Univerzitet Singidunum

SISTEMI ZA RAD U REALNOM VREMENU

Real-Time Systems

Marko Tanasković
Bojan Gutić

Beograd, 2025.

UNIVERZITET SINGIDUNUM

Marko Tanasković
Bojan Gutić

SISTEMI ZA RAD U REALNOM VREMENU

PRVO IZDANJE

Beograd, 2025.

SISTEMI ZA RAD U REALNOM VREMENU

Autori:
dr Marko Tanasković,
mast. Bojan Gutić

Recenzenti:
Prof. Mladen Veinović
Prof. Marina Marjanović

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32
www.singidunum.ac.rs

Za izdavača:
dr Milovan Stanišić

Tehnička obrada:
Miloš Višnjić

Dizajn korica:
Aleksandar Mihajlović, MA

Godina izdanja:
2025.

Tiraž:
600 primeraka

Štampa:
Birograf, Beograd

ISBN: 978-86-7912-817-1

Copyright:
© 2025. Univerzitet Singidunum
Izdavač zadržava sva prava.
Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

SADRŽAJ

I DEO
OSNOVNI POJMOVI VEZANI ZA SISTEME ZA RAD U REALNOM VREMENU

1. Definicija Sistema za rad u realnom vremenu 9
2. Definicija ugrađenih (embedded) sistema 10
3. Primarne karakteristike sistema za rad u realnom vremenu 11
4. Tipična struktura sistema za rad u realnom vremenu 12
5. Oblasti primene sistema za rad u realnom vremenu 14
6. Tipovi zadataka koji se javljaju kod sistema za rad u realnom vremenu 15
7. Tipovi vremenskih ograničenja i podela sistemima za rad u realnom vremenu 16
8. Izazovi pri projektovanju sistema za rad u realnom vremenu 18
9. Ograničenja pri projektovanju sistema za rad u realnom vremenu 20
10. Osobine koje sistemi za rad u realnom vremenu treba da poseduju 21
11. Eliminacija grešaka u sistemima za rad u realnom vremenu 23
12. Ciklus razvoja softvera za sisteme za rad u realnom vremenu 25
13. Česte zablude o sistemima za rad u realnom vremenu 28

II DEO
OSNOVE PROJEKTOVANJA I IMPLEMENTACIJE SISTEMA ZA RAD U REALNOM VREMENU

1. Merenje vremena 33
2. Prekidi (interrupts) 35
3. Konačni automati 46
4. Podela poslova na programske zadatke 48
5. Raspoređivanje programskih zadataka 49
6. Sinhronizacija i komunikacija među programskim zadacima 54
7. Modelovanje sistema za rad u realnom vremenu 57

III DEO
PRAKTIČNI PRIMERI

1. Treptanje svih dioda 67
2. Kontrola diode preko dugmeta 69
3. Kontrola diode preko dugmeta sa prekidom 71

4. Eksterna dioda 73
5. Treptanje diode – kašnjenje 74
6. Treptanje diode – sa tajmerom 75
7. Semafor 77
8. Naizmenično paljenje dioda na dva porta 82
9. Naizmenično paljenje dioda na dva porta sa treptanjem 84
10. Inkrementalno paljenje kolona dioda 87
11. Inkrementalno paljenje kolona dioda na jednom portu koristeći dugme 90
12. LED Chaser – Knight Rider 92
13. Kontrola diode preko PWM signala 94
14. Analogno čitanje vrednosti koristeći potenciometar P1 97
15. LCD ispis 99
16. Očitavanje temperature sa LM35 i ispis na LCD 101
17. Osnova snake igrice 104
18. Simon Says 109
19. Igra hvatanja 115
20. 7-segmentni displej 120
21. GLCD crtanje 123
22. USB Uart – Hello World 128
23. USB Uart – Prijem i slanje poruka 130
24. USB Uart – ispisivanje poruke na LCD displej 132
25. Buzzer 134
26. Igra odbrojavanja 137
27. Bitska štoperica 140
28. Promena frekvencije treptanja 142
29. Štoperica na 7-segmentnom displeju 145
30. PIN kod na 7-segmentnom displeju 149
31. Varijabilni prekid 153
32. Bluetooth komunikacija sa mobilnim telefonom 156
33. LCD ispis preko Bluetooth-a 158
34. Kretanje karaktera na LCD displeju putem Bluetooth-a 160
35. Upravljanje LED rasvetom putem Bluetooth-a 163

LITERATURA 168
BIOGRAFIJE AUTORA 170

I DEO

OSNOVNI POJMOVI VEZANI ZA SISTEME ZA RAD U REALNOM VREMENU

1. DEFINICIJA SISTEMA ZA RAD U REALNOM VREMENU

Gotovo svi sistemi koji nas okružuju menjaju se sa vremenom. Za neke od tih vremenskih promena sami smo odgovorni kroz naše aktivnosti, dok su druge izvan naše kontrole. U ovom kursu razmatraćemo samo sisteme koji su upravljani računskim sistemom u obliku programa.

Računari (najčešće u obliku mikrokontrolera) se danas ugrađuju u gotove sve uređaje sa kojima se čovek u svakodnevnom životu susreće. Ovakvi ugrađeni računari kontrolišu rad uređaja i omogućavaju njihovo jednostavnije korišćenje od strane ljudi. U ovom kursu, centar naše pažnje su najviše ugrađeni sistemi koji rade u realnom vremenu.

Sistemi koji rade u realnom vremenu (Real Time Systems) su sistemi čija **ispravnost** zavisi ne samo od **logičkih rezultata izračunavanja**, već i od **vremena** u kome su ti rezultati proizvedeni.

Pod ispravnošću sistema se podrazumeva da sistem radi tačno ono što se od njega očekuje uz prihvatljiv kvalitet vezan za rezultate njegovog funkcionisanja.

Pod logičkom ispravnošću rezultata se podrazumeva da računar, odnosno program koji se na njemu izvršava daju očekivane izlazne rezultate za date ulaze. U principu svi programi moraju biti logički ispravni (u protivnom ne bi ni bili od velike koristi). Međutim nekada (kao što je to slučaj sa sistemima za rad u realnom vremenu) samo logička ispravnost rezultata nije dovoljna.

Kod sistema za rad u realnom vremenu, samo logička tačnost rezultata nije garant da će sistem raditi ispravno. Uz logičku tačnost, potrebno je da rezultati budu proizvedeni u pravom trenutku. Kod ovih sistema vremenska neusklađenost rezultata usled kašnjenja izračunavanja na primer, može značajno smanjiti performanse čitavog sistema ili čak dovesti do njegovog oštećenja, pa i uništenja.

Tipični primeri računarskih sistema kod kojih je važna logička tačnost rezultata, ali ne i vremenska tačnost (dakle oni nisu sistemi za rad u realnom vremenu) su personalni računar ili džepni kalkulator. Pri izračunavanju pomoću džepnog kalkulatora od presudne je važnosti da rezultat izračunavanja bude tačan, međutim tačno vreme u kome će se rezultat pojaviti na displeju džepnog kalkulatora nije toliko važno i obično se od strane korisnika percipira kao da rezultati stižu trenutno. Slično, kod personalnih računara i korišćenja programa kao što je editor teksta. Za korisnika je bitno da program radi ono što se od njega očekuje, to jest da kada korisnik otkuca neko slovo na tastaturi, da se to isto slovo pojavi i na monitoru. Vreme koje protekne od kucanja na tastaturi do pojavljivanja slova na ekranu nije presudno i obično ga korisnik vidi kao trenutno. Ukoliko to nije slučaj i postoji nekakvo kašnjenje, korisnik će smatrati da njegov računar nije dovoljno brz, ali ne da i neispravno radi, jer on iako je spor, radi tačno ono što se od njega i očekuje.

