izmijenjeni u odnosu na prošlu verziju.

Za razumijevanje konačnog rješenja potrebno je razjasniti koja je uloga povezivač skripti. Povezivač skripte, zadužene su za opisivanje načina na koji se odjeljci memorije u ulaznim datotekama trebaju preslikati u izlaznu datoteku te manipuliraju memorijom izlazne datoteke. Dakle, predefinirana povezivač skripta nosi informacije o cjelokupnoj memoriji, uzima podatke svih mogućih komponenti, povezuje ih, alocira memoriju i predaje izlazne podatke izvršnoj datoteci.

Riješene za parcijalno ažuriranje je sljedeće: kreirana je nova povezivač skripta koja je ograničena na povezivanje samo onih komponenti na kojima je izrađena probna particija, uzima u obzir samo onaj dio memorije na kojem se nalazi funkcija *FunB*. Unutra novokreirane povezivač skripte potrebno je definirati parametre koji su navedeni u tablici 3.8. [19]

Tablica 3.8. Komponente koje su nužne za kreiranje povezivač skripte

1.	***Architecture***	Definiranje načina na koji povezivač treba pretvoriti logičke adrese u fizičke adrese za ciljnu vrstu jezgre
2.	***Derivative***	Definira konfiguraciju unutarnje sabirnice i memorijskog sustava
3.	***Processor***	Ukoliko se radi o ugradbenom sustav s više procesora, potrebno je definirati koji će procesori biti korišteni
4.	***Memory and bus***	Definiranje vanjske memorije i sabirnica koje će biti korištene
5.	***Board***	Definiranje specifikacija ploče koja će biti korištena
6.	***Section layout***	Komponenta omogućava precizno upravljanje ulaznim odjeljcima: moguće je definirati mjesta gdje će se smještati novi ulazi (odjeljci)

Kreiranjem nove povezivač skripte prikupljene su sve datoteke koje su neophodne za kreiranje parcijalnog ažuriranja. Kako bi postupak parcijalnog ažuriranja bio uspješno izvršen potrebno je izmanipulirati/kreirati novu *build.cmd* skriptu, jer ona poziva sve datoteke koje su nužne

za izgradnju izvršne datoteke tj. parcijalnog ažuriranja. Predefinirana *build.cmd* skripta, je skripta koja sadrži popis svih komponenti koje su uključene u postupak kreiranja izvršne datoteke. Skripta se sastoji od niza varijabli koje predstavljaju putanje do ključnih komponenti. Kreirana je nova *build.cmd* skripte koja će implementirati samo one komponente/ dijelove koji su ključni za parcijalno ažuriranje projekta. Suvišne komponente tj. komponente koje nisu potrebne, izbacit će se iz skripte na način da se ukloni njihova *path* varijabla iz skripte. Ovakvim postupkom stvorili smo prilagođenu *build.cmd* skriptu. Važno za napomenuti jest da nova *build* skripta ne poziva predefiniranu povezivač skriptu već novokreiranu povezivač skriptu. Konačno, završetkom ove faze, kreiran je demonstrativni postupak parcijalnog ažuriranja.

4. TESTIRANJE RJEŠENJA ZA PARCIJALNO AŽURIRANJE PROGRAMSKE PODRŠKE

Da bi se analizirala svojstva napravljenog rješenja za parcijalno ažuriranje programske podrške u sklopu rada napravljena su sljedeća testiranja: testiranje vremena koje je potrebno za izgradnju izvršne datoteke, testiranje vremena koje je potrebno za učitavanje izvršne datoteke na ploču, te testiranje vremena koje je potrebno kako bi se detektirala pogreška.

4.1. Testiranje vremena izgradnje izvršne datoteke

Testiranje parcijalnog ažuriranja programske podrške temelji se na vremenu koje je potrebno za izgradnju izvršne datoteke. Unutar tablice 4.1., prikazana je usporedba dvaju vremena (vrijeme normalne izgradnje i vrijeme izgradnje pomoću parcijalnog ažuriranja), normalno vrijeme izgradnje izvršne datoteke je 57 minuta i 6 sekundi, dok je vrijeme izgradnje izvršne datoteke s parcijalnim ažuriranjem jednako 9 minuta i 45 sekundi. Normalna izgradnja izvršne datoteke je skoro 6 puta duža od one parcijalne, jer računalo prilikom normalne izgradnje izvršne datoteke koristi jako

