SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2927

Izrada upravljačkih programa za ugradbene uređaje u tehnologiji modernog C++-a

Hrvoje Bolešić

Zagreb, lipanj 2022.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod

Sve je bitnije proizvesti bolji i kvalitetniji softver. Svaki korisnik želi što bolju moguću aplikaciju, iako je definicja *"što je bolje"* drugačija za svakog pojedinca, nerijetko uključuje efikasnost kao jedan od glavnih parametara. Uz sve napore struke činjenica je ta da je softver sve manje efikasan iako, prema *Moore*ovom zakonu(engl. *Moore's Law*)[9], bi trebalo biti baš obrnuto. *Moore*-ov zakon govori o tome kako se broj tranzistora u integriranim krugovima(engl. *Integrated Circuits, IC*) udvostručava svake dvije godine. Iz toga bi bilo logično zaključiti kako sami integrirani krugovi postaju efikasniji i time da i programi izvršavani na njima bivaju također efikasniji. Na žalost, nije tako jednostavno, u stvarnosti ispada da iako hardver postaje efikasniji, softver je sve manje efikasan. O tome govori, tzv., *Wirth*-ov zakon, koji govori kako brzina software pada više no što brzina hardvera raste. *Wirth*-ov zakon:

> "Software is getting slower more rapidly than hardware is getting faster."[10]

Dodatno, ne govori samo o brzini softvera već i o njegovoj kompleksnosti koja višestruko raste u usporedbi s kompleksnosti hardvera. Objašnjenje zašto se ovo događa leži u činjenici kako današnje aplikacije nude funkcionalnosti koje nisu striktno vezane za glavnu funkcionalnost prvotne aplikacije. Tako na primjer, danas će mnogi uređivači teksta zahtjevati internetsku vezu iako sama primarna funkcionalnost uređivanja teksta nema potrebu za internetskom vezom. Ovo je tipičan primjer gdje aplikacija nudi dodatne funkcionalnosti koje imaju dodatne ovisnosti, konkretno u primjeru internetsku vezu, i time uvode dodatnu kompleksnost u izradu samog proizvoda na benefit korisnika koji dobiva više mogućnosti. Samim uvođenjem dodatnih mogućnosti krajnji proizvod postaje memorijski, a vrlo često i procesivno zahtjevniji.

Dodatna i možda bitnija činjenica zašto softver postaje sve kompleksniji je obilje, često specijaliziranog, hardvera dostupnog za korištenje unutar aplikacija. Poznato je kako smo pred krajem *Moore*-ova zakona, odnosno činjenicom da se tranzistori više neće moći fizički smanjiti kako bi se mogao udvostručiti broj tranzistora na jednom

integriranom krugu. Buduća ubrzanja su jedino moguća tako da se posao raspodijeli na više procesora, odnosno, došli smo do samih limita što je moguće ostvariti na jednom procesoru. Samim time logično je očekivati samo još više specijaliziranih IC-ova unutar budućih platformi kako bi se koristili za izradu aplikacija. Direktna je implikacija da će softver time biti još kompleksniji, što je i za očekivati po *Wirth*-ovom zakonu.

Pitanje je *"Kako se boriti sa kompleksnošću?"*, *"Da li je moguće potpuno elimirati kompleksnost?"*. Na žalost, iskustva pokazuju da je kompleksnost nužna, odnosno moderan softver zahtjeva kompleksna rješenja. Međutim, struka je razvila alate koji mogu pomoći oko kompleksnosti. Do sada vjerojatno najbolji alat za suzbijanje kompleksnosti mora biti **apstrakcija**(engl. *Abstraction*). Iako se pojam abstrakcije gotovo uvijek spominje u kontesktu Objektno orjentiranog programiranja(engl. *Object Oriented Programing, OOP*), sami koncept nije nužno vezan samo za OOP, međutim nedvojbeno je da koncept ima svoje začetke u Objektno orjentiranoj paradigmi(engl. *Object oriented paradigm*). Apstrakcija je proces izgradnje jednostavnijeg modela kompleksnih jedinica tako da sadrži dovoljno informacije za ispunjenje zadatka u određenom kontekstu.

Od dva navedena razloga koja objašnjavaju zašto *Writh*-ov zakon vrijedi, prvi se za konektst ovog diplomskog rada može zanemariti i protumačiti kao tzv. *Code Bloat*, odnosno kompleksnost koja je uvedena zbog dodavanja "nepotrebnog" koda. Drugi razlog je mnogo interesantniji i intrigantniji jer se radi o problemu tehničke naravi, ključnim za ovaj diplomksi rad. Kako je predhodno rečeno razlog povećanja kompleksnosti softvera je dodavanje specijaliziranog hardvera za kojeg je najčešće potrebno izgraditi upravljački program za jednostavniju uporabu uređaja. Konkretno ovaj diplomski rad promatra kako se nositi sa kompleksnosti i izazovima izrade upravljačkih programa za moderne aplikacije na ugradbenim uređajima. Činjenica o tome da softver generalno postaje kompleksniji je možda najviše vidljiva upravo u domeni ugradbenih uređaja, pogotvo tamo gdje se pokušava izgraditi generično riješenje. Moderan softver u domeni ugradbenih uređaja nužno zahtjeva modernije tehnologije nego što je trenutna praksa kako bi se sama domena programske potpore za ugradbene uređaje nastavila razvijati.

1.1. Što je ugradbeni uređaj?

Definicija ugradbenog uređaja se neprestano mijenja kako se razvijaju novi proizvodi s novim izazovima. Prema *Jack G. Ganssle*-ovom i *Michael Barr*-ovom riječniku, *Embedded Systems Dictionary*[2], definicija ugradbenog uređaja je:

> "A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car."

Definicija je osmišljena 2003. godine i za to vrijeme vrlo dobro definira ugradbene uređaje, međutim, danas ne obuhvaća mnoge proizvode koji se smatraju ugradbenim uređajima. Ponajviše ova definicija ne obuhvaća uređaje koje nemaju samo jednu dediciranu funkciju, danas ugradbeni uređaji nerijetko imaju i više funkcija. Neke ličnosti u domeni ugradbenih uređaja također navode ovu definiciju međutim pokušavaju je upotpuniti ili preoblikovati. Jedan od utjecajnijih ličnosti u domeni ugradbenih uređaja, *Dan Saks*, daje svoju jednostavniju definiciju[5]:

> "The job of a computer in an embedded system is to be something other than a general-purpose computer."

Ono što pokušava izreći ovom definicijom je da je ugradbeni uređaj bilo koji uređaj za koje se ne bi smatralo da je računalo. Iako manje precizna definicija, bolje služi u razumjevanju o čemu je riječ kada se govori o ugradbenim uređajima. Drugi ljudi, poput tvorca jezika C++, *Bjarne Straustrup*-a, se ne upuštaju u striktno definiranje[7] ugradbenih uređaja već se više oslanjaju na sličnosti u razvoju softvera među raznim uređajima i priča o razvoju softvera za ugradbene uređaje na osnovi te sličnosti. Kako bi dodatno istaknuli da je definiranje ugradbenih uređaja težak zadatak, jedan od autora prije navedenog riječnika, *Jack. G. Ganssle*, se 2018.[1] godine osvrnuo na danu definiciju i iskazao svoj skepticizam oko nje te tražio čitatelje za sugestije.
Za potrebe ovog diplomskog rada odabiremo definiciju bliže *Dan Saks*-u, te definiramo ugradbeni uređaj kao bilo koji uređaj koji nije smatran računalom generične namjene.

1.2. Što je upravljački program?

Upravljački programi(engl. *Device Drivers*) su[3]:

> "They(device drivers) are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works."

Gornja definicija definira značenje upravljačkog programa u kontekstu *Linux* operacijskog sustava. Općenito operacijski sustavi zahtijevaju upravljačke programe za pojedini hardver kako bi ga mogli utilizirati u samom radu sustava. Ono što je možda prešutno u ovoj defniciji je to da sam operacijski sustav definira jedno sučelje za široki spektar različitih uređaja, što je nužno kako bi se olakšala interakcija između operacijskog sustava i uređaja. Ukoliko bi svaki uređaj imao zasebno sučelje operacijski sustavi bi bili vrlo glomazni i bili bi još kompliciraniji za sami razvoj. Međutim, činjenica da svi uređaji implementiraju jedno definirano sučelje je u ovoj definiciji izrečena samo zato što se govori o upravljačkim programima u kontekstu operacijskih sustava. Za potrebe ugradbenih uređaja nije nužno da upravljački programi imaju identično sučelje, štoviše, u većini slučajeva dizaj kakav se koristi za implementaciju upravljačkih programa u domeni operacijskih sustava bi donio nepotrebnu kompleksnost, nedeterminističnost, vremensku i prostornu složenost, sve odlike nepoželjne u domeni ugradbenih uređaja.
Svi navedeni nedostatci su često razlog zašto se u razvoju programske podrške za ugradbene uređaje ne koristi tehnologija operacijskih sustava te se sam program izvodi na tzv. "golom metalu"(engl. *Bare Metal*), odnsono program se izvodi na procesoru bez dodatnih posrednika poput operacijskih sustava, virtualnih mašina, interpretera i sl. Upravljački programi se svakako mogu gledati kao apstrakcija hardvera i utoliko, kao i što sama definicija govori, moraju biti sposobni donekle sakriti korišteni uređaj. Ovdje se ponovno vidi korištenje abstrakcije kao alata za suzbijanje kompleksnosti, ukoliko ne bi postojao koncept upravljačkih programa, neposredno korištenje uređaja bi unosilo dodatnu kompleksnost u izradu softvera za ugradbeni uređaj. U kontekstu ovog diplomskog rada upravljački programi poštuju gore navedenu definiciju osim što nisu namjenjeni da imaju jedinstveno sučelje jer bi ta činjenica dovela do pesimizacije krajnjeg softvera. Ova pesimizacija je sasvim razumljiva i poželjna u području generičnog računarstva, međutim vrlo je kobna i često se izbjegava u domeni ugradbenih uređaja zbog gore navedenih razloga.