Sa druge strane sistemi kao što su autopilot u avionu ili sistem protiv blokade kočnica (ABS) u automobilu su sistemi za rad u realnom vremenu. Za ove sisteme nije dovoljno da mikrokontroler koji izračunava vrednosti upravljačkih veličina koje se primenuju na aktuatore (u slučaju autopilota to su motori koji zakreću delove krila, a kod ABS-a to je pumpa koja reguliše pritisak u kočionom sistemu) daje logički ispravne rezultate. Potrebno je i da ti rezultati stignu na vreme kako bi određeni upravljački manevri mogli da se izvrše uz pomoć aktuatora. Dakle rezultati moraju stići ni suviše brzo ni suviše kasno, jer u suprotnom upravljački algoritam neće raditi ispravno, što može dovesti do posledica koje mogu biti i katastrofalne.

2. DEFINICIJA UGRAĐENIH (EMBEDDED) SISTEMA

Ugrađeni (embedded) sistem je računarski sistem sa **specifičnom funkcionalnošću u okviru većeg** mehaničkog ili električnog sistems, često **sa realnim vremenskim ograničenjima**.

Ugrađeni sistem je samo jedan od delova mnogo većeg mehaničkog ili električnog sistema. Najčešće zapravo ugrađeni sistemi predstavljaju srce takvih sistema, jer vrše njihovo upravljanje i omogućavaju interfejs sa operaterom. Računarski sistemi koji se koriste kao ugrađeni najčešće su veoma specifični. Za razliku od klasičnih personalnih računara, ovi uređaji imaju mnogo skromnije procesorske i memorijske resurse. Međutim, oni imaju mnogo više različitih izlaznih i ulaznih portova koji im služe za interakciju sa ostatkom sistema u koji su ugrađeni.

Većina ugrađenih sistema su zapravo sistemi za rad u realnom vrenu. Međutim, pojam ugrađenih sistema i sistema za rad u realnom vremenu nisu sinonimi, jer postoji jedan veoma mali broj ugrađenih sistema koji ne rade u realnom vremenu, a postoji i jedna malo veća grupa sistema za rad u realnom vremenu koji nisu ugrađeni sistemi. Za potrebe ovog kursa uglavnom ćemo razmatrati sisteme za rad u realnom vremenu koji su istovremeno i ugrađeni sistemi.

3. PRIMARNE KARAKTERISTIKE SISTEMA ZA RAD U REALNOM VREMENU

Kao što je već rečeno, u sistemima za rad u realnom vremenu često imamo prisutna vremenska ograničenja. Kod nekih sistema (kao što je autopilotski sistem na primer) neispunjavanje ovih vremenskih ograničenja može dovesti do veoma ozbiljnih posledica. Čest je slučaj da u sistemima za rad u realnom vremenu imamo izračunavanja koja se ponavljaju sa tačno određenom vremenskom periodom.

Sistemi za rad u realnom vremenu najčešće imaju specifičnu namenu, te su uglavnom oni istovremeno i ugrađeni (embedded) sistemi. Ovi sistemi vrlo često zapravo imaju upravljačku ulogu u većem mehaničkom ili električnom sistemu. Zapravo, može se reći da su implementirani sistemi automatskog upravljanja uglavnom sistemi za rad u realnom vremenu.

Od sistema za rad u realnom vremenu često se traži visok stepen pouzdanosti. To znači da oni najčešće moraju da rade bez greške, ali i da budu u mogućnosti da pravilno odreaguju na greške koje se javljaju u sistemu koji ih okružuje. Zbog toga se u dizajniranju sistema za rad u realnom vremenu dosta vodi računa o upravljanju greškama i nepredviđenih situacija, a izbacivanju računarskih sistema za rad u realnom vremenu na tržište često prethodi formalno i opsežno testiranje sistema.

Sistemi za rad u realnom vremenu često sadrže i nekakav korisnički interfejs koji omogućava operateru da prati stanje sistema i da zadaje upravljačke zadatke sistemu. Umesto operatera ova komunikacija može biti i sa drugim računarskim sistemima koji vrše superviziju ili strateško planiranje. U svakom slučaju mnoge od ulaznih informacija (kao što su ulazi od strane operatera ili pojedini signali iz okruženja u kome se sistem nalazi) dolaze asinhrono i teško ih je predvideti.

Sistemi za rad u realnom vremenu su često dosta složeni i između njihovih delova kako hardverskih, tako i softverskih postoji velika povezanost. Delovi softvera u sistemima za rad u realnom vremenu (različiti programski zadaci) često zavise jedni od drugih, jer dele ista memorijska mesta, podatke i hardverske resurse.

Od sistema za rad u realnom vremenu se često traži veoma brz odziv na spoljašnje stimulanse i informacije.

4. TIPIČNA STRUKTURA SISTEMA ZA RAD U REALNOM VREMENU

Sistemi za rad u realnom vremenu se sastoje od hardverske platforme na kojoj se izvršava aplikativni softver. Računarski sistem za rad u realnom vremenu komunicira sa svojom okolinom, koja je najčešće sistem kojim sistem za rad u realnom vremenu upravlja. Interakcija sa okolinom odvija se preko senzora i aktuatora.

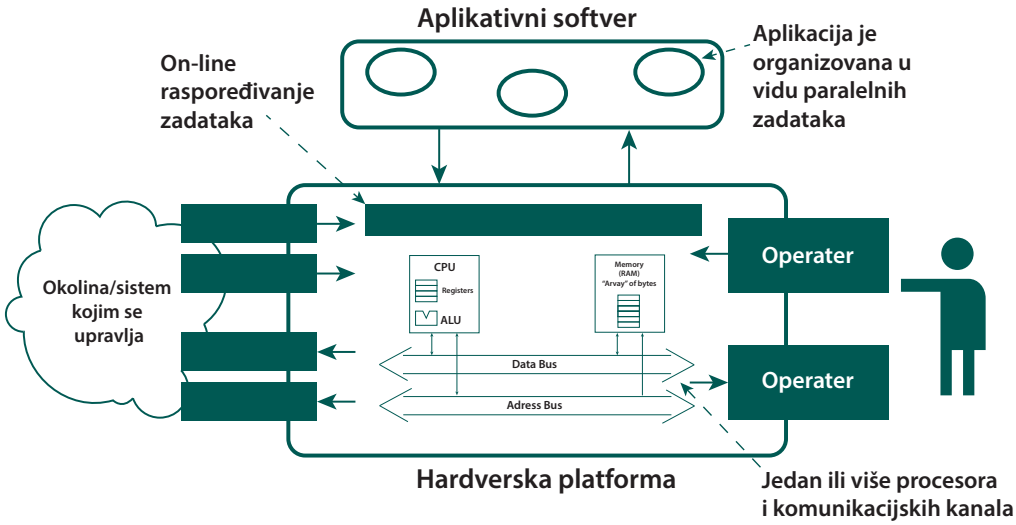
Senzori služe za merenje različitih fizičkih veličina u sistemu kojim se upravlja (npr. senzori temperature, pritiska i slično). Merena fizička veličina se uz pomoć senzora pretvara u električni napon, koji se zatim uz pomoć analogno digitalnog konvertora pretvara u broj pohranjen u memoriji hardverske platforme, a koji aplikativni softver može da koristi za svoja izračunavanja.