Tablica 4.1. *Vrijeme izgradnje izvršne datoteke*

	Normalna izgradnja izvršne datoteke	Izgradnje izvršne datoteke pomoću parcijalnog ažuriranja
Vrijeme početka izgradnje izvršne datoteke	8:17:29	11:01:21
Vrijeme završetka izgradnje izvršne datoteke	9:14:35	11:11:06
Trajanje	57 minuta i 6 sekundi	9 minuta i 45

veliku količinu računalnih resursa tj. mora obraditi i implementirati veliki broj komponenti.

No, s druge strane parcijalna izgradnja izvršne datoteke definirana je na način da su implementirane samo one komponente na kojima se izvršavala promjena u odnosu na prošlu verziju - na ovaj način računalo štedi veliku količinu vremena i računalnih resursa. Parcijalno ažuriranje kao glavni zadatak ima smanjenje vremena i računalnih resursa koji su potrebni za izgradnju izvršne datoteke.

4.2. Testiranje vremena za implementaciju izvršne datoteke

Sljedeća faza testiranja projekta je određivanje prosječnog vremena koje je potrebno kako bi se izvršna datoteka implementirala na ploču. Uređaj koji se koristi prilikom implementacije je *miniWiggler*, a osim same implementacije njegova zadaća je i otkriti pogreške u kodu, ukoliko one postoje. Dakle, testiranje je realizirano pomoću dvadeset mjerenja (pokusa), za svako mjerenje u tablici 4.2. zapisano je vrijeme koje je bilo potrebno za implementaciju izvršne datoteke na ploču.

Analizom podataka iz tablice 4.2. dobiveno je prosječno (srednje, očekivano) vrijeme koje je potrebno za implementaciju izvršne datoteke na ploču i ono iznosi 99,75 sekundi. Dakle, srednje vrijeme je konstanta kojom se predstavlja niz varijabilnih podataka (2. stupac tablice 4.2.). Također, u tablici 4.2. nalaze se informacije o podatcima koji su korišteni za proračun normalne distribucije. Standardnu devijaciju σ moguće je dobiti pomoću izraza:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{4-1}$$

gdje je n ukupan broj mjerenja, $\bar{x}$ predstavlja srednja vrijednost tj. očekivano vrijeme implementacije koda na ploču, dok x_i predstavlja stvarno vrijeme koje je izmjereno za vrijeme testiranja implementacije. Primjenom formule (4-1) dobivena je standardna devijacija vremena implementacije koda na ploču σ_{imp} koja iznosi 2,268201 sekundi. Standardna devijacija predstavlja statistički pojam koji označava mjeru raspršenosti podataka u skupu. Interpretira se kao prosječno odstupanje od prosjeka u apsolutnom iznosu. Dakle, prosječno odstupanje vremena od prosječnog vremena koje je potrebno za implementaciju koda na ploču iznosi 2,268201 sekundi.

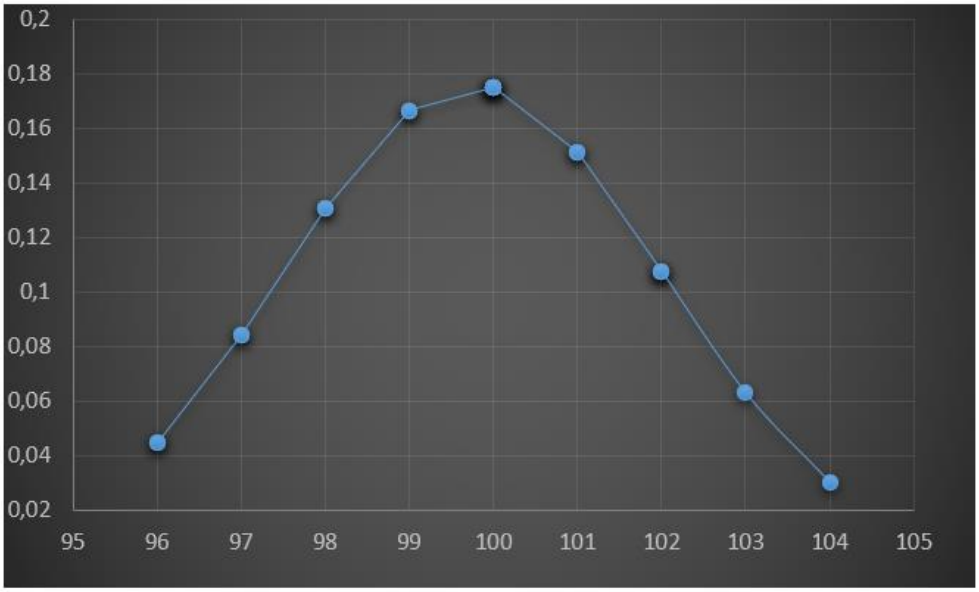
Tablica 4.2. *Vrijeme koje je potrebno za implementaciju koda na ploču*