1.3. Moderni C++?

> "C++ is a language for defining and using light-weight abstractions. It has significant strengths in areas where hardware must be handled effectively and there are significant complexity to cope with. This includes many resource constrained systems and much foundational and infrastructure code."

Gornja definicija[8] je data od samog tvorca programskog jezika C++, *Bjarne Stroustrup*-a, i odgovara na pitanje "*što je C++?*". Jezik je napravljen kao nasljednik C programskog jezika s kojim je i danas kompatibilan, gotovo svi C programi su validni C++ programi. C++ posuđuje koncepte iz začetnika objektno orjentirane paradigme, programski jezik Simula, te odlučuje koncepte poput razreda, sučelja, nasljeđivanja i sl. ponuditi korisnicima kako bi kreirali abstrakcije na korisničkoj razini koji su u mnogočemu ekvipotentni apstrakcijama ugrađenim direktno u sam jezik. Ovo je vrlo moćan koncept gdje se korisniku omogućuje da efektivno može poširiti jezik sa vlastitim apstrakcijama bez ulaženja u internu implementaciju i standardizaciju jezika. Doduše nije C++ jedini jezik koji omogućava kreiranje apstrakcija na korisničkoj razini, međutim interna filozofija jezika C++ ga odvaja od drugih jezika. Glavna filozofija jezika C++, koja na neki način vodi daljnji razvoj jezika, može se ukratko izreći kao[6]:

> "What you don't use, you don't pay for."

Gornji citat je ono što razlikuje C++ od toliko drugih jezika, ono što se pokušava izreći se još zove i **Zero-Overhead** princip, točnije govori se o tome kako je jezik koncipiran na način da korisnik može napisati program tako da "ne plaća" ništa drugo osim onoga što mu je potrebno za implementaciju softvera. Ovo se čini možda suvišno za izreći ali u praksi se mnogo puta ispostavlja da su određena svojstva jezika povezana i ukoliko korisnik želi koristiti jedno svojstvo nužno povlači i drugo o kojem ovisi čak iako u trenutnom kontekstu nema logičke ovisnosti, već samo dizajnerske ovisnosti. Ono što se pokušava reći da se dizajneri mnogih jezika odlučuju dizajnirati jezična svojstva na način da određena svojstva jezika nisu međusobno isključiva čak iako logički to jesu. Štoviše, takav dizajn često dovodi do vremenske i prostorne složenosti na korist smanjenja korisničke kompleksnosti. C++ kao jezik koji se reklamira kao moguća opcija u razvoju operacijskih sustava, domeni ugradbenih uređaja, autoindustriji, itd., nije dizajniran tako da "žrtvuje" vremensku i prostornu složenost za smanjenje kompleksnosti. To naravno ne znači kako jezik ne nudi nikakve mehanizme za suzbijanje

kompleksnosti, nego da pesimiziranje vremenske i prostorne složenosti nije prihvatljivo kako bi se smanjila kompleksnost.

Dakako, ništa od predhodno navedenog ne odgovara na pitanje "*što je moderni C++?*". Moderni C++ je često vidljiva sintagma u C++ zajednici i šire koja implicira da postoji modernija inačica jezika C++, to je istina, međutim to ne znači da je to potpuno drugi jezik. Sam jezik C++ nastao je 1985. godine, a 1998. je prvi put standardiziran kao *ISO*[1] standard, iz toga je vidljivo da sam jezik ne asocira na riječ "moderan" jer potječe iz 80-tih godina prošlog stojeća. Međutim, recentniji standardi jezika(2011. godina nadalje) su revitalizirali jezik sa novim, sigurnijim i sve u svemu boljim svojstvima koje se uvelike oslanjaju na moderne tehnike statičke analize i povijesno dobre prakse. Ugrubo gledajući, sintagma "moderan C++" govori o jeziku C++ nakon standarda uvedenog 2011. godine. Dakako standardizacija jezika C++ nije stala i još uvijek je procesu izgradnje međutim C++11 standard se generalno uzima kao početak "modernog C++"-a. Osim C++11 standarda, postoje C++14, C++17 i najnoviji C++20 standard. Za potrebe ovog rada koriste se jezična svojstva iz svih navedenih standarda međutim ne spominje se eksplicitno za svako svojstvo iz kojeg standarda potječe jer je ta činjenica suvišna za kontekst ovog rada.

[1] International Organization for Standardization

2. Moderne tehnologije u programskoj potpori za ugradbene računalne sustave

Stanje danas u razvoju programske potpore za ugradbene sustave je takvo da je još uvijek daleko najpopularnija tehnologija programski jezik C. O ovoj činjnici govori *Dan Saks* u svojoj konferenciskoj prezentaciji[5], gdje jasno prikazuje trendove najpopularnijih tehnologija u domeni ugradbenih uređaja gdje C neupitno prednjači nad bilo kojom drugom tehnologijom. Gledajući retrospektivno kroz povijest može se naslutiti zašto je i danas, u 2022. godini, tehnologija izbora mnogih inženjera u domeni ugradbenih uređaja još uvijek programski jezik C, jednostavno povijesno gledano C je bio dugo vremena prisutan i ustalio se u inženjerskoj praksi kao jedini pravi odgovor. Toliko se ustalio u inženjerskom poslu da ga inženjeri diljem svijeta, u različitim domenama, nazivaju *Lingua Franca*. Dodatno, iz tehničke perspektive, programski jezik C je savršeno dizajniran tako da inženjer ima potpunu kontrolu nad svojim programom, te ostavlja vrlo malo prostora za neku drugu tehnologiju, osim možda asemblerskog koda platforme, uz to postoji i C prevodilac za gotovo svaku postojeću platformu. Postoji ogromna količina softvera napisanih u programskom jeziku C, a neki su i napisani kako bi pospiješili produktivnost samog jezika. Gotovo svi popularni operacijski sustavi danas svoje jezgre imaju razvijene uz pomoć C-a, što direktno povlači i činjenicu da postoji potražnja za C programerima. Programski jezik C je fenomen u računarskoj povijesti i vjerojatno će još dugo vremena biti "zajednički nazivnik" svih tehnologija u računarstvu. Možda najveći razlog zašto C neće tako skoro biti zamjenjen nekom drugom tehnologijom je dizajnerska odluka jezika da jezik vjerno prikazuje platformu na kojoj se izvršava. Ukoliko ništa od navedenog ne uvjerava čitatelja kako je jezik C tu da ostane, onda je tu činjenica da nove generacije inženjera ne mogu ništa izgubiti s poznavanjem programskog jezika C, što znači da je moguće da će u budućnosti biti korišten u obrazovne svrhe.

Sa svim tim rečenim, programski jezik C ima mane i ovaj diplomski rad smatra kako postoje bolje, modernije alterntive u pojedinim kontekstima. Ne opovrgava kako je programski jezik C validan izbor u mnogim slučajevima, međutim tvrdi da mnogi programeri koji danas koriste C, za svoj trenutni projekt bi trebali koristiti nešto drugo, nešto modernije. Također, prije prelaska u raspravu o modernim tehnologijama, vrijedi napomentuti kako programski jezik C nije jedina starija tehnologija korištena u domeni ugradbenih uređaja, ali je toliko dominantna da se može ugrubo reći da i je jedina. Za kontekst ovog rada možemo reći kako je programski jezik C predstavnik starih tehnologija za razvoj programske podrške u domeni ugradbenih uređaja.
Moderne tehnologije, kako to obično biva u inženjerskoj praksi, su zasnovane na prijašnjim tehnologijama - iteriranjem nad starim idejama dolazimo do novih. Kao i za stare tehnologije, kako bi se olakšala rasprava, za svrhu ovog diplomskog rada odabran je predstavnik modernih tehnologija, očekivano, programski jezik C++, odnosno moderna inačica jezika C++. Ovaj odabir nije slučajan, već je izabran iz razloga što je druga najpopularnija tehnologija navedena u konferenciskoj prezentacija *Dan Saks*-a[5] upravo programski jezik C++. Iako možda izgleda šaljivo, kroz ovaj diplomski rad pokušati će se prezenirati izazovi i rješenja modernih tehnologija u izradi upravljačkih programa u ugradbenim uređajima tako što će se uspoređivati odabrani predstavnici starih i novih tehnologija, programski jezik C i C++. Ovo je napravljeno kako bi se pokušalo doći do konkretnih problema i rješenja koja konkretna moderna tehnologija, ovdje C++, rješava. Osim programskog jezika C++ kao predstavnika modernih tehnologija, vrijedi spomenuti vrlo obečavajuću modernu tehnologiju, programski jezik Rust, koja se predstavlja kao tehnologija koja će zamjeniti programski jezik C.
C++, kao predstavnik modernih tehnologija, ima sve funkcionalnosti programskog jezika C, međutim dodatno pruža jezične mehanizme kojima korisnici mogu izgraditi domenske, "zero-overhead" abstrakcije. Ovo je glavna odrednica modernih tehnologija u odnosu na stare, više mogućnosti abstrakcije bez pesimizacije nad vremenskom i prostornom složenošću. Samim uvođenjem abstrakcija u izvorni kod smanjuje se direktna kompleksnost s kojom se inženjer treba nositi, međutim sveopća kompleksnost programa se dakako povećava, međutim bez uporabe abstrakcija kompleksnost bi bila mnogo veća nego bez njih. Činjenica da se u program uvodi abstrakcija direktno implicira da se samo ponašanje sakriva iza njih što mnogi inženjeri domene ugradbenih uređaja tumače kao negativnu posljedicu. Ovo je možda glavni razlog zašto moderne tehnologije još uvijek nemaju zamaha u domeni ugradbenih uređaja.