Izračunati upravljački signali, koje izračunava aplikativni softver i koji su zapravo brojevi u memoriji hardverske platforme se uz pomoć A/D konvertora pretvaraju u naponske signale koji zatim pobuđuju aktuatore. Aktuatori su najčešće elektro motori, pumpe, ventili, grejači i slični uređaji koji mogu svojom aktivnošću da menjaju okolinu u kojoj se nalaze.

Sistemi za rad u realnom vremenu koji treba da komuniciraju sa operaterom najčešće sadrže i periferne uređaje kao što su operaterski displeji i paneli koji su mnogo jednostavniji od standardnih perifernih uređaja u personalnim računarima kao što su monitor, miš i tastatura. U slučaju da sistem ne komunicira sa operaterom, već sa drugim računarskim sistemom koji deluje kao master, sistem za rad u realnom vremenu sadrži odgovarajući hardver neophodan za takvu vrstu komunikacije.

Aplikativni softver kod sistema za rad u realnom vremenu je najčešće organizovan u vidu programskih zadataka koji se paralelno izvršavaju. Taj paralelizam može biti stvarni u smislu da hardverska platforma ima više procesora. Međutim, on je mnogo češće prividan, jer najčešće hardverska platforma ima samo jedan procesor, a zadaci se izvršavaju naizmenično, tako što se procesorsko vreme naizmenično dodeljuje svakom od programskih zadataka, što daje privid paralelnosti. Zbog ovakvog načina izvršavanja aplikativnog softvera, neophodno je da postoji mehanizam koji će raspoređivati procesorsko vreme i dodeljivati ga različitim zadacima. Pri tome, neophodno je da različiti zadaci budu dobro sinhronizovani u smislu da ne pristupaju memoriji ili drugim hardverskim komponentama u isto vreme. Podršku za ovakav paralelni rad aplikativnog softvera pruža operativni sistem. Za sisteme koji rade u realnom vremenu potrebni su operativni sistemi koji su drugačiji od klasičnih operativnih sistema koji se koriste u personalnim računarima i nazivaju se operativnim sistemima u realnom vremenu (Real Time OS).

Slika 1: Tipična struktura sistema za rad u realnom vremenu

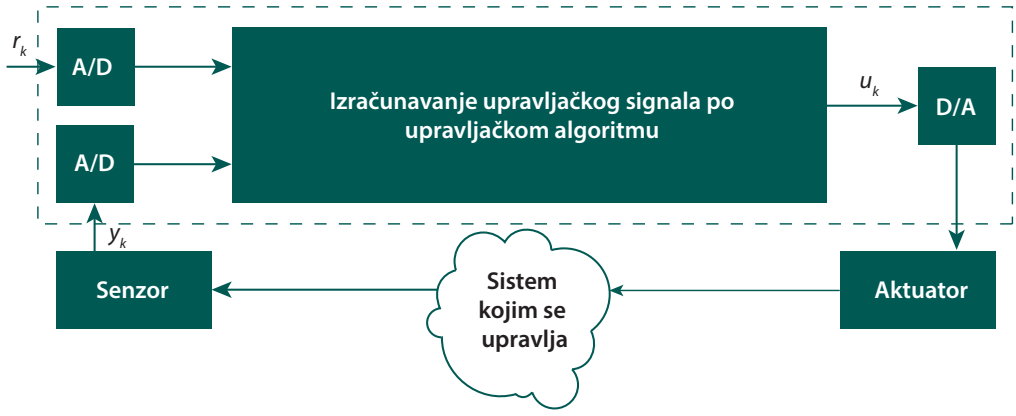


Jednostavan primer sistema za rad u realnom vremenu – sistem za automatsko upravljanje

Kao jednostavan primer sistema za rad u realnom vremenu možemo uzeti standardni sistem automatskog upravljanja koji ima jedan senzor i jedan aktuator. Sistem još kao ulaznu veličinu uzima analognu referentnu vrednost merene fizičke veličine. Aplikativni softver koji bi se koristio u ovakvom jednostavnom slučaju bi se satojao od jednog zadatka, a to je izračunavanje upravljačke veličine u_k na bazi merenja y_k i referentne vrednosti r_k. Ovo izračunavanje bi se ponavljalo periodično sa izabranom upravljačkom periodom.

Sistemi za rad u realnom vremenu koji se sreću u praksi obično su dosta složeniji od onoga što je dato u ovom jednostavnom primeru. Najčešće oni imaju mnogo veći broj ulaza i izlaza, jer sistem ima više od jednog senzora i aktuatora. Obično je u takvim situacijama slučaj da postoji mogućnost izbora između različitih algoritama upravljanja. Industrijski kontroleri takođe često imaju periferne uređaje za komunikaciju sa operaterom ili mehanizam za komunikaciju sa drugim ugrađenim sistemima.

Slika 2: Sistem automatskog upravljanja kao jednostavni sistem za rad u realnom vremenu



5. OBLASTI PRIMENE SISTEMA ZA RAD U REALNOM VREMENU

Sa ubrzanim razvojem računarskih sistema u poslednje vreme, ugrađeni računarski sistemi postaju sve prisutniji u našem svakodnevnom životu. Tako da možemo reći da su danas ugrađeni i sistemi za rad u realnom vremenu svuda oko nas. Neke od oblasti primene sistema za rad u realnom vremenu su sledeće :

- **U energetici,** za upravljanje kako hidro, termo i nuklearnim elektranama, tako i za upravljanje obnovljivim izvorima energije, ali i za upravljanje sistemima za distribuciju električne enerhije.
- **U teškoj industriji**, za upravljanje velikim hemijskim postrojenjima kao što su rafinerije nafte na primer, ali i u industriji gasa, kao i u čeličanama, cementarama i ostalim velikim fabričkim postrojenjima.
- **U proizvodnim pogonima**, kako za upravljanje i integraciju procesa na proizvodnim trakama, tako i za monitoring kvaliteta finalnih proizvoda.
- **U auto industriji**, za sistema protiv blokade kočnica (ABS), za sisteme za upravljanje motorom kako bi se smanjila njegova potrošnja i optimizovao rad. Za sisteme za detekciju prepreka i automatsko parkiranje, kao i za sisteme za automatsku vožnju automobila koji su danas predmet naučnih istraživanja, ali polako nalaze put i do komercijalno dostupnih automobila.