Oznaka mjerenja:	Vrijeme implementacije:	Vrijeme implementacije u sekundama [s]	Normalna raspodjela
x_1	00:01:40	100	0,17482
x_2	00:01:38	98	0,130607
x_3	00:01:40	100	0,17482
x_4	00:01:48	101	0,151105
x_5	00:01:36	96	0,044842
x_6	00:01:39	99	0,166528
x_7	00:01:41	101	0,151105
x_8	00:01:37	97	0,08434
x_9	00:01:42	102	0,107536
x_{10}	00:01:40	100	0,17482
x_{11}	00:01:44	104	0,030399
x_{12}	00:01:42	102	0,107536
x_{13}	00:01:38	98	0,130607
x_{14}	00:01:40	100	0,17482
x_{15}	00:01:36	96	0,044842
x_{16}	00:01:39	99	0,166528
x_{17}	00:01:43	103	0,06301
x_{18}	00:01:40	100	0,17482
x_{19}	00:01:37	97	0,08434
x_{20}	00:01:42	102	0,107536
Podatci koji su korišteni za proračun normalne distribucije:			
Broj mjerenja (n)	20	Srednje (očekivano) vrijeme ($\bar{x}$)	99,75 s
Najveće vrijeme	104 s		
Najmanje vrijeme	96 s	Varijanca (σ^2)	5,144736
Min/Max devijacija	8 s	Standardna devijacija (σ)	2,268201

Nadalje, normalnu distribuciju moguće je izračunati pomoću izraza:

$$f(x, \bar{x}, \sigma) = \frac{1}{\sigma \cdot \sqrt{2\pi}} e^{-\frac{(x-\bar{x})^2}{2 \cdot \sigma^2}} \tag{4-2}$$

gdje σ predstavlja standardna devijaciju, σ^2 predstavlja varijancu, $\bar{x}$ stoji za srednju vrijednost tj. očekivano vrijeme implementacije koda na ploču, dok x_i predstavlja stvarno vrijeme koje je izmjereno za vrijeme testiranja implementacije. Dakle, četvrti stupac tablice 4.2. predstavlja vrijednosti normalne raspodijele koje su dobivene primjenom formule (4-2). Na temelju rezultata normalne raspodijele kreiran je graf [20] koji je prikazan na slici 4.1. Graf normalne raspodijele još se naziva i Gaussova krivulja tj. krivulja vjerojatnosti [21].



Sl. 4.1. *Graf normalne raspodjele za implementaciju izvršne datoteke*

Gaussova krivulja je unimodalna (ima jedan vrh ili tjeme), ima oblik zvona i proteže se od vrijednosti 96 do vrijednosti 104. Krivulja je simetričnog oblika. Specifičnost krivulje je ta što se 50 % podataka za koje se krivulja crta nalazi na jednoj strani krivulje, dok je preostalih 50 % podataka na drugoj strani. Iz krivulje se može očitati da je najčešće vrijeme implementacije izvršne datoteke na ploču jednako 99,75 sekundi što predstavlja očekivanu (srednju) vrijednost. Iz krivulje, se također mogu očitati i ekstremne vrijednosti. Gornji ekstrem predstavlja najduže vrijeme

implementacije i iznosi 104 sekunde, dok donji ekstrem predstavlja najkraće vrijeme implementacije i iznosi 96 sekundi. Dakle, najčešće vrijeme implementacije izvršne datoteke iznosi 99,75 sekundi, dok se ekstremna vremena pojavljuju puno rjeđe i jednaka su 96 sekundi i 104 sekunde.