2.1. Potrebe modernih sustava

Optimalno svaki korisnik bi htio dobiti proizvod koji ne koša ništa, a radi sve i to u bilo kakvim uvijetima, međutim to je nerealno i naivno. Stvarnost je ipak takva da se proizvod dizajnira po uporabi, odnosno korisničkim zahtjevima. To vrijedi i u domeni ugradbenih uređaja i više nego u generičkom računarstvu na klasičnim računalima, jer u parametre sustava u domeni ugradbenih uređaja ulaze i parametri okoline. Mnogi ugradbeni uređaji moraju uzeti u obzir parametre okoline u kojoj je predviđeno da rade. Tako npr. ugradbeni uređaj koji je dizajniran za mjerenje temperature kotla nuklearne elektrane mora uzeti u obzir, naravno, visoku temperaturu i dodatno određene razine radijacije koja bi mogla ometati rad uređaja. Ovo je samo primjer parametara o kojima inženjeri klasičnog računarstva gotovo nikad ne moraju razmišljati, međutim to je svakodnevnica u domeni ugradbenih sustava. Jedan od glavnih parametara u domeni ugradbenih uređaja je možda vijek trajanja baterije iz koje se sam uređaj napaja. Svi navedeni parametri dodaju kompleksnost u izradu samog proizvoda, a time i samog softvera. Kroz povijest softver ugradbenih uređaja je bio nadasve jednostavan, većinom zato što nije bilo očekivanja ni želje da ugradbeni uređaji imaju kompleksnije funkcionalnosti. Odnosno, čak i da je imalo želje za time, jednostavno ugradbeni uređaji tada nisu bili u poziciji osigurati nikakve kompleksnije funkcionalnosti. Današnja očekivanja su puno drugačija, veliki problem softvera danas je taj da je generalna percepcija da je softver lagano izmjeniti i modificrati. Ovo je istina za manje projekte gdje je modifikacija softvera relativno jednostavna, međutim već za imalo veće projekte vrlo je teško modificirati postojeći softver koji nije bio namjenjen za modifikaciju bez uvođenja grešaka. Dodatno, iako su očekivanja za softver ugradbenih uređaja porasla još uvijek se softver ugradbenog uređaja smatra jednostvanim, što je nadasve kriva pretpostvka. Točno je kako softver rađen za točno specijaliziran uređaj može biti jednostavan, međutim nakon izrade specijaliziranog softvera za specifičan uređaj taj softver nije ponovno upotrijebiv za niti jedan drugi, što je veliki propust i kosi se sa ustaljenom inženjerskom praksom.

Moderni sustavi zahtjevaju izgradnju ponovno uporabljivog softvera sa mogućnosti izgradnje programa za specifičan uređaj kao da je sam program građen specifično za taj uređaj. Ovaj zahtjev je ono što je generalno teško postići u domeni računarstva u potpunosti, međutim to ne znači da ne bi trebali težiti ka tom cilju. Stari alati poput programskog jezika C definitivno omogućavaju izgradnju specijaliziranog softvera za pojedini uređaj, međutim nemaju mehanizme koji se mogu usporediti s mehanizmima modernih tehnologija koje omogućavaju izgradnju ponovno upotrebljivog softvera za

druge ugradbene uređaje. Jednostavno rečeno, moderne tehnologije su podobnije za izradu knjižnica i razvojnih okruženja, koji su po definiciji ponovno upotrebljivi, nego bilo koja stara tehnologija. Ovo ne znači kako nitko više ne smije koristiti programski jezik C nego da je potrebno koristiti pravi alat za posao(engl. *Right tool for the right job*).

2.2. Usporedba tehnologija

Programski jezik C++, kao predstavnik modernih tehnologija, pokriva sve funkcionalonsti koje programski jezik C nudi, dakle nema smisla navoditi sve mehanizme koje moderne i stare tehnologije djele jer je to tako već dugi niz godina te će se ovaj diplomski rad baviti onim funkcionalnostima koje moderne tehnologije, C++, nude. Generalno gledano sposobnost kreiranja korisničkih abstrakcija je ono što C++ nudi dodatno uz sve što C nudi, međutim vrijedi konkretno prodiskutirati točne mehanizme koje programski jezik C++ nudi kako bi se izgradile korisničke abstrakcije.

2.2.1. Deterministična kostrukcija i destrukcija

Vrlo kontroverzan mehanizam u skupinama C programera je definitivno deterministička konstrukcija i destrukcija, odnosno u C++ zajednici ovaj mehanizam se često oslovljava sa kraticom **RAII**(engl. *Resource Acquisition Is Initialization*). Radi se o mehanizmu koji služi za automatsko stjecanje i odpuštanje resursa, poput dinamičke memorije, datotečnih opisnika, mrežnih socket-a, itd. Kontroverzan je u krugovima C programera jer sakriva ponašanje, odnosno prevodilac automatski generira kod koji nije eksplicitno napisan u izvornom kodu. Zvuči kao opravdana zabrinutost, nitko ne želi da mu pervodilac izgenerira nešto što mu inženjer nije naložio, međutim prevodilac će izgenerirati samo ono što je inženjer zatražio samo što taj izvorni kod nije vidljiv na mjestu na kojem je kod zapravo izgeneriran. Da pobliže objasnimo o čemu je riječ pogledajmo slijedeći primjer napisan u programskom jeziku C:

```
#include <stdlib.h>

int main()
{
    int* a = (int*)malloc(sizeof(int));
    *a = 2;
    return *a;
```

```
}
```

Gore navedeni primjer izgleda kao potpuno ispravni i validni C program, međutim, iskusni inženjeri će uočiti pogrešku koja može dovesti do problema, a to je ne oslobađanje dinamički alocirane memorije. Memorija alocirana sa ***malloc*** mora nužno biti dealocirana sa pozivom funkcije ***free*** kojoj se mora predati pokazivač koji je inicijaliziran adresom alocirane memorije. Ukoliko se ovo ne napravi dolazi do curenja memorije(engl. *Memory leak*) i može, u ekstremnim slučajevima, dovesti do rušenja cijelog programa. Isti program također je validan C++ program, međutim smjernice modernog C++ bi savjetovale korištenje pametnih pokazivača(engl. *Smart pointers*) umjesto manualne manipulacije nad sirovim pokazivačima(engl. *Raw pointers*). Pravilan C/C++ kod bi izgledao ovako:

```
#include <stdlib.h>

int main()
{
    int* a = (int*)malloc(sizeof(int));
    *a = 2;
    int b = *a;
    free(a);

    return b;
}
```

Dok bi siguran i preporučen kod u modernom C++ izgledao ovako:

```
#include <memory>

int main()
{
    std::unique_ptr<int> a = std::make_unique<int>(2);
    return *a;
}
```

U ovom slučaju koristi se razred ***std::unique_ptr<int>*** koji će na kraju funkcije(nakon ***return***) dealocirati prije alociranu memoriju. Također vidljiva je još jedan smjernica modernog C++, a to je da se ne alocira memorija direktno u korisničkom kodu, odnosno vidljivo je ovdje da nema direktnog poziva ***malloc***, već je za alokaciju memorije

i samu inicijalizaciju zadužen pomoćni funkcijski obrazac ***std::make_unique***[1]. Ono što se zapravo događa u pozadini ovog koda može se opisati slijedećim pseudokodom:

```
int main()
{
    int* a = (int*)malloc(sizeof(int));
    *a = 2;
    return *a;
    free(a); // Called after return.
}
```

Ono što je zanimljivo vidjeti da je pseudokod efektivno jednak pravilno napisanom C programu, štoviše, još bolje izražava što korisnik želi, a to je vratiti vrijednost na koju pokazuje pokazivač ***a*** i nakon toga odpustiti memoriju. Funkcijski, pravilno napisani C i C++ pseudokod su ekvivalentni i rezultirali bi jednakim prevedenim kodom, međutim C++, odnosno C++ pseudo je ekspresivniji jer govori prevodiocu informaciju da se memorija odpušta nakon što se vrijednost na koju pokazuje pokazivač vrati iz funkcije, što nije moguće u programskom jeziku C. Ono s čim treba biti oprezan ovdje je činjenica da se kod izvršava nakon povratka iz funkcije, ovo nije istina, to je samo zgodan opis kada se destrukotor izvršava, međutim kao ni u C ni u C++ nije moguće eksplicitno izvršiti kod nakon povratka iz funkcije.
Važno je reći da je primjer nadasve jednostavan i teško da bi iskusan programer napravio ovakvu grešku u ovom formatu, međutim ovakve greške su česta pojava u većim projektima i mehanizam destruktora omogućava automatsko odpuštanje resursa(u ovom primjeru resurs je memorja). Dodatno, primjer nema nikakvu primjenu u ovom obliku i ne smije se gledati kao ništa više nego edukacijski primjer.
Zaključak ovdje je kako C++ nudi ekspresivniji i sigurniji mehanizam za izražavanje korisničke namjere. Ista funkcionalnost se može dobiti kombinirajući primitive programskog jezika C, međutim time se zamagljuje prava intencija programera i time kod biva sve kompleksniji. Kritika kako je poziv destruktora sakriven se može poistovjetiti sa time da i poziv slobodne funkcije u C-u sakriva ponašanje programa, međutim niti jedan C programer neće zbog te činjenice ne koristiti primitiv funkcije. Samim time što inženjer donosi odluku kako svoj softver piše uz pomoć programskog jezika C++ donosi odluku kako će za svaki konstruirani objekt na kraju njegova životnog ciklusa, ako on postoji, biti pozvan destruktor. Samom tom odlukom poziv destruktora

[1] O obrascima pogledajte ovdje.

postaje eksplicitan jer je poznato da će se uvijek pozvati, a time se i pobija argument kako se sakriva ponašanje. Dakako, postoji i dalje argument da autori knjižnica pisanih u programskom jeziku C++ mogu unositi neočekivano ponašanje kroz destruktore, međutim to mogu i kroz obične funkcije. Jedini problem je to što je programerima teško osvjestiti postojanje destruktora, osvještavanje činjenice da postoje destruktori čini same destruktore eksplicitnima i time pobija argument da postoji implicitno, nevidljivo ponašanje.