- **U avio industriji**, za autopilotske sisteme, kao i za sisteme monitoringa stanja letelice u toku leta, ali i za složene sisteme koji se koriste za planiranje ruta avio saobraćaja u kontroli letenja.
- **U kućnim aparatima**, kao upravljačke jedinice za moderne (pametne) kućne uređaje koji se sve više koriste. Primeri takvih uređaja su štedljive veš mašine, autonomni usisivači (robotski usisivači koji samostalno idu po kući i usisavaju) kao i mnogi drugi. U ovu grupu spadaju i sistemi za objedinjeno upravljanje kućnim uređajima i aparatima (smart home aplikacije).
- **U robotici**, kao upravljačke jedinice za robotske ruke koje se koriste u industriji, ali i kao upravljačke jedinice za sve prisutnije humanoidne robote.
- **U vojnoj industriji**, za prepoznavanje objekata, vođenje letelica i projektila kao i za prikupljanje i obradu obaveštajnih informacija.
- **U medicini**, za upravljanje složenim medicinskim uređajima kao i za monitoring stanja pacijenta i blagovremeno alarmiranje medicinskog osoblja. Sve su češći i primeri sistema za rad u realnom vremenu koji upravljaju radom veštačkih organa kao što je na primer veštački pankreas.
- **U telekomunikacijama**, za upravljanje velikim distribuiranim telekomunikacijskim sistemima i njihovim delovima.

Treba imati u vidu da je navedena lista daleko od kompletne liste svih oblasti u kojima se mogu sresti sistemi za rad u realnom vremenu.

6. TIPOVI ZADATAKA KOJI SE JAVLJAJU KOD SISTEMA ZA RAD U REALNOM VREMENU

Zadaci koji se javljaju u okviru sistema za rad u realnom mogu se prema periodičnosti podeliti na:

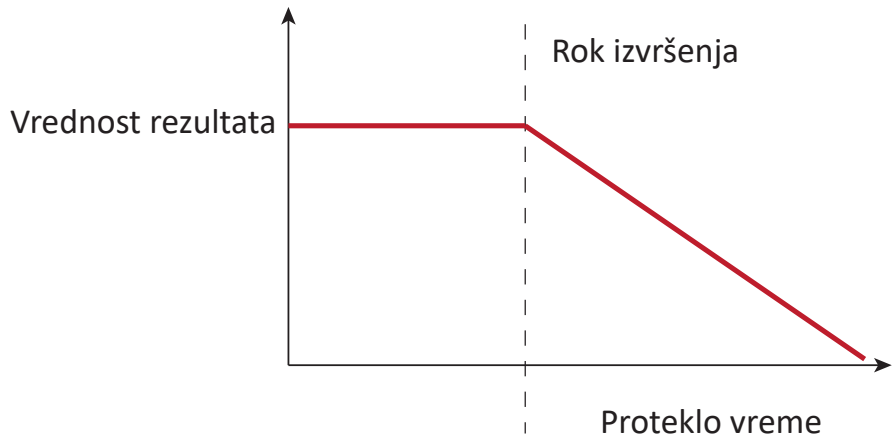
- **Periodične**, koji su vođeni vremenom i čije izvršavanje se ponavlja periodično sa datom periodom. Tipičan primer je zadatak periodičnog očitavanja vrednosti senzora temperature u aplikaciji pametne zgrade ili zadatak periodičnog izračunavanja upravljačke veličine kod kontrolera.
- **Aperiodične**, koji su vođeni događajima i kod kojih se trenutak zahteva za izvršavanjem ne može predvideti. Tipičan primer je zadatak koji se pokreće kada operater pritisne određeni taster na komunikacijskom displeju. Vreme početka izvršenja ovog zadatka ne može se predvideti, jer zavisi isključivo od volje operatera.
- **Sporadične**, koji su zapravo aperiodični zadaci sa poznatim minimalnim vremenom ponovnog javljanja.

7. TIPOVI VREMENSKIH OGRANIČENJA I PODELA SISTEMIMA ZA RAD U REALNOM VREMENU

Vremenska ograničenja koja se javljaju u kontekstu dizajniranja sistema za rad u realnom vremenu mogu se podeliti prema strogosti na:

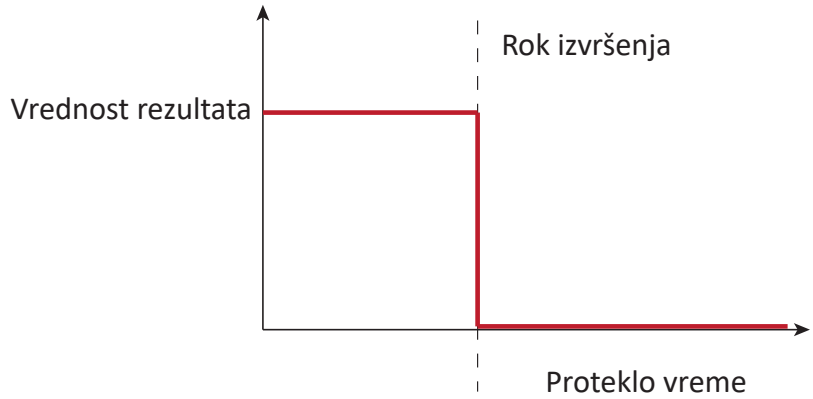
- **Meka (soft) ograničenja**, kod kojih logički tačan rezultat izračunavanja koji pristigne posle nekog zadatog vremenskog roka ima određenu kvalitativnu vrednost koja će biti manja u odnosu na situaciju da je rezultat stigao u okviru zadatih rokova, ali će postojati (biće veća od nule). Primer takvog ograničenja je recimo u algoritmu za kontrolu pametnih zgrada, gde određena kašnjenja u sistemu regulacije temperature nisu kritična i činjenica da sistem ipak uspeva da reguliše temperaturu u zgradi čak i sa zakašnjenjem je bolja nego da sistem uopšte nije u stanju da snizi temperaturu.

Slika 3: Vrednost rezultata sa protokom vremena kod mekih (soft) vremenskih ograničenja



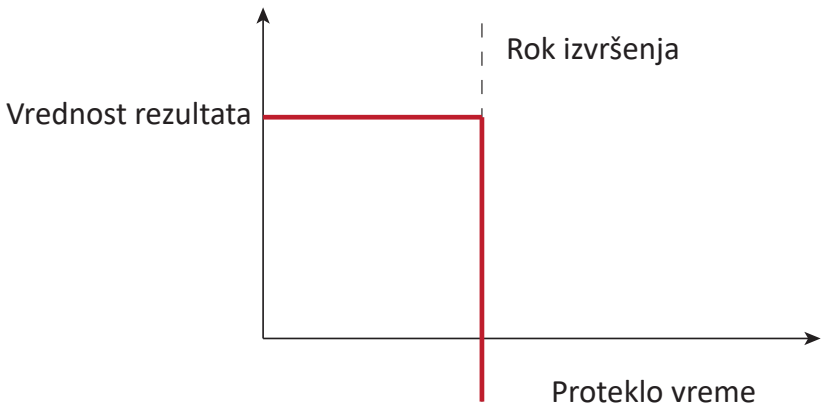
- **Čvrsta (firm) ograničenja**, su ograničenja kod kojih rezultat operacije pristigao posle vremenskog roka nema nikakvu vrednost (njegova vrednost je jednaka nuli). Primer takvog vremenskog ograničenja je ograničenje pri prijemu poziva lifta. Ukoliko sistem ne registruje poziv lifta na određenom spratu na vreme, lift će proći taj sprat i osoba koja ga je pozvala će ostati da čeka. U tom slučaju činjenica da je sistem ipak registrovao poziv posle izvesnog zakašnjenja, osobi koja ostaje da čeka neće biti velika uteha.