4.3. Testiranje vremena za detekciju pogreške

Treća faza testiranja projekta bazirana je na određivanju prosječnog vremena koje je potrebno kako bi detektirala/ otkrila pogreška. Uređaj koji se koristio prilikom detekcije pogrešaka je *miniWiggler*, njegova funkcionalnost je otkrivanje sintaktičkih pogrešaka unutar koda, ukoliko one postoje. Testiranje je realizirano izvođenjem dvadeset pokusa, za svaki pokus u tablici 4.3. zapisano je vrijeme koje je bilo potrebno kako bi se detektirala pogreška u kodu, prilikom implementacije izvršne datoteke na ploču. U ovim pokusima izvršna datoteka nikada neće biti spuštena na ploču, jer postoji pogreška i ne moguće je izvršiti operaciju implementacije do kraja. U ovom dijelu rada, unutar programskog koda, namjerno su unesene pogreške kako bi se moglo testirati vrijeme koje je potrebno kako bi prevoditelj detektirao pogrešku. Greške koje su kreirane sintaktičkog su tipa, jer prevoditelj nema mogućnost detekcije logičkih pogreška. Pogreške su sljedeće: poziv funkcije iz biblioteke koja nije implementirana, tipfeler prilikom pozivanja određene funkcije, izostavljanje interpunkcijskog znaka točka zarez, itd. Također, važno je napomenuti da su pogreške kreirane na različitim mjestima unutar koda, kako bi se utvrdilo ovisi li vrijeme detekcije o mjestu na kojem je pogreška.

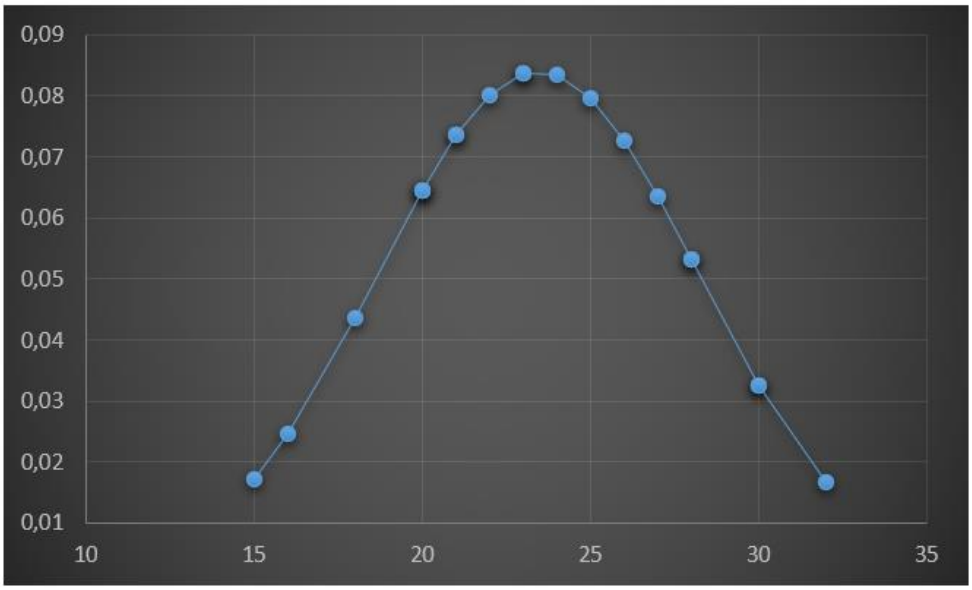
Analizom podataka iz tablice 4.3. dobiveno je prosječno (srednje, očekivano) vrijeme koje je potrebno za detekciju pogreške prilikom implementacije izvršne datoteke na ploču i ono iznosi 23,45 sekundi. Također, u tablici 4.3. nalaze se informacije o podatcima koji su ključni za proračun normalne distribucije.

Primjenom formule (4-1) dobiveno je vrijeme koje je potrebno za detekciju pogreške prilikom implementacije koda na ploču, $\sigma_{detekcije}$ koje iznosi 4,751454 sekundi.