2.2.2. Mehanizmi statičkog polimorfizma

Statički polimorfizam je mogućnost tretiranja objekata različitih tipova kao da su isti tip na temelju sličnog ponašanja, međutim za razliku od dinamičkog polimorfizma koji razriješava ponašanje u vremenu izvođenja, statički polimorfizam razriješava ponašanje u vremenu prevođenja. Jednostavnije rečeno, prevodilac odlučuje u vremenu prevođenja koju funkciju će pozvati prevedeni program na osnovu konkretnih tipova koji su predani kao argumenti obrazca ili funkcije. Kako bi pobliže objasnili pojam statičkog polimorfizma pogledajmo nekoliko mehanizama s pripadajućim primjerima s kojima se takva vrsta polimorfizma može ostvariti.

Obrasci(engl. *Templates*) U programskom jeziku C++ postoje dvije vrste obrazaca, funkcijski obrasci i razredni obrasci. O samom mehanizmu obrazca se može razmišljati kao normalnom kodu kojem na pojedinim mjestima nedostaje tip ili vrijednost. Ovakvo banalno objašnjenje je dovoljno za razumjevanje slijedećeg primjera koji pokazuje kako iskonstruirati i koristiti funkcijski obrazac.

```
template<typename T>
T add(T first, T second)
{
    return first + second;
}

int main()
{
    return add<int>(2, 3);
}
```

U gore navedenom primjeru ***add*** je funkcijski obrazac koji kao parametre uzima dva argumenta nekog tipa ***T*** i vraća njihovu sumu. Na osnovu obrazca ***add*** će se u vremenu prevođenja izgenerirati funkcije sa konkretnim tipom umjesto generičnog tipa ***T*** ovisno o tome za koji tip će biti potrebno izgenerirati funkciju. U ovom primjeru poziv ***add<int>(2, 3)*** govori prevodiocu kako je potrebno izgenerirati funkciju ***add*** sa parametrom obrazca ***T*** jednakim ***int***. Postvalja se pitanje "Zar nije ovaj kod mogao biti napisan i programskom jeziku C, bez funkcijskog obrazca?" i odgovor je svakako da je, međutim ono što je ovdje važno za uočiti da se iz funkcijskog obrasca ***add*** mogu izgenerirati funkcije za bilo koji tip ***T*** i to samo onda kada se koriti u korisničkom kodu. Dakle slijedeći kod pokušava ilustrirati prednost funkcijskih obrazaca naspram običnih C funkcija.

```
template<typename T>
T add(T first, T second)
{
    return first + second;
}

int main()
{
    return add<int>(2, 3) +
           add<double>(2.0, 3.0) +
           add<float>(2.0f, 3.0f);
}
```

Umjesto gornjeg C++ koda gotovo identičnu funkcionalnost je moguće postići u programskom jeziku C sa slijedećim kodom. Važno je napomenuti kako je slijedeći kod jedank kodu kojeg prevodilac izgenerira za gornji primjer.

```
int add(int first, int second)
{
    return first + second;
}

double addf(double first, double second)
{
```

```
    return first + second;
}

float addd(float first, float second)
{
    return first + second;
}

int main()
{
    return add(2, 3) + addd(2.0, 3.0) + addf(2.0f, 3.0f);
}
```

Pažljivi čitatelj će zapaziti razliku između C i C++ koda, a to je da se sve funkcije u C programskom jeziku nužno moraju imati jedinstveno ime. Tako je ovdje funckija koja prima dva ***int***-a nazvana ***add***, funkcija koja prima dva ***double***-a nazvana ***addd*** i funkcija koja prima dva ***float***-a nazvana ***addf***[2]. Razlog zašto u programskom jeziku C++ se funkcije mogu zvati jednako je tzv. preopterećivanje funkcija[3]. Osim navedene razlike kodovi su ekvivalentni u funkcionalnosti koju pružaju. U ovom primjeru se opet da zapaziti ekspresivnost i konciznost programskog jezika C++ koji omogućava da se operacija zbrajanja obuhvati s imenom "*add*" neovisno o tipovima nad kojima sama funkcija manipulira. Samim davanjem jednog imena za operaciju smanjuje se komlpeksnost korisničkog koda s istom, ili boljom, razinom ekspresivnosti kao i ekvivalentna implementacija u programskom jeziku C.
Valja napomenuti kako postoje i obrasci razreda, međutim zaključci za korištenje tog mehanizma bi bili jednaki pa je ovaj diplomski rad odlučio izostaviti diskusiju o obrascima razreda.

Preopterećivanje funkcija(engl. *Function overloading*) Mehanizam preopterećivanja funkcija omogućava da sa funkcija zove istim imenom međutim nudi mogućnost drugačije implementacije na osnovu tipova predanih argumenata. Kako bi se bolje ilustriralo o čemu je riječ pogledajmo primjer:

```
#include <iostream>
```

[2] Slični nazivi funkcija se mogu vidjeti u C standardnoj biblioteci.
[3] O preopterećivanju funkcija pogledajte ovdje.

```
void someFunction(int a)
{
    std::cout << "Int\n";
}

void someFunction(float a)
{
    std::cout << "Float\n";
}

int main()
{
    someFunction(2);      // Calls the one that
                          // takes `int` as an argument.

    someFunction(2.0f); // Calls the one that
                          // takes `float` as an argument.
}
```

Preopterećivanje je vrlo jednostavan i intuitivan mehanizam, s gornjim primjerom se pokušava ilustrirati kako će prvi poziv funkcije ***someFunction(2)*** pozvati implementaciju funkcije ***someFunction*** koja prima ***int*** kao parametar, dok će drugi poziv pozvati implementaciju koja prima ***float*** kao parametar. Ovaj jednostavan mehanizam omogućava grupaciju istih operacija, koji ne operiraju nužno nad identičnim tipovima, pod istim imenom, što je vrlo moćan alat u komplesnom softveru gdje nije nužno najbitnija činjenica koja funkcija od zadanih je točno pozvana. Ponovno se vraćamo na svojstvno ekspresije jezika C++ koje je definitivno više pogodno korisniku, a C programski jezik ne nudi ništa slično. Bitno je istaknuti kako preopterećivanje funkcija, iako je pogodan alat za korisničke abstrakcije, ne uvodi nikakvu dodatnu pesimizaciju - generirani kod je identičan kodu koji bi inženjer morao napisati ručno u programskom jeziku C samo bi funkcije trebao nazvati različitim imenima.

Koncepti(engl. *Concepts*) Mehanizam koncepata nije mehanizam koji omogućava statički polimorfizam već oplemenjuje mehanizme statičkog polimorfizma. Mehanizam koncepta je efektivno predikat koji ispituje svojstva ugrađenog ili korisničkog tipa koji je evaluiran u vremenu prevođenja. Sa ovim mehanizmom moguće je napra-

viti tipsku introspekciju i dodatno specijalizirati algoritam ovisno o tipskim svojstvima. Prije uvođenja mehanizma koncepata, programski jezik C++ je mogao ostvariti tipsku introspekciju sa zlogalsnim mehanizmom *SFINAE*(engl. *Substitution Failure Is Not An Error*), koji je nadasve kompliciran i vrlo korisnički nepogodan za korištenje. Glavni razlog uvođenja koncepata u jezik je omogućiti programersku ekspresiju oko njegovih intencija. Ukoliko pogledamo slijedeći primjer, vidljivo je da funkcijski obrazac ***firstCharacter*** ima smisla samo za tipove koji su poput tipa ***std::string***, međutim prevodilac nema nikakvu naznaku da bi se funkcijski obrazac trebao samo koristiti za takve tipove.

```
#include <string>
#include <string_view>
#inculde <vector>

template<typename T>
char firstCharacter(T& stringLike)
{
    return stringLike[0];
}

int main()
{
    std::string first =
        std::string("First");
    std::string_view second =
        std::string_view("Second");
    const char* third = "Third";
    std::vector<int> fourth =
        std::vector<int>({1, 2, 3, 4, 5});

    char char1 = firstCharacter(first);
        // `char1` is "F".

    char char2 = firstCharacter(second);
        // `char2` is "S".
```

```
    char char3 = firstCharacter(third);
        // 'char3' is "T".

    char char4 = firstCharacter(fourth);
        // 'char4' is 1. // This should not compile.

    return 0;
}
```

Pošto prevodilac ne može znati da bi ***firstCharacter*** trebao biti korišten samo sa tipovima sličnim ***std::string***-u, funkcija može biti i istancirana i sa nekim drugim tipom poput ***std::vector<int>***. Ovo je nešto što je neželjeno ponašanje i koncepti nam ovdje mogu pomoći na slijedeći način:

```
#include <string>
#include <string_view>
#inculde <vector>
#include <concepts>

template<typename T>
concept StringLike =
    std::same_as<T, std::string> ||
    std::same_as<T, std::string_view> ||
    std::same_as<T, const char*>;

template<typename T>
requires StringLike<T>
char firstCharacter(T& stringLike)
{
    return stringLike[0];
}

int main()
{
    std::string first =
        std::string("First");
    std::string_view second =
```

```
        std::string_view("Second");
    const char* third = "Third";
    std::vector<int> fourth =
        std::vector<int>({1, 2, 3, 4, 5});

    char char1 = firstCharacter(first);
        // `char1` is "F".
    char char2 = firstCharacter(second);
        // `char2` is "S".
    char char3 = firstCharacter(third);
        // `char3` is "T".
    char char4 = firstCharacter(fourth);
        // Now this does NOT compile.

    return 0;
}
```

U gornjem primjeru definirali smo jednostavan koncept koji je efektivno predikat koji je istinit ako je dani tip ***T*** jednak ***std::string***, ***std::string_view*** ili ***const char****, inače je neistinit. Ukoliko koncept nije zadovoljen, funkcijski obrazac ***firstCharacter*** neće biti moguće istancirati sa zadanim tipom i prevodilac će javiti grešku u vremenu prevođenja. Tako u gornjem primjeru nije moguće pozvati ***firstCharacter*** sa ***std::vector<int>*** substituiranim kao ***T***. Mehanizam koncepata iako nije mehanizam statičkog polimorfizma je usko povezan s njima te ponovno dijeli osobinu ekspresije koja uvelike poboljšava kvalitetu i sigurnost koda u modernim kompleksnim sustavima.