Slika 4: Vrednost rezultata sa protokom vremena kod čvrstih (firm) vremenskih ograničenja



- **Stroga (hard) ograničenja** su ograničenja kod kojih ukoliko rezultat ne stigne pre zadatog roka, može doći do oštećenja sistema ili čak do njegovog uništenja i fatalnih posledica. Primer takvih ograničenja su ograničenja u sistemu autopilota. Ukoliko sistem ne izda određene upravljačke komande na vreme, može doći do gubitka kontrole aviona, pa čak i do njegovog pada.

Slika 5: Vrednost rezultata sa protokom vremena kod strogih (hard) ograničenja



Prema tipovima vremenskih ograničenja koja sadrže, sistemi za rad u realnom vremenu mogu se podeliti na meke (soft), čvrste (firm) i stroge (hard). S obzirom na njihovu kompleksnost, sistemi za rad u realnom vremenu često imaju veći broj vremenskih ograničenja svih tipova. Kategorizacija sistema se u tom slučaju vrši prema najstrožijem vremenskom ograničenju.

8. IZAZOVI PRI PROJEKTOVANJU SISTEMA ZA RAD U REALNOM VREMENU

S obzirom da su često dosta složeni i da moraju da ispune striktne vremenske zahteve i okvire, projektovanje sistema za rad u realnom vremenu često sa sobom nosi mnoge izazove.

Projektovanje ovih sistema podrazumeva upravljanje hardverskim sistemima kao što su linije za komunikaciju sa drugim uređajima, različiti industrijski terminali kao i računarskim resursima, koji su kod ugrađenih sistema dosta ograničeni.

Pored toga, kao što je već rečeno, sistemi za rad u realnom vremenu moraju biti u stanju da reaguju na događaje koji su aperiodični i nepredvidivi. To zapravo znači da u komunikaciji sa drugim sistemima, kao i sa svojom okolinom, sistemi za rad u realnom vremenu mogu dobijati poruke u neočekivanim vremenskim trenucima i na njih moraju da reaguju.

Pored reakcija na poruke iz okruženja, sistemi za rad u realnom vremenu moraju da budu otporni i na greške i nepredviđene promene u svom okruženju. Rad računarskih sistema koji rade u realnom vremenu mora biti takav da ne samo da oni ne budu uzročnici grešaka i problema u funkcionisanju sistema, već moraju imati programske mehanizme da eliminišu ili barem ublaže posledice grešaka koje nastaju drugde u sistemu i koje ukoliko se ne tretiraju od strane upravljačkog računarskog sistema mogu dovesti do zaustavljanja rada sistema ili čak do njegovog oštećenja.

Sistemi za rad u realnom vremenu često imaju veliki broj izlaza i ulaza i moraju da upravljaju velikim brojem izlaznih i ulaznih podataka. To podrazumeva programsko upravljanje redovima čekanja i baferima podataka.

Pošto se aplikativni softver najčešće sastoji od većeg broja programskih zadataka, poželjno je da se ovi zadaci izvršavaju paralelno kako bi se maksimalno uposlili svi hardverski resursi sistema (koji su često ograničeni). Zbog toga kod sistema sa više procesora, mora da postoji sistem za optimalno raspoređivanje procesora zadacima. Kod sistema sa samo jednim procesorom, teži se prividnoj paralelizaciji. Ona podrazumeva da se procesor dodeljuje jednom zadatku određeno vreme, onda se on prekida, pa se procesor dodeljuje sledećem zadatku i tako u krug, čime se ostvaruje privid kao da se svi zadaci izvršavaju paraleno. Ovakav način izvršavanja programskih zadataka dovodi do boljeg iskorišćenja dostupnih hardverskih resursa. Međutim, ovakav postupak prividne paralelizacije zahteva rešavanje niza dodatnih izazova. Potrebno je da postoji mehanizam koji će čuvati kontekst programskog zadatka kada se on prekida kako bi on po ponovnoj dodeli procesora mogao da nastavi svoja izračunavanja tamo gde je prekinut.

Promena konteksta podrazumeva da se svi relevantni rezultati u trenutku prekida sačuvaju u odgovarajuće memorijske lokacije i da se po ponovnom aktiviranju zadatka mogu koristiti. Takođe mora se obezbediti da se komunikacija između zadataka ne naruši i da se izvršavanje zadataka sinhronizuje. Kao primer uzmimo slučaj dva programska zadatka od kojih jedan upisuje neku vrednost u datu promenljivu, a drugi programski zadatak je čita da bi je koristio za dalja izračunavanja. U ovakvom slučaju mora se sprečiti da prvi programski zadatak bude prekinut u toku upisa vrednosti u datu memorijsku lokaciju, jer se onda može desiti da drugi programski zadatak očita pogrešnu vrednost. Takođe, način na koji se određuje kojem zadatku će kada biti dodeljen procesor mora da uzme u obzir činjenicu da u sistemima za rad u realnom vremenu postoje određena čvrsta vremenska ograničenja i da pojedini zadaci moraju da se izvrše u određenim vremenskim okvirima. Zbog toga je u sistemima za rad u realnom vremenu često potrebno da se dodeljivanje prioriteta zadacima vrši dinamički na bazi proteklog vremena.

U sistemima za rad u realnom vremenu, merenje vremena je veoma važno. Zbog toga je potrebno da postoje pouzdani mehanizmi za komunikaciju sa hardverskim časovnikom realnog vremena, kako bi se obezbedilo precizno merenje proteklog vremena koje je neophodno u donošenju odluka kada će se koji programski zadatak izvršavati.

Prema svemu do sada rečenom, dizajniranje sistema za rad u realnom vremenu je dosta izazovno i softver kod ovakvih sistema je dosta složen, a često i sam kod može biti veoma dugačak. Povrh toga, pre puštanja u rad ovakvih sistema potrebno je dobro testirati njihovu funkcionalnost i proveriti da li rade kako treba. Samo pronalaženje grešaka u kodu (bagova) kod ovako složenih sistema, koji nekada mogu biti i distribuirani (podeljeni na više hardverskih platformi) može biti veoma izazovno. Način da se bori sa kompleksnošću kod ovakvih sistema je podela na manje i jednostavnije funkcionalne celine kojima se onda bave posebni timovi programera.

Često je slučaj da u trenutku razvoja koda, nisu dostupni svi delovi hardverskog sistema, pa se, da bi se ubrzao proces razvoja, u ovoj fazi koriste emulatori hardverskih komponenti, što može dodatno da komplikuje sam proces dizajniranja sistema.

Uzevši sve ovo u obzir, možemo reći da proces dizajniranja i implementacije sistema za rad u realnom vremenu nije lak i da ga prate mnogi izazovi. Međutim ti izazovi nisu nerešivi i pravilnim pristupom, svaki od njih se može prevazići, a kao krajnji rezultat dobiti sistem za rad u realnom vremenu koji funkcioniše u skladu sa željenim specifikacijama.