Tablica 4.3. *Vrijeme koje je potrebno za detekciju sintaktičke pogreške*

Oznaka mjerenja:	Vrijeme detekcije:	Vrijeme detekcije u sekundama [s]	Normalna raspodjela
x_1	00:00:15	15	0,01727063
x_2	00:00:20	20	0,06450612
x_3	00:00:22	22	0,08014214
x_4	00:00:24	24	0,08340152
x_5	00:00:18	18	0,04349046
x_6	00:00:16	16	0,02456067
x_7	00:00:23	23	0,08358644
x_8	00:00:27	27	0,0635138
x_9	00:00:30	30	0,03246621
x_{10}	00:00:32	32	0,01663248
x_{11}	00:00:20	20	0,06450612
x_{12}	00:00:21	21	0,07351049
x_{13}	00:00:28	28	0,05308347
x_{14}	00:00:23	23	0,08358644
x_{15}	00:00:30	30	0,03246621
x_{16}	00:00:25	25	0,07961143
x_{17}	00:00:20	20	0,06450612
x_{18}	00:00:28	28	0,05308347
x_{19}	00:00:26	26	0,07270095
x_{20}	00:00:21	21	0,07351049
Podatci koji su korišteni za proračun normalne distribucije:			
Broj mjerenja (n)	20	Srednje (očekivano) vrijeme ($\bar{x}$)	23,45
Najveće vrijeme	32s		
Najmanje vrijeme	15 s	Varijanca (σ^2)	22,576315
Min/Max devijacija	17 s	Standardna devijacija (σ)	4,751454

Dakle, prosječno odstupanje vremena od prosječnog vremena koje je potrebno za detekciju pogreške iznosi 4,751454sekundi. Nadalje, četvrti stupac tablice 4.3. daje informacije o vrijednostima, normalne raspodijele, koje su dobivene primjenom formule (4-2). Na temelju rezultata normalne raspodijele kreiran je graf koji je prikazan na slici 4.2.



Sl. 4.2. *Graf normalne raspodjele za detekciju sintaktičke pogreške*

Na grafu je prikazana Gaussova krivulja koja ima oblik zvona i proteže se od vrijednosti 15 do vrijednosti 32. Krivulja je simetričnog oblika. Iz krivulje se može očitati da najčešće vrijeme detekcije pogreške prilikom implementacije izvršne datoteke na ploču iznosi 23,45 sekundi što predstavlja očekivanu (srednju) vrijednost. Vrijeme detekcije nije ovisno o mjestu na kojem se pojavljuje pogreška. Iz krivulje, se također mogu očitati i ekstremne vrijednosti. Gornji ekstrem predstavlja najduže vrijeme detekcije pogreške i ono iznosi 32 sekunde, dok donji ekstrem predstavlja najkraće vrijeme detekcije i ono iznosi 15 sekundi. Dakle, najčešće vrijeme detekcije pogreške prilikom implementacije izvršne datoteke iznosi 23,45 sekundi, dok se ekstremna vremena pojavljuju puno rjeđe i jednaka su 15 sekundi i 32 sekunde.

5. ZAKLJUČAK

Parcijalno ažuriranje programske podrške vrlo je atraktivna tema u svijetu automobilske industrije. Osim što se primjenjuje u automobilskoj industriji, koncept parcijalnog ažuriranja programske podrške moguće je primijeniti i u drugim proizvodnim sektorima koji se bave ugradbenim sustavima. Razlog je ušteda vremena koje je neophodno za testiranje cjelokupne programske podrške, a samim time skraćuje se i vrijeme za isporuku krajnjim korisnicima. Parcijalno ažuriranje, osim uštede vremena, omogućava uštedu računalnih resursa i dovodi do smanjenja opterećenja samog računala i raspodijeljenih procesa. U okviru rada razvijene su i testirane skripte za demonstrativno parcijalno ažuriranje, skripte nisu univerzalne što znači da su skripte koje su napisane primjenjive samo za projekt na kojem su razvijane, jer je cilj rada demonstrirati funkcionalni postupak koji omogućava univerzalno parcijalno ažuriranje, prikupiti potrebne informacije i postaviti temelja za daljnji razvoj projekta. Razvijeni koncept parcijalnog ažuriranja testiran je na razvojnoj ploči na TC397 Infineon mikrupravljaču, a testiranje je obavljeno pomoću *miniWiggler* uređaja. Testiranje je podijeljeno u 3 ciklusa. Prvim ciklusom testiranja utvrđeno je vrijeme izgradnje izvršne datoteke sa i bez parcijalnog ažuriranja. Normalno vrijeme izgradnje izvršne datoteke je 57 minuta i 6 sekundi, dok je vrijeme izgradnje izvršne datoteke s parcijalnim ažuriranjem jednako 9 minuta i 45 sekundi. Dakle, izgradnja izvršne datoteke pomoću parcijalnog ažuriranja skraćuje vrijeme izgradnje gotovo 6 puta. Drugim ciklusom testiranja dobiveno je prosječno vrijeme koje je potrebno za implementaciju izvršne datoteke na ploču i ono iznosi 99,75 sekundi. Trećim ciklusom testiranja utvrđeno je vrijeme koje je potrebno za detekciju pogreške i ono iznosi 4,751454 sekundi. Iz navedenih rezultata vidljivo je da predloženi koncept rješenja smanjuje vrijeme potrebno za ažuriranje programske potrebe te bi se uz određene modifikacije mogle napraviti skripte koje su univerzalne i primjenjive na svim projektima koji su pisani prema AUTOSAR standardu. Pored automobilske industrije, koncept parcijalnog ažuriranja moguće je proširiti i prilagoditi i za druge industrije koje su vezane za ugradbene sustave.