2.2.3. Stroga tipizacija

Kada se priča o strogoj tipizaciji govori se tome da ne postoje mehanizmi, ili ih je jako malo, kojima se mogu uvesti neželjene konverzije između tipova. Programski jezik C++ je poznat po tome što ima vrlo strogu tipizaciju te je vrlo teško napraviti neželjene konverzije, štoviše, postoje mehanizmi jezika kojima se omogućuje eksplicitno zabranjivanje pojedinih implicitnih konverzija. S druge strane, također postoje određene implicitne konverzije između tipova koje postoje kako bi se olakšala čitljivost samog koda, međutim povijesno gledano mnogi bi rekli kako je i tih par praktičnih konverzija opasnost koja uvodi mogućnost greške. S tim rečenim, velika većina jezika C++ je bazirana na vrlo strogoj tipizaciji što je pogotovo vidljivo na korisničkim tipo-

vima. Stroga tipizacija nije odlika programskog jezika C i zbog te činjenice postoje mnoge prilike za uvođenje vrlo teško uočljivih pogrešaka. Slijedeći primjer ilustrira jednu od takvih pogrešaka.

```
int functionTwo(int* p)
{
    return *p;
}

int functionOne(void* p)
{
    return functionTwo(p);
}

int main()
{
    short n = 1;
    return functionOne(&n);
}
```

Gornji primjer ilustrira pogrešku koju je vrlo teško dijagnosticirati, pomni čitaoc će uvidjeti kako se u funkciju ***functionOne*** predaje pokazivač na ***short*** koji se implicitno konvertira u pokazivač na ***void*** što je benigna konverzija sama po sebi. Pogreška je nakon toga taj pokazivač tretirati kao pokazivač na ***int*** i derefencirati ga, što je napravljeno u ***functionTwo***. Razlog zašto je ovo greška je taj što tipovi ***short*** i ***int*** nikada nisu iste veličine, štoviše, ***short*** je uvijek kraći od ***int***. Time što se memorija na kojoj se nalazi ***short*** dereferencirana kao memorija na kojoj se nalazi ***int*** se čita memoriju koja nije inicijalizirana i time čini program nepredvidivim i krivim. Istina je kako je sličan kod u ponekim aplikacijama potreban, međutim problem leži u implicitnosti konverzija. Za prevođenje ovakvog koda uz pomoć C prevodioca korisnik ne bi dobio niti upozorenje jer je ovo sasvim validan C kod i to je ono što bi svakog inženjera trebalo zabrinuti. Ono što programski jezik C++ uvodi je eksplicitnu konverziju između tipova pokazivača i to na nekoliko načina ovisno o ozbiljnosti konverzije. U ovom slučaju korisnik programskog jezika C++ bi bio primoran eksplicitno konvertirati pokazivače ili običnom C konverzijom("***return functionTwo((int*)p)***") ili koristiti preporučeni način konverzije uz pomoć, u ovom slučaju, ***reinterpret_cast***-a("***return***

functionTwo(reinterpret_cast<int*>(p))"). Ovime se ne bi riješio inicijalni problem i program bi idalje bio kriv, međutim bilo bi lakše za diagnosticirati grešku jer bi korisnik trebao pregledati samo sve nesigurne konverzije koje su napravljene sa korištenjem ***reintrepret_cast***-a. Unutar prosječne aplikacije pisane uz pomoć programskog jezika C++ većina eksplicitnih konverzija je moguća bez korištenja ***reinterpret_cast***-a, a slučajevi u kojima je takva konverzija potrebna su generalno dobro osigurani drugim mehanizmima. Čak i u slučaju da je ***reinterpret_cast*** korišten u kontekstu gdje točnost programa nije osigurana nekim drugim mehanizmom idalje programer ima moguće opasne konverzije izolirane na način da samo treba pretražiti kod u potrazi za ***reinterpret_cast***-om. Na kraju valja napomenuti kako korištenje ***reinterpret_cast***-a umjesto obične C konverzije ne donosi nikakvu pesimizaciju u krajnji proizvod, štoviše, ***reinterpret_cast*** se ponaša jednako kao i obična C konverzija samo omogućava drugo ime kako bi izrazilo kako je ova konverzija potencijalno nesigurna.

2.2.4. Standardni kontejneri

Standardni kontejneri predstavljaju skup razreda i algoritama koji su dio C++ standardne knjižnice i omogućavaju funkcionalnosti za lakše rukovanje s kolekcijama podataka. Sami kontejneri su implementirani od strane autora prevodilaca što implicira da su visoko optimizirani. Iako su ovdje navedeni samo standardni kontejneri kao funkcionalnost koju nude moderne tehnologije zapravo je glavna funkcionalnost veća podrška standardne biblioteke koja dolazi u razvojnom paketu koju programski jezik C nema razrađenu do mjere do koje programski jezik C++ ima. Programski jezik C++ u svojoj standardnoj biblioteci ima mnoge korisne razredne obrasce poput ***std::array***, ***std::vector***, ***std::map***, ***std::set***, itd. dok programski jezik C ne nudi nikakvu sličnu alternativu. Istina je kako neki od ponuđenih kontejnera nisu iskoristivi u određenim kontekstima domene ugradbenih uređaja, zbog zahtjeva dinamičke memorijske alokacije, međutim dizajnirani su na način da se mogu koristiti ukoliko se ponudi prihvatljivi memorijski alokator. Dodatno, standardni kojntejneri su odlično dokumentirani što uvelike pomaže u njihovom korištenju. Vrlo dobro razrađene standardne biblioteke su velika funkcionalnost koju programski jezik C++ nudi bez ikakvih vanjskih ovisnosti, dok mnoge starije tehnologije ostavljaju izradu takve funckcionalnosti na svojim korisnicima.

3. Izrada upravljačkog programa

Srž ovog diplomskog rada je izraditi i analizirati upravljački program za određeni periferni uređaj ugradbenog uređaja. Izrađeni upravljački program će biti uspoređen sa implementacijom napisanom u programskom jeziku C od strane proizvođača samog uređaja, ta činjnica bi trebala osigurati najbolju kvalitetu upravljačkog programa. Ekspresivnost izvornog koda je bitan faktor u izradi programskog proizvoda i bitan u ovom diplomskom radu jer je pretpostavka kako su moderne tehnologije ekspresivnije jer imaju bolju sposobnost izricanja inženjerske namjere. Međutim, vrlo je teško objektivno kvantificirati ekspresivnost izvornog koda, tako da će faktor ekspresivnosti biti bez objektivne kvantifikacije, ostavljen na čitatelju da sam procijeni ekspresivnost modernih naspram starih tehnologija. Osim ekspresivnosti izvornog koda, objektivna svojstva od interesa za ovaj diplomski rad su definitivno vrijeme izvođenja i veličina krajnjeg programa. Pretpostavka je kako se ova dva svojstva ne razlikuju između odabranih tehnologija i time dokazuje kako mehanizmi modernih tehnologija ne uvode pesimizaciju ni u jednom od odabranih svojstva.

3.1. Opis platforme

Platforma koja se koristi za provedbu ovog diplomskog rada je razvojna pločica proizvedena od strane kompanije *Aconno* koja koristi mikrokontroler kompanije *Nordic Semiconductors*. Točan mikrokontroler je *nRF52840* koji je vrlo popular u *IoT*(engl. *Internet of things*) domeni, prvenstveno zbog podrške bežične komunikacije putem popularnih protokola poput *Wi-Fi*, *Bluetooth*, *BLE*(engl. *Bluetooth Low Energy*), itd. Osim mogućnosti bežične komunikacije mikrokontroler je obskrbljen sa mnogo dodatne periferije za procesiranje signala, kriptografiju, serijsku komunikaciju, sinkronizaciju, itd. Sam mikroprocesor je izgrađen po *ARM Cortex-M4* arhitekturi koja dolazi s ugrađenom *FPU*(engl. *Floating Point Unit*) jedinicom i sklopovljem za otklanjanjem grešaka, te podržava *ARMv7-M* instrukcijski set.
Gore navedeni uređaj je odabran zbog svoje široko podržane *ARM* arithekture, vrlo do-

bre podrške putem *Nordic Semiconductors* foruma i zajednice inženjera, mnogobrojne i močne periferije, lakoće samog programiranja, podrške modernog prevodioca specifično izgrađenim za *ARM* procesore i mogućnosti korištenja sklopovlja za otklanjanje pogrešaka. Za razvoj programa za sami ugradbeni uređaj koristi se osobno računalo sa operacijskim sustavom *Linux* i ostalim alatima.