9. OGRANIČENJA PRI PROJEKTOVANJU SISTEMA ZA RAD U REALNOM VREMENU

Pri projektovanju sistema za rad u realnom vremenu, uvek treba voditi računa o sledećim ograničenjima koja se javljaju :

- **Vremenska ograničenja**. Ova ograničenja nalaze se u samoj osnovi sistema za rad u realnom vremenu. Kako je jedan od glavnih ciljeva projektovanja ovih sistema obezbeđivanje da rezultati izračunavanja budu ne samo logički ispravni, već i da stignu na vreme, o ovom aspektu se mora voditi poseban račun u toku dizajniranja sistema.
- **Ograničenja pri upotrebi resursa**. Resursi kod ugrađenih sistema su često oskudni i sistem poseduje samo one resurse koji su mu neophodni za rad. Pri korišćenju tako oskudnih računarskih resursa mora se voditi računa o načinu na koji se resursi dodeljuju različitim programskim zadacima.
- **Ograničenja isključenja**. U okviru programskih zadataka mogu postojati kritični delovi koda koji se ni u kom slučaju ne smeju prekidati da bi se izvršavali drugi procesi. O ovome treba voditi računa pri pisanju koda za sisteme za rad u realnom vremenu.
- **Odnosi prvenstva**. Može se desiti da programski zadaci budu takvi da se izvršavanje jednog zadatka ne može (ili čak ne sme) početi pre nego što svoj rad ne završi neki drugi zadatak. O ovome se mora voditi računa u procesu dodele procesora različitim zadacima.
- **Ograničenja paralelnosti**. Kako sistemi za rad u realnom vremenu često moraju imati brze odzive na pobude iz okoline i kako su raspoloživi hardverski resursi oskudni, radi povećanja vremena odziva i bolje uposlenosti resursa, zahteva se da se različiti programski zadaci paralelno izvršavaju, čak i u situacijama kada hardverska platforma ima samo jedan procesor (kvazi paralelnost).
- **Zahtevi za komunikacijom među zadacima**. Programski zadaci često moraju da komuniciraju jedni sa drugima i čest je slučaj da jedan programski zadatak koristi rezultate izračunavanja drugog programskog zadatka kao svoje ulaze. O potrebi za komunikacijom mora se voditi računa pri ostvarenju paralelizma i kvazi paralelizma.
- **Zahtevi za tolerisanjem otkaza i grešaka**. Sistemi za rad u realnom vremenu moraju biti takvi da mogu da se nose sa nepredviđenim situacijama i greškama u sistemu i njegovoj okolini.
- **Različiti stepeni kritičnosti**. Često se aplikativni softver sastoji od različitih zadataka koji su vrememenski kritični i onih koji to nisu. Pri projektovanju sistema mora se voditi računa da se uvek veći prioritet da izvršavanju zadataka sa većim stepenom vremenske kritičnosti.

10. OSOBINE KOJE SISTEMI ZA RAD U REALNOM VREMENU TREBA DA POSEDUJU

Eventualni problemi i greške nastale u toku operacije sistema za rad u realnom vremenu mogu dovesti do ozbiljnih posledica. Kao ilustrativan primer možemo uzeti problem u sistemu za upravljanje roverom poslatim na Mars u okviru patfinder misije. Vozilo se uspešno prizemljilo na mars i počelo svoju misiju prikupljanja podataka o crvenoj planeti. Međutim posle samo nekoliko dana, sistem je počeo čudno da se ponaša. Naime dolazilo je do čestih reseta računarskog sistema usled čega su se prikupljeni podaci nepovratno gubili, tako da je vozilo čiji je razvoj koštao milione dolara postalo praktično beskorisno. Teško je i zamisliti kako su se u takvoj situaciji osećali ljudi koji su radili na razvoju sistema za rad u realnom vremenu koji je zakazao. Srećom, oni su uspeli da stvar poprave, menjajući sadržaj pojedinih registara u računaru specijalnom vezom komunikacije sa Zemlje i misija je uspešno nastavljena. Međutim nemaju svi slučajevi otkaza u sistemima za rad u realnom vremenu srećan kraj. U nekim slučajevima može doći i do gubitka života. Tipični primeri takvih situacija su greške pri radu auto pilota ili sistema za automatsku vožnju automobila (auto-cruise).

Zbog svega ovoga, sistemi za rad u realnom vremenu treba da poseduju sledeće osobine:

Predvidivost

Sistem mora biti projektovan tako da se tačno zna kako određena akcija koja se pokreće utiče na sistem u celini. Takođe, vreme reakcije na moguće događaje u sistemu mora biti unapred poznato. Ovo omogućava da se ponašanje u različitim situacijam sistema može sa preciznošću i unapred predvideti, što sistem čini predvidivim.

Pouzdanost

Sistem mora biti dizajniran tako da se greške ne vide spolja. One praktično ostaju maskirane, ali se pamte radi njihove kasnije analize i eventualnog sprečavanja i otklanjanja. Na ovaj način korisnik sistema praktično i ne zna da je u sistemu došlo do greške i zato je za njega sistem pouzdan (nije sklon greškama). Sa druge strane sistem je svestan grešaka i daje mogućnost da se one analiziraju i isprave.

Robusnost

Sistem mora da bude projektovan tako da ostane imun na promene okoline ili neke nepredviđene situacije i otkaze. Često je slučaj da operater uradi nešto nepredviđeno ili pogrešno, a da sistem mora i u takvoj situaciji da reaguje ispravno. Ovo se često dešava kod bele tehnike, gde korisnici često (pogrešno) koriste uređaje i bez čitanja tehničkih uputstava, a sistem i dalje ispravno funkcioniše, jer je projektovan tako da bude otporan na neke nepredviđene situacije.

Robusnost u odnosu na ozbiljnije otkaze u sistemu može se ostvariti uvođenjem hardverske i softverske redundanse, gde sistem ima duplirane pojedine hardverske delove da bi mogao da nastavi normalan rad u slučaju otkaza nekih od njih. U takvim situacijama duplirane softverske komponente obično koriste drugačije algoritme izračunavanja, što povećava bezbednost čitavog sistema.

Modularnost

Sistemi za rad u realnom vremenu su pre svega komercijalni sistemi koji se pojedinačno ili kao delovi većih i složenijih sistema prodaju na slobodnom tržištu. Usled toga, često se javlja težnja za njihovim menjanjem i unapređenjem kako bi bili konkurentniji. Da bi se olakšalo stalno unapređenje ovih sistema, poželjno je da oni budu projektovani modularno, tako da pri potrebi za nekom promenom, ne mora da se radi redizajniranje čitavog sistema, već samo da se zameni manji modul na koga se izmene i odnose.

Postoji niz načina i dobrih praksi uz koje mogu da se ostvare ove željene osobine koje sistemi za rad u realnom vremenu treba da imaju. Ove tehnike uključuju korišćenje gotovih (off the shelf) komponenti u najvećoj mogućoj meri. U procesu dizajniranja uvek treba tražiti rešenja za zadati problem, a ne problem prilagođavati nekom rešenju koje je lako/brzo ostvariti. Otpornost sistema na greške i njegova funkcionalnost treba da budu viđeni kao celina u postupku dizajniranja. Za zaštitu od grešaka treba koristiti standardne i verifikovane metode. Kod treba da bude takav da njegove osobine u smislu vremena izvršavanja, pristupa memoriji i ostalim resursima budu predvidive. U cilju što efikasnijeg raspoređivanja procesora paralelnim programskim zadacima, treba koristiti usluge operativnog sistema za rad u realnom vremenu. Takođe treba koristiti alate koji pomažu u izarčunavanju vremena potrebnog da se određeni zadaci izvrše.