LITERATURA

[1] „Embedded računala i internet stvari (IoT)“, velj. 09, 2017. https://www.ucionica.net/racunala/embedded-racunala-i-internet-stvari-iot-4038/ (pristupljeno ruj. 08, 2020).

[2] „Specification of Standard Types“, str. 22.

[3] „MotionWise“, *TTTech Auto*. https://www.tttech-auto.com/products/automated-driving/motionwise/ (pristupljeno kol. 26, 2020).

[4] „AURIX™ Family – TC39xXX - Infineon Technologies“. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc39xxx/#!tools (pristupljeno kol. 28, 2020).

[5] I. T. AG, „KIT_MINIWIGGLER_3_USB - Infineon Technologies“. https://www.infineon.com/cms/en/product/evaluation-boards/kit_miniwiggler_3_usb/ (pristupljeno ruj. 07, 2020).

[6] „Hitex: UDE MemTOOL“. https://www.hitex.com/tools-components/development-tools/pls-development-environment/ude-memtool (pristupljeno ruj. 13, 2020).

[7] H. Martorell, J.-C. Fabre, M. Lauer, M. Roy, i R. Valentin, „Partial Updates of AUTOSAR Embedded Applications – To What Extent?“, u *2015 11th European Dependable Computing Conference (EDCC)*, ruj. 2015, str. 73–84, doi: 10.1109/EDCC.2015.18.

[8] „SystemDesk“. https://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm (pristupljeno ruj. 07, 2020).

[9] „Towards a unified formal model for supporting mechanisms of dynamic component update | ACM SIGSOFT Software Engineering Notes“. https://dl.acm.org/doi/10.1145/1095430.1081720 (pristupljeno ruj. 01, 2020).

[10] P. Ruckebusch, S. Giannoulis, I. Moerman, J. Hoebeke, i E. De Poorter, „Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g“, *Internet Things*, sv. 3–4, str. 104–119, lis. 2018, doi: 10.1016/j.iot.2018.09.010.

[11] „Blank Diagram: Lucidchart“. https://app.lucidchart.com/documents/edit/940ef5af-c00f-4f52-baf8-2570f3dc7a10/0_0 (pristupljeno kol. 28, 2020).

[12] „FREE TASKING VX-TOOLSETFOR TRICORE/AURIX | TASKING“. https://www.tasking.com/landing/Free-TASKING-TriCore (pristupljeno ruj. 07, 2020).

[13] „TTTech_MotionWise-Illustration_high-res.jpg (2953×1806)“. https://www.tttech-auto.com/wp-content/uploads/TTTech_MotionWise-Illustration_high-res.jpg (pristupljeno ruj. 08, 2020).

[14] „TASKING announces new Multi Core Performance Tool | TASKING“, kol. 26, 2020. https://www.tasking.com/content/tasking-announces-new-multi-core-performance-tool (pristupljeno kol. 26, 2020).

[15] „Infineon Memtool. Get the software safely and easily.“, *Software Informer*. https://infineon-memtool1.software.informer.com/ (pristupljeno ruj. 08, 2020).