3.2. Alati

Za samo učitavanje krajnjeg programa na ugradbeni uređaj koristi se program komandne linije *nrfjprog* pruženog od strane proizvođača samog mikrokontrolera. Alat je vrlo jednostavan za korištenje i za svoj rad zahtjeva samo spajanje razvojne pločice sa osobnim računalom putem USB kabla. Za razvoj samog izvornog koda koristi se razvojno okruženje *CLion* od tvrtke *JetBrains* koji je vjerojatno jedino dobro razvojno okruženje dostupno na *Linux* operacijskom sustavu za razvoj programa uz pomoć programskog jezika C/C++. Verzioniranje koda je ostvareno uz pomoć alata *git* koji je danas, *de facto*, standard za verzioniranje. Iako *git* dolazi kao komandno linijski alat većinom je korištena ekstenzija razvojnog okruženja *CLion* koja indirektno koristi terminalni alat. Verzioniranje koda je bitna sastavnica projekta, međutim ovaj diplomski rad nema veliku povijest iz čega se da zaključiti kako je alat *git* korišten vrlo oskudno. Veći faktor od do sada spomenutih tehnologija je defnitivno alat za izgradnju programa, *CMake*. *CMake* je ekstenzivno korišten, pokušavajući poštovati moderne smjernice koje dolaze sa modernijim verzijama alata. Valja ispraviti grešku i reći kako nije točno reći da je *CMake* alat za izgradnju alata, već je *CMake* generator za alat za izgradnju programa, što komplicira mentalnu sliku o tome što *CMake* jest, međutim to je istina. *CMake* je korišten kako bi se nadomijestilo stari, još i danas popularan alat, *Make*, koji je poput programskog jezika C, dobar alat, međutim današnji alati nude bolje mehanizme za izgradnju projekata. *CMake* u ovom diplomskom radu interno koristi *Make* kao krajnji alat za izgradnju programa, međutim sam projekt nema nikakve veze sa alatom *Make* te se potencijalno može koristi drugi alati za izgradnju programa poput, danas popularnog, *Ninja*-e.

Svi dosada nevedeni alati su striktno pomoćni alati u razvoju softvera, alat bez kojega bi svi ovi alati bili suvišni jest prevodilac(engl. *Compiler*). Točnije sam prevodilac je alat koji se sastoji od niza alata, zato se ponekad naziva i lanac alata(engl. *Toolchain*). U okviru ovog diplomskog rada koristi se *GNU*, C i C++ prevodilac, točnije

ARM GNU lanac alata verzije 10.3[1]. Ovo je vrlo moderna verzija *GNU* prevodioca koja je potrebna za kontekst ovog diplomskog rada kako bi se pokazale mogućnosti modernog C++. Kako bi se izrađeni upravljački program mogao uspoređivati s nečim, dodatno je bilo potrebno uvesti ovisnost o *nRF5 SDK*-u(engl. *Software Development Kit*) koji sadrži implementacije svog softvera za *Nordic Semiconductors* proizvode, pa tako i za *nRF52840* mikrokontroler. Uz *Nordic Semiconductors* SDK kako bi realizirali povratnu informaciju, sa ugradbenog uređaja nazad na osobno računalo, dodatno je uvedena ovisnost o *Segger RTT* modulu koji omogućava korištenje standardnih funkcija poput ***printf*** tako da se poruke proslijede osobnom računalu koje koristi alat *JLinkRTTViewer* kao serijski monitor za prihvaćanje proslijeđenih poruka. Na taj način korisnik može lakše analizirati ponašanje samog ugradbenog uređaja. Ukoliko je potrebno, koristi se *GDB*(engl. *GNU Debugger*) zajedno sa *JLinkGDBServer*-om kako bi se zaista duboko analizralo samo izvršavanje mašinskog koda. Kako bi se analizirali krajnji produkti, objektni kod i krajnji program, koristili su se alati iz paketa alata *binutils* koji dolazi zajedno s *GNU* lancom alata.

3.3. Postav

Projekt ovog dimplomskog rada je osmišljen kao jednostavna okolina za mjerenje(engl. *Benchmark envirnoment*). Baza projekta je zapravo program koji predstavlja jedno okruženje za mjerenje, a kontekst koji se mjeri se uvodi kao ovisnost putem primitiva funkcije zajedno sa dodanim svojstvima samog mjerenja. Točnije baza projekta je izvršni program koji ovisi o funkciji čije će vrijeme izvršavanja biti mjereno i potom ispisano prosječno vrijeme izvršavanja, broj ciklusa i totalno vrijeme izvođenja. Kako bi pomnije objasnili kako izgleda sama baza projekta pogledajmo izvorni kod datoteke u kojoj se nalazi ulazna točka programa.

```
#include <cstdio>
    // `std::printf`
#include <Benchmark.hpp>
    // `BenchmarkDescriptor`, `benchmark`
#include <Config.hpp>
    // `name`, `repeat`, `run`
#include <Stopwatch.hpp>
    // `Stopwatch`
```

[1]Link na stranicu prevodioca: https://developer.arm.com/downloads/-/gnu-rm

```
static const Benchmark::BenchmarkDescriptor
    descriptor{name(), repeat(), &run};

int main() {
    Benchmark::Meta::Stopwatch auto
        stopwatch{Stopwatch::Stopwatch{}};

    if (not stopwatch.init())
    {
        std::printf("Can't init stopwatch!");
        return 1;
    }

    const auto [totalTime]
        {Benchmark::benchmark(stopwatch, descriptor)};
    const auto [name, cycles, dummy]
        {descriptor};

    std::printf(
        "Name: %10s\n\tTotal time: %luus\n\tCycles: %lu\n
            \tAverage: %luus\n",
        name.data(),
        static_cast<std::uint32_t>(totalTime.count()),
        static_cast<std::uint32_t>(cycles),
        static_cast<std::uint32_t>(totalTime.count() /
            cycles)
    );
}
```

U datoteci je vidljvo nekoliko interesantnih dijelova, jezgra cijelog programa je enkapsulirana unutar razreda ***Benchmark::BenchmarkDescriptor*** koji opisuje kontekst koji će biti mjeren. Konstrukcija ovog tipa zahtijeva ime mjerenog konteksta, broj ponavljanja izvršavanja te samu funkciju koja će se izvršavati. Sve informacije za inicijalizaciju instance ***Benchmark::BenchmarkDescriptor*** su uvedene putem funckija ***name***, ***repeat*** i ***run***. Same deklaracije funkcija su navedene u zaglavlju ***Config.hpp***, međutim

same implementacije su odvojene od baznog programa kako bi mogli izgraditi više programa sa drugim kontekstima za mjerenje. U kasnijem poglavlju su izdvojene konkretne implementacije za navedene funkcije ovisno o kojem kontekstu je riječ.
Slijedeća bitna sastavnica baznog izvršnog programa je štoperica(engl. *Stopwatch*) koja je enkapsulirana unutar razreda ***Stopwatch*** te je implementirana kao adapter na već postojeći upravljački program za timer koji je implementiran od strane *Nordic Semiconductors* kompanije. Razred ***Stopwatch*** se koristi samo u kontekstu ovog diplomskog rada i zato sama implementacija nije toliko ekstenzivna i prenosiva jer nije bilo potrebe za ponovnu uporabu. Prvotna ideja diplomskog rada nije uopće uključivala praćenje vremena na samom ugradbenom uređaju već je bila ideja eksternalizirati praćenje na osobnom računalu ili sl. Međutim, kroz vrijeme razvoja se ispostavilo kako su brzine izvođenja programa bile prevelike i kako bi greška sa eksternaliziranim praćenjem vremena bila prevelika, štoviše, toliko velika da se ne bi mogao izvući nikakav objektivan zaključak iz dobivenih mjerenja. Zbog toga je odlučeno praćenje vremena internalizirati na ugradbenom uređaju uz pomoć perifernog uređaja timera. Dodatno, kako bi se olakšalo praćenje vremena implementacija razreda ***Stopwatch*** interno koristi već postojeći upravljački program implementiran od strane *Nordic Semiconductors* kompanije. Altrenativa je bila samostalno izgraditi novi upravljački program, međutim time bi se još dodatno zakomplicirao razvoj cijelog projekta pa je odlučeno koristiti već izgrađeno sučelje za periferni uređaj timer-a.
Komponenta glavnog programa koja zapravo odrađuje izvršavanje funckcije konteksta, ***run***, jest poziv funkcije ***Benchmark::benchmark*** čija je deklaracija definirana u zaglavlju ***Benchmark.hpp***. Funkcija nakon mjerenja vraća totalno vrijeme potrebno za izvođenje te se potom to vrijeme u glavnoj funkciji dijeli sa brojem ciklusa kako bi se dobilo prosječno vrijeme izvođenja i poptom se ispisuje pomoću funkcije ***std::printf***. Bitno je ovdje napomenuti kako je u okruženju ovog programa nadomješten sistemski poziv funkcije ***write*** s implementacijom od strane *Segger RTT* sistemskog poziva. Zbog ove promjene funkcija ***std::printf*** proslijeđuje poruku, putem *RTT* protokola, prema osobnom računalu kako bi korisnik lakše mogao pregledati rezultate mjerenja. Za detalje, čitatelja se potiče da pogleda točne postavke projekta kako bi saznao točno kako je ovo napravljeno jer takvi detalji nisu bitni za kontekst rada.
Valja podiskutirati i analizirati samu implementaciju funkcije ***Benchmark::benchmark*** jer se ispostavlja kako nije trivijalna za implementirati. Izvorni kod funkcije je dan u nastavku.

```
Benchmark::BenchmarkResult Benchmark::benchmark(
```

```
    PolymorphicStopwatch stopwatch,
    const BenchmarkDescriptor &benchmarkDescriptor
) noexcept
{
    Benchmark::BenchmarkResult result{.totalTime = std::
        chrono::microseconds{}};

    warmup(1'000);

    for (std::size_t i{0}; i < benchmarkDescriptor.repeat
        ; ++i) {
        stopwatch.begin();
        benchmarkDescriptor.run();
        result.totalTime += stopwatch.time();
    }

    return result;
}
```

U implementaciji funkcije je vidljiva varijabla ***result*** koja je vraćena kao konačna vrijednost iz funkcije, a prvotno je inicijalizrana sa vrijednostću ***0ms*** te se unutar petlje taj broj povećava za vrijeme izvođenja jednog izvršavanja konteksta, odnosno funckije ***run*** koja je doustupna putem ***benchmarkDescriptor***-a. Ono što je zanimljivo u ovoj implementaciji je poziv funkcije ***warmup(1'000)***. Ova funkcija je napravljena kako bi se mjerenje moglo odraditi još preciznije bez mogućnosti utjecaja procesorskih komponenti poput prediktora granjanja. U prošlim iteracijama funckije ***Benchmark::benchmark*** nije postojao ovaj poziv i time je dolazilo do nedeterminističkih mjerenja koja su bila uočljiva na testom primjeru. Kako bi objasnili kako konstruirati primjer, odnosno konteskt mjerenja pogledajmo testni primjer.