11. ELIMINACIJA GREŠAKA U SISTEMIMA ZA RAD U REALNOM VREMENU

Za sistem za rad u relanom vremenu kažemo da je otporan na greške ukoliko može da nastavi sa obavljanjem svojih specifičnih zadataka i u prisustvu hardverskih i softverskih grešaka/otkaza. Sistemi za rad u realnom vremenu treba da budu otporni na greške pre svega da bi zaštitili ljudske živote u situacijama u kojima oni zavise od ispravnosti rada sistema (autopilot, car cruise control). Takođe, kod sistema koji imaju neku važnu i kritičnu misiju, sistemi za rad u realnom vremenu moraju biti otporni na greške (primer sa mars rover misijom). Kao što je već pomenuto u uređajima koji se masovno koriste, moraju biti otporni na pogrešne akcije korisnika. Ponekad sistemi za rad u realnom vremenu rade u teškim uslovima (industrija nafte i gasa na primer) u kojima su otkazi hardverskih komponenti česti. I u takvim uslovima ovi sistemi moraju da funkcionišu da ne bi dolazilo do prečestog zaustavljanja postrojenja i ogromnog gubitka novca.

Greške u sistemima za rad u realnom vremenu možemo prema trajanju podeliti na:

- **Tranzijentne greške**, koje nastaju u sistemu u određenom trenutku, traju jedno izvesno vreme i onda nestaju. Tipičan primer tranzijentnih grešaka su greške u komunikaciji.
- **Permanentne greške**, koje ostaju u sistemu dok ne budu otklonjene. Tipičan primer je trajno uništena hardverska komponenta, kao što je prekinut kabl napajanja na primer.
- **Intermitentne greške**, koje su u suštini tranzijentne greške koje se javljaju sa vremena na vreme. Tipičan primer su hardverske komponente osetljive na temperaturu. Usled zagrevanja, one prestaju da rade, onda se hlade i u jednom trenutku počinju ponovo korektno da rade, sve dok se ponovo ne pregreju i greška se ponovo pojavi.

Prema uzroku nastajanja greške mogu biti:

- **Fizičke**, nastale usled otkaza neke od hardverskih komponenti.
- **Greške u dizajnu**, nastale usled pogreške u projektovanju sistema.
- **Greške operatera**, koje nastaju usled nepravilnog rukovanja sistemom.
- **Greške nastale usled drastičnih promena u okruženju** (primer nestanak napajanja).

Sistem se može načiniti otpornim na greške kroz prevenciju grešaka i tolerisanje grešaka.

Prevencija grešaka

Prevencija grešaka se vrši kroz izbegavanje i eliminaciju grešaka. Izbegavanje grešaka se postiže korišćenjem dobro oprobanih strategija u dizajnu sistema kojima se mogućnost pojave grešaka u fazi implementacije svodi na minimum. Eliminacija grešaka podrazumeva detaljno i formalizovano testiranje koda pre izbacivanja na tržište i otklanjanje svih mogućih uočenih grešaka

Tolerisanje grešaka

Tolerancija na greške podrazumeva dizajn sistema u kome će sistem nastaviti da funkcioniše u nekom obimu i posle pojave greške. Stepeni tolerantnosti na greške mogu biti.

- **Potpuna tolerantnost na greške** (full fault tolerance), kod koje sistem nastavlja da radi sa punom funkcionalnošću i usled prisustva greške, bar na neko određeno vreme.
- **Blaga degradacija** (fail soft), kod koje sistem nastavlja da radi sa smanjenom funkcionalnošću ili smanjenim performansama do otklanjanja greške.
- **Bezbedno gašenje** (fail safe), gde sistem sprečava svoje uništenje ili oštećenje usled pojave greške prihvatajući kontrolisan zastoj u radu (bezbedno gašenje).

12. CIKLUS RAZVOJA SOFTVERA ZA SISTEME ZA RAD U REALNOM VREMENU

Razvojni ciklus bilo kog softverskog rešenja sastoji se iz sledećih koraka:

1. Izrada specifikacije
2. Dizajn softverske arhitekture
3. Implementacija
4. Ispitivanje ispravnosti (testiranje)
5. Održavanje i nadgradnja (maintenance)

Postoje različite metode i filozofije o tome kako treba organizovati razvoj odnosno implementaciju jednog složenog softverskog rešenja. Neke od najpoznatijih metoda su sledeće:

- **Metod vodopada** (waterfall metod) kod koga da bi se započelo sa nekom aktivnošću u razvoju, mora biti prvo završena aktivnost koja je bila započeta pre nje.
- **Iterativni / evolucijski metod**, kod koga se teži da prva verzija kompletnog softvera bude što pre završena. Usled veoma kratkog roka kvalitet ovakve verzije često nije visok, pa se ona u narednim iteracijama (otuda i naziv metode) poboljšava dok se ne dostigne željeni kvalitet.
- **Inkrementalni metod**, u kome se prvo završava softver sa osnovnom funkcionalnošću, a ostale funkcionalnosti se dalje inkrementalno dodaju. Ovaj metod omogućava smanjenje rizika u razvoju i rezultuje kraćim vremenskim periodima potrebnim da se softver izbaci na tržište. Zbog svega ovoga on se danas najčešće i koristi pri izradi komercijalnog softvera.

Inkrementalni metod razvoja softvera u praksi najčešće prati i filozofija agilnog razvoja softvera. Agilni razvoj softvera prati set principa prema kojima zahtevi i rešenja vremenom evoluiraju kroz napore timova koji se sami organizuju. U agilnom razvoju softvera akcenat se stavlja na adaptivno planiranje i ranu isporuku rešenja, brze i efikasne reakcije na nastale promene. Sledeći spisak principa predstavljaju manifesto agilnog razvoja softvera.

1. Zadovoljstvo korisnika se postiže **ranom i kontinualnom** isporukom korisnog softvera
2. Promene specifikacija su dobrodošle čak i u kasnijim fazama razvoja
3. Softver koji funkcioniše se isporučuje često (**nekoliko nedelja, a ne nekoliko meseci**)

4. Bliska, dnevna saradnja ljudi iz menadžmenta i ljudi koji rade na razvoju softvera
5. Projekti su povereni motivisanim pojedincima, kojima se veruje
6. Komunikacija licem u lice je najbolja forma komunikacije u timu
7. Softver koji funkcioniše je osnovna jedinica za merenje napretka
8. Održivi razvoj – **brzina** razvoja treba da bude **konstantna**
9. Konstantna pažnja se posvećuje visokom stepenu kvaliteta i dobrom dizajnu
10. Jednostavnost – umetnost da se **maksimizuje obim posla koji ne mora da se uradi** je od presudne važnosti
11. Najbolji zahtevi i dizajn nastaju u **timovima koji se sami organizuju**
12. Tim redovno (na bazi dotadašnjeg iskustva) razmatra kako može da se unapredi i implementira dogovorene promene

Postoje različiti konkrektni metodi u razvoju koji se oslanjaju na ova pravila. Verovatno najčešće korišćen metod agilnog razvoja softvera u industriji je **Scrum** metod.