[16] I. T. AG, „Every second car produced today contains Infineon microcontrollers; the company has just delivered its 100 millionth TriCore™ microcontroller - Infineon Technologies“. https://www.infineon.com/cms/en/about-infineon/press/press-releases/2012/INFATV201211-017.html (pristupljeno ruj. 08, 2020).

[17] „DAP miniWiggler V3“, str. 13.

[18] „Lauterbach debugger for Embedded - Nohau Solutions AB“, *Nohau*. http://nohau.eu/products/debugger/ (pristupljeno ruj. 07, 2020).

[19] Tasking, *TASKING VX-toolset for TriCore User Guide*, MA160-800 (v6.2r2) izd. 2018.

[20] „How to Make a Bell Curve in Excel (Step-by-step Guide)“, *Trump Excel*, srp. 28, 2017. https://trumpexcel.com/bell-curve/ (pristupljeno kol. 26, 2020).

[21] „Gaussova krivulja | Hrvatska enciklopedija“. https://www.enciklopedija.hr/natuknica.aspx?ID=21410 (pristupljeno kol. 26, 2020).

Parcijalno ažuriranje programske podrške na AURIX TC397 platformi

SAŽETAK

U ovom radu predložen je postupak parcijalnog ažuriranja programske podrške u ugradbenim sustavima. Parcijalno ažuriranje povećava fleksibilnosti i prilagodljivosti programske podrške na izmjene sustava koji se nalazi u automobilu. Osim toga, parcijalno ažuriranje omogućava uštedu vremena koje je neophodno za testiranje cjelokupne programske podrške, a samim time skraćuje se i vrijeme za isporuku krajnjim korisnicima. Parcijalno ažuriranje, osim uštede vremena, omogućava uštedu računalnih resursa i dovodi do smanjenja opterećenja samog računala i raspodijeljenih procesa. Postupak parcijalnog ažuriranja razvijen je na *MotionWise* platformi, softver je napisan pomoću C programskog jezika i razvijen je prema AUTOSAR standardu. Rad je implementiran na razvojnoj ploči na TC397 *Infineon* mikrupravljaču. Parcijalno ažuriranje realizirano je pomoću dvije verzije programske podrške. Prva verzija je ujedno i osnovna verzija, unutar nje napisane su ključne funkcije koje ispisuju poruke. Druga verzija sadržava samo ono što je izmijenjeno u odnosu na prvu verziju. Testiranjem dobivenih rezultata, zaključeno je da predložena metoda za parcijalno ažuriranje programske podrške uvelike ubrzava proces implementacije izvršne datoteke na ploču.

Ključne riječi: MotionWise, parcijalno ažuriranje, ugradbeni sustavi, C programski jezik, AUTOSAR, Tasking, ušteda vremena

Partial software update on the AURIX TC397 platform

ABSTRACT

This thesis proposes a method for partial updating of software in embedded systems. The partial updating increases flexibility and adaptability of the software to modify the system located in a car. In addition, partial update saves the time necessary to test the entire software, and in the same time shortens the time for delivery to end users. Partial updating, in addition to saving a time, saves computer resources and reduces the load on the computer itself and distributed processes. The partial update is developed on the MotionWise platform, the software is written using the C programming language and developed according to AUTOSAR standard. Example is implemented on development board on the Aurix TC397 Infineon microcontroller. The partial update is realized using two versions of software. The first version is also the basic version and it contains main functions which print messages. The second version contains only what has been changed in comparison to the first version. By testing the obtained results, it was concluded that the proposed method for partial updating of the support program greatly speeds up the process of implementing the executable file on the board.

Keywords: MotionWise, partial update, embedded systems, C programming language, AUTOSAR, Tasking, time saving

ŽIVOTOPIS

Barbara Matijević rođena je 7. rujna 1996. u Osijeku. Završila je Osnovnu školu „Gorjani“ u Gorjanima. Nakon toga upisuje III. gimnaziju u Osijeku koju uspješno završava 2014. godine. Iste godine upisuje preddiplomski studij smjer Računarstva na Fakultetu Elektrotehnike, Računarstva i Informacijskih Tehnologija koji završava 2018. godine. Trenutno pohađa drugu godinu diplomskog studija Informacijskih i podatkovnih znanosti, DRD.

_____

PRILOZI