Testni primjer se sastoji od dvije datoteke izvornog koda, jedan koji je napisan koristeći programski jezik C, dok je drugi napisan koristeći programski jezik C++. Doljnji lijevi primjer prikazuje testni primjer napisan u programskom jeziku C, dok desni primjer prikazuje ekvivalentni primjer napisan u tehnologiji C++.

```
void run()
{
    for (size_t i = 0U; i <
        100; ++i)
    {
        __asm__ volatile(
            "nop"
        );
    }
}

const char *name()
{
    return "C::Test";
}
size_t repeat()
{
    return 10000;
}
```

```
void run()
{
    for (size_t i = 0U; i <
        100; ++i)
    {
        asm volatile(
            "nop"
        );
    }
}

const char *name()
{
    return "C++::Test";
}
size_t repeat()
{
    return 10000;
}
```

Testni primjer je konstruiran na način da bude identičan u obije tehnologije kako bi dokazali da je izkonstruirani sustav za mjerenje validan. Dodatno je pregledan izgenerirani asemblerski kod koji je identičan u oba slučaja tako da se sa sigurnošću može reći kako su primjeri identični, odnosno ***run*** funckije. Uz identične ***run*** funkcije gotovo su identične i ostale dvije funkcije međutim postoji mala razlika u imenu kako bi se raspoznao jedan kontekst od drugoga. Neki čitatlji će se pitati zašto se u jednoj verziji koristi ***__asm__***, a u drugoj ***asm***, te zašto se uoće koristi takav konstrukt. Odgovor na pitanje zašto se koristi je taj da ukoliko bi ostavili ***for*** petlju praznom prevodilac bi potpuno izostavio(optimizirao) petlju i funkcija efektivno ništa ne bi radila. Kako bi to izbjegli koristi se tzv. *inline assembly* kako bi se spriječilo prevodioca da optimizira petlju. Nadalje, odgovor na pitanje zašto se razlikuju sintakse je jednostavno taj da programski jezik C nema ključnu riječ ***asm*** već ***__asm__***, međutim funkcionalno za ovaj primjer rade identičnu stvar.

Nakon što je objašnjeno zašto i kako je izkonstruiran testni primjer valja reći kako su vremenski rezultati za oba programa, neovisno o tehnologiji, bili identični. Dakle, sustav za mjerenje je validan i može se koristiti za daljnja mjerenja. Ukoliko bi se vre-

mena izvođenja drastično razlikovala između dvaju verzija to bi značilo kako sustav za mjerenje nije napravljen dobro, srećom to ovdje nije slučaj.

3.4. Upravljački program

Upravljački program koji je izrađen u sklopu ovog diplomskog rada je upravljački program za generator slučajnih brojeva(engl. *Random number generator*). Uređaj za generiranje slučajnih brojeva je nadasve jednostavan uređaj sa svega nekoliko funkcija međutim dovoljno je kompleksan za kontekst ovog diplomskog rada. Sam uređaj je mapiran u memoriju mikrokontrolera te sadrži nekoliko bitnih registara, registar za pokretanje generatora slučajnih brojeva, registar za zaustavljanje generatora slučajnih brojeva, registar za pohranu slučajnog broja, registar za zastavicu koja govori da li je slučajan broj spreman za čitanje, odnosno da li je slučajan broj izgeneriran do kraja i na kraju registar za omogućavanje iznimke kada je slučajan broj izgeneriran. Iz posljednja dva navedena registra može se zaključiti kako sam uređaj može raditi na dva načina, uz pomoć mehanizma iznimaka i uz pomoć radnog čekanja. U nastavku pogledajmo relevantno sučelje upravljačkog programa koje nudi *nRF5 SDK*.

```
typedef void (* nrfx_rng_evt_handler_t)(uint8_t rng_data)
    ;
typedef struct
{
    bool     error_correction : 1;
    uint8_t  interrupt_priority;
} nrfx_rng_config_t;

nrfx_err_t nrfx_rng_init(nrfx_rng_config_t const *
    p_config, nrfx_rng_evt_handler_t handler);
void nrfx_rng_start(void);
void nrfx_rng_stop(void);
void nrfx_rng_uninit(void);

void nrf_rng_int_enable(uint32_t mask);
void nrf_rng_int_disable(uint32_t mask);
bool nrf_rng_int_get(nrf_rng_int_mask_t mask);
uint8_t nrf_rng_random_value_get(void);
```

Dva definirana tipa ***nrfx_rng_evt_handler_t*** i ***nrfx_rng_config_t*** su tipovi koji definiraju prototip funkcije koja može primiti novo izgenerirani slučajni broj i tip koji enkapsulira postavke upravljačkog programa pri inicijalizaciji. Slijedeće četiri funkcije dosta očito služe za inticijalizaciju, pokretanje, zaustavljanje i deinitcijalizaciju uređaja. Isto tako slijedeće četiri funkcije u nastavku pružaju funkcionalnost dozvoljavanja/nedozvoljavanja iznimke kada se broji izgenerira, dohvaćanje zastavice koja govori o tome da li je iznimka omogućena ili onemogućena te poslijednje za dohvaćanje same slučajno generirane vrijednosti. Sučelje kakvo je prikazano ostvaruje svoju zadaću, pruža korisniku funkcionalnost generiranja slučajnih brojeva putem iznimaka, međutim ne postoji direktan način za ostvariti dohvaćanje slučajnog broja koristeći radno čekanje. Ovime se korisnika prisiljava na korištenje mehanizma iznimki čak iako to nije njegova želja, time ovaj upravljački program biva manje fleksibilan. Dodatno bitno je uočiti kako navedene strukture i funkcije ne mogu biti enkapsulirane u razred ili slično jer odabrana tehnologija za implementaciju, programski jezik C, ne podržava takve mehanizme.
S druge strane pogledajmo relevantno sučelje upravljačkog programa implementiranog u programskom jeziku C++.

```
namespace Rng
{

enum class Policy { Polling, Interrupt };

template <Policy kPolicy = Policy::Interrupt>
class Driver {
    /*--- Help type ---*/
    struct ConditionStruct {
        bool isDone() const noexcept;
        void waitOn() const noexcept;
    };

    /*--- Constructor ---*/
    Driver() noexcept;

    /*--- Methods ---*/
```

```
    std::uint8_t
    get() noexcept
    requires(kPolicy == Policy::Polling);

    template <std::size_t N>
    std::array<std::uint8_t, N>
    get() noexcept
    requires(kPolicy::value == Policy::Polling);

    ConditionStruct
    set(
        std::invocable<std::uint8_t> auto &userHandler
    ) noexcept
    requires(kPolicy::value == Policy::Interrupt);
};
}
```

Na prvi pogled izgleda intimidirajuće sučelje, međutim puno je ekspresivnije i sigurnije za korištenje u odnosu na prije pokazani upravljački uređaj. Prvo valja napomenuti kako je u ovom slučaju upravljački uređaj realiziran uz pomoć obrasca razreda koji kao svoj jedini argument prima mod rada upravljačkog programa(engl. *Policy*) i tu odluku ostavlja na samom korisniku i time postaje više ekstenzivan od svojeg konkurenta. Odlukom korisnika o modu rada, korisnik raspolaže sa pripadajućim sučeljem, zapravo se u ovoj definiciji obrasca razreda nalaze dva sučelja ovisno o modu rada. Desno je sučelje u modu rada koji korsti radno čekanje dok je lijevo sučelje koje koristi iznimke.

```
class Driver<Policy::Polling>
    {
     /*--- Constructor ---*/
     Driver() noexcept;

     /*--- Methods ---*/
     std::uint8_t
     get() noexcept;

     template <std::size_t N>
     std::array<
          std::uint8_t,
          N
     >
     get() noexcept;
};
```

```
class Driver<Policy::Interrupt
    > {
     /*--- Constructor ---*/
     Driver() noexcept;

     /*--- Methods ---*/
     ConditionStruct
     set(
     std::invocable<
          std::uint8_t
     >
          auto &userHandler
     ) noexcept;
};
```

Valja reći kako desno sučelje ima dvije metode, međutim prva navedena je zapravo specijalizacija druge. Prva metoda vraća jedan slučajan broj dok druga metoda vraća njih ***N***, korisnik može koristiti što god mu odgovora. S druge strane sučelje koje se bazira na mehanizmu iznimki ima samo jednu metodu koja uzima kao argument funktor[2] koja prima novo izgenerirani broj. Funkcija također vraća tip nad kojim je moguće provjeriti da li je generator izgenerirao dovoljno brojeva te je moguće čekati dok generator ne izgenerira dovoljno brojeva. Kako bi generator znao kada je dovoljno brojeva izgenerirano funktor vraća ***bool*** vrijednost kojom može odrediti koliko slučajnih brojeva je dovoljno za potrebe korisnika.