Često se dešava da se u toku samog procesa razvoja softvera specifikacije i zahtevi menjaju. Razlozi za to mogu biti brojni. Načešći su da promene u specifikacijama zahtevaju korisnici ili da tržište diktira promenu pravca u kome određeno softversko rešenje treba da se razvija. Zbog ovakve nepredvidivosti i potrebe da se specifikacije često menjaju, tradicionalni načini razvoja softvera ne funkcionišu najbolje. Scrum metodologija prepoznaje činjenicu da se problem ne može potpuno sagledati i razumeti na početku projekta i prihvata praktičan pristup. Taj pristup se ogleda u tome da se energija u toku razvoja ne ulaže na pokušaj da se problem u celosti sagleda i definiše, već se fokus stavlja na maksimizovanje mogućnosti razvojnog tima da konstantno isporučuje korisne inkremente u razvoju softvera i da može brzo da se prilagodi promenama specifikacija, promenama tržišta ili promenama u tehnologiji. Način na koji se ovo postiže je stalna retrospekcija u okviru razvojnog tima i stalno unapređenje procesa rada kroz implementaciju dogovorenih promena.

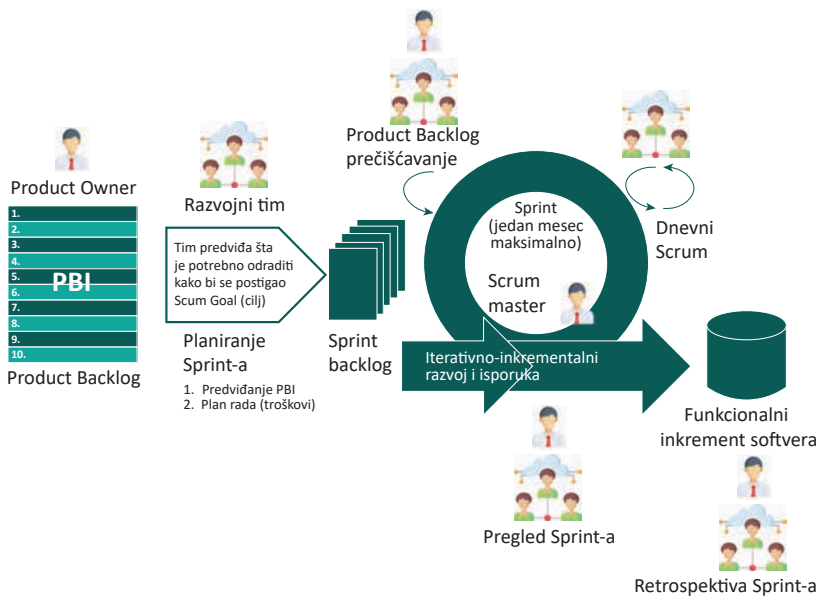
Osnovne tri uloge u Scrum načinu rada su:

- **Product owner** koji je osoba koja održava vezu između menadžmenta i razvojnog tima. Njen zadatak je da definiše takozvane user stories za razvojni tim. User stories zapravo predstavljaju definicije funkcionalnih inkrementa koda koje razvojni tim treba da implementira.
- **Razvojni tim** je tim od 3 do 9 inženjera koji razvijaju softver. Oni dobijaju user stories od product ownera, ali rad i zadatke u okviru tima sami organizuju i raspoređuju. Oni zaista funkcionišu kao tim i sastaju se na dnevnom nivou kako bi razgovarali o napretku i eventualnim problemima.

- **Scrum master** je osoba koja je najčešće i sama član razvojnog tima i ima zadatak da timu pomogne u prevazilaženju svih prepreka sa kojima se na dnevnom nivou susreće. Scrum master nije klasičan vođa tima, već je on fasilitator tima i pomaže timu da što bolje iskoristi sve prednosti Scrum metodologije.

U Scrum metodologiji rad tima je organizovan u sprintove. Jedan sprint traje oko mesec dana i na početku sprinta tim dobija user stories od product owner-a koje treba da implementira u tom sprintu. Tim onda ima sastanak na kome planira podelu rada i vremenske okvire kada šta treba da se završi. U toku trajanja sprinta, članovi tima imaju svakodnevne sastanke od po 10-ak minuta na kojima razmenjuju informacije o napretku i eventualnim problemima. Cilj ovih sastanaka je razmena ideja i eventualna pomoć članovima tima koji se susreću sa poteškoćama. Dnevni satstanci takođe omogućavaju timu da prati izvršenje inicijalnog plana i da na vreme reaguje u slučaju da ne može da ispuni sve što je planirano. Na kraju sprinta tim isporučuje funkcionalni inkrement softvera i održava dva sastanka. Prvi sastanak je pregled sprinta gde se razgovara o tome šta je od postavljenih zadataka postignuto, a na čemu eventualno mora da se radi i u toku sledećeg sprinta. Informacije sa ovog sastanka bitne su za product owner-a i za planiranje user stories-a za sledeći sprint. Drugi sastanak je retrospektiva sprinta, gde članovi tima razgovaraju o tome šta je bilo dobro u njihovom radu, a šta bi eventualno trebalo promeniti. Promene dogovorene na ovom sastanku sprovode se već u sledećem sprintu. Grafički prikaz organizacije rada u okviru Scrum metodologije dat je na sledećoj slici.

Slika 6: Grafički prikaz organizacije rada u okviru Scrum metodologije



13. ČESTE ZABLUDE O SISTEMIMA ZA RAD U REALNOM VREMENU

Jedna od najčešćih zabluda u vezi sa sistemima za rad u realnom vremenu je ta da su to brzi sistemi. Veliki broj sistema za rad u realnom vremenu su zaista brzi, ali postoji i značajan broj onih koji nisu brzi. Na primer sistemi za regulaciju velikih hemijskih postrojenja, ili kontrolu temperature u zgradama su dosta spori.

Mnogi poistovećuju problematiku dizajniranja sistema za rad u realnom vremenu sa problemom kvazi paralelnog organizovanja programskih zadataka. Kao što ćemo u ovom kursu videti, problematika dizajniranja sistema za rad u realnom vremenu mnogo je šira.

Često se misli da postoji nekakva univerzalna, uvek primenljiva metodologija za dizajniranje sistema za rad u realnom vremenu. Ovo je pogrešno, jer su specifičnosti pojedinih sistema jako velike i zavise od oblasti primene. Ne postoji dakle neki univerzalni recept za dizajniranje sistema za rad u realnom vremenu koji bi uvek bio primenljiv. Postoje međutim pravila dobre prakse, koja ako se poštuju čine dizajniranje sitema za rad u realnom vremenu manje zahtevnim. Cilj ovog kursa je da nauči upravo ta pravila dobre prakse koje će biti veoma značajna svakom budućem embedded inžinjeru.

II DEO

OSNOVE PROJEKTOVANJA I IMPLEMENTACIJE SISTEMA ZA RAD U