3.5. Konteskt mjerenja

Kako je u prijeašnjem poglavlju "Postav" bilo objašnjeno za testni primjer, tako je u ovom poglavlju konstruiran kontekst mjerenja, odnosno primjer, koji demonstrira korištenje oba od prezentiranih upravljačkih programa. Primjer koji je konstruiran je nadasve jednostavan, upravljački uređaj se koristi kako bi se izgeneriralo 1000 slučajnih brojeva od 0 do 255 te ih se potom sumira i ispisuje njihova suma. Valja napomenuti da ovako konstruiran primjer ne mjeri samo vrijeme generiranje brojeva već i vrijeme

[2]Funktor je objekt koji se ponaša kao funkcija

trajanja ispisa i vrijeme trajanja sumacije brojeva, međutim jedini dio koji se mjenja unutar ovog primjera jesu implementacije upravljačkih programa sve ostale komponente su konstantne. To je bitno za napomenuti kako se krajnje prezentirani rezultati ne bi smatrali kao absolutno vrijeme generiranja brojeva. Važno je bilo iskonstruirati primjer koji je kompliciraniji od samog generiranja slučajnih brojeva kako bi primjer bio donekle vjerodostojan stvarnoj uporabi. Dodatno valja napomenuti kako su zapravo prevedena tri konteksta za mjerenje iz razloga što se upravljački program u novijoj izvedbi, uz pomoć programskog jezika C++, može koristiti na dva načina te je smatrano kako bi bilo interesantno usporediti vrijeme izvođenja i veličinu krajnjeg programa između ta dva načina korištenja upravljačkog programa. Na kraju valja spomenuti kako sam uređaj za generiranje slučajnih brojeva posjeduje svojstveno vremenske nedeterminističnosti na razini mikrosekunda, odnosno za generiranje jednog broja nije uvijek potrebno identična količina vremena, međutim iako je ovo istina mikrosekunde u slučaju ovog rada nisu presudne u rezultatima mjerenja. Zbog nedeterminističnosti uređaja postojat će određena mala devijacija između vremena međutim to će biti par mikrosekundi razlike koje ne bi trebale previše utjecati na cijelokupno mjerenje.

4. Rezultati

Tablica 4.1: Rezultati mjerenja

Compile mode	**Debug**			**Release**		
Technology	**C++**		**C**	**C++**		**C**
Driver mode	**Interrupt**	**Polling**	**Interrupt**	**Interrupt**	**Polling**	**Interrupt**
Program size[kB]	983	983	946	827	826	833
Program execution time[μs]						
1	27797	27883	27450	27141	27147	27157
2	27816	28263	27551	27117	27170	27135
3	27832	27883	27598	27139	27147	27125
4	27810	27888	27441	27135	27139	27120
5	27818	27948	27499	27142	27165	27132
6	27842	27879	27495	27144	27172	27122
7	27834	27871	27470	27141	27171	27207
8	27791	27866	27502	27145	27161	27117
9	27787	27866	27501	27129	27146	27113
10	27797	27884	27474	27136	27153	27139
Avarage execution time	27812.40	27923.10	27498.10	27136.90	27157.10	27136.70
Deviation	19.31	121.70	46.85	8.45	12.16	27.82

Iznad je prikazana tablica sa rezultatima mjerenja vremena izvođenja konkeksta koji koristi uređaj za generiranje slučajnih brojeva uz pomoć upravljačkih uređaja. Jedan primjer koristi upravljački program koji je implementiran uz pomoć programskog jezika C, a drugi upravljački program implementiran uz pomoć programskog jezika C++. Iz tablice se može vidjeti i veličine pojednih program, točnije prikazane su veličine *ELF*[1] izvršnih datoteka i te veličine su samo proporcionalne veličini stvarnog programa na mikrokontroleru, odnono, veličine se mogu jedino uspoređivati međusobno međutim ne predstavljaju absolutne veličine programa stavljenog na sam mikrokontroler. Iz tablice je također vidljivo kako je za svaki kontekst, u svrhu mjerenja vremena

[1] *ELF(engl.* Executalble and Linkable Format*)* je format izvršnih datoteka koji se koristi na *UNIX* operacijskim sustavima poput *Linux* operacijskog sustava.

izvršavanja, napravljeno 10 nezavisnih mjerenja te je naposlijetku izražena srednja vrijednost i standardna devijacija svakog konteksta mjerenja. Cijeli proces je ponovljen za dva načina prevođenja, *Debug* i *Release*. Valja ponovno istaknuti kako vremena u tablici mjere izvršavanje konstuiranog primjera, objašnjenog u predhodnom poglavlju, a ne samu brzinu generiranja slučajnih brojeva od strane samog perifernog uređaja, jer takvo mjerenje mjerilo samo nedeterminističnost samog perifernog uređaja, a ne brzinu izvršavanja napisanog softvera.

5. Zaključak

Ukoliko pogledamo tablicu rezultata vidljivo je kako je veličina programa u određenom modu prevođenja izgrađena uz pomoć programskog jezika C++ uvijek proporcionalno manja s obzirom na tehnologiju programskog jezika C. Ovi rezultati ne govore o tome da je to istina za svaki slučaj međutim pobija teze koje govore da su programi izgrađeni uz pomoć jezika C++ uvijek veći od funkcijski ekvivalentnih C programa. Ovdje se vidi snaga ekspresije modernog programskog jezika koja omogućava provodiocu da elminira određene dijelove koda te omogućava ekstenzivnije i agresivnije optimizacije. S druge strane ako se pogledaju prosječna vremena izvođenja vidljivo je kako programski jezik C još uvijek prednjači programskom jeziku C++ u *Debug* modu prevođenja. Ovo je za očekivati jer programski jezik C++ je nešto kompleksniji od programskog jezika C i time gubi na vremenu izvođenja bez uključivanja optimizatora na višim razinama optimizacije. Kada se osposobi optimizator u *Release* modu prevođenja može se vidjeti kako su prosječna vremena izvođenja identična za slučajeve kada se upravljački program koristi u modu s iznimkama. Upravljački program u modu s radnim čekanjem uvijek biva najsporija opcija što se vrlo vjerojatno može prepisati samom dizajnu takvog moda, a ne samoj tehnologiji izvedbe. Radno čekanje se općenito izbjegava jer troši procesorske resurse, a kao što je ovdje i pokazano, ne daje zadovoljavajuće rezultate. Vjerojatno su se zato i dizajneri unutar *Nordic Semiconductors* kompanije i odlučili kako neće podržati mod radnog čekanja unutar upravljačkog programa za generator slučajnih brojeva, međutim tom odlukom je implicitno rečeno da mikrokontroler mora koristiti mehanizam iznimki koji ponekad nije dostupan ili nedopušten. Na stranu sa implementacijom upravljačkog programa u modu s radnim čekanjem, oba upravljačka programa u modu s inznikama daju jednako vrijeme izvođenja, odnosno mala greška se ovdje pripisuje nedeterminističnosti uređaja za generiranje slučajnih brojeva i zaključujemo kako su vremena izvođenja identična. Ovime se pokazuje kako se korištenjem pomno odabranih modernih tehnologija ne žrtvuje vrijeme izvođenja i kako korisnik može imati potpunu moć da time što će na kraju prevodilac emitirati u krajnji izvršni program. Ono što još valja reći kako je kvaliteta i ekspresivnost koda

puno bolja u modernih tehnologija, ovo svojstvo nije kvantificirano, međutim čitatelja se snažno potiče da pregleda izvorne kodove programa i procijeni sam. Ono što valja reći je to da je istina kako moderne tehnologije, naročito programski jezik C++, jesu kompliciranije u odnosu na tehnologije poput programskog jezika C, međutim upravo zbog te kompleksnosti daju mogućnost ekspresije koju čak i prevodilac može razumjeti i na osnovu tih informacija generirati bolji, sigurniji i manji program. Svojstvo ekspresivnosti kroz koncept abstrakcije je svojstvo koje moderne tehnologije nude u odnosu na stare i to će se u budućnosti samo perpeturati kako bi se stare tehnologije istisnule iz domena koje bi stvarno trebale koristiti nešto modernije. Na kraju treba reći kako će programski jezik C uvijek imati mjesto u budućnosti računarstva međutim, kako to obično biva, inžinjeri moraju osvjestiti kako programski jezik C nije uvijek najbolji alat za sve probleme, pogotovo u domeni ugradbenih uređaja, i kako postoje bolji, moderniji alati, poput programskog jezika C++.

LITERATURA

[1] Jack Ganssle. What's embedded? https://www.embedded.com/whats-embedded/, 3 2008. Pristupljeno: 19.5.2022.

[2] J.G. Ganssle i M. Barr. *Embedded Systems Dictionary*. R and D Developer Series. Taylor & Francis, 2003. ISBN 9781578201204. URL https://books.google.hr/books?id=zePGx82d_fwC.

[3] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *An Introduction to Device Drivers*, stranice 1–12. O'Reily Media, Inc., 1005 Gravestein Highway North, Sebastopol, CA 95472, 2005.

[4] Dan Saks. extern c: Talking to c programmers about c++. CppCon 2016. CppCon, 2016. Youtube link: https://www.youtube.com/watch?v=D7Sd8A6_fYU.

[5] Dan Saks. Writing better embedded software. Meeting Embedded 2018, Meeting C++ 2018. Meeting C++, 2018. Youtube link: https://www.youtube.com/watch?v=3VtGCPIoBfs.

[6] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA, 1995.

[7] Bjarne Stroustrup. Keynote: What can c++ do for embedded systems developers? NDC Conferences. NDC Conferences, 2018. Youtube link: https://www.youtube.com/watch?v=VoHOLDdfDhk.

[8] Bjarne Stroustrup. The c++ programming language. https://www.stroustrup.com/C++.html, 8 2021. Pristupljeno: 23.5.2022.

[9] Techopedia. Moore's law. https://www.techopedia.com/definition/2369/moores-law, 6 2012. Pristupljeno: 19.5.2022.

[10] Techopedia. Wirth's law. https://www.techopedia.com/definition/24381/wirths-law, 8 2018. Pristupljeno: 19.5.2022.

Izrada upravljačkih programa za ugradbene uređaje u tehnologiji modernog C++-a

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: C++, C, Ugradbeni uređaj, Programska potpora, Upravljački program, Moderne thenologije

Title

Abstract

Abstract.

Keywords: C++, C, Embedded device, Software, Device driver, Modern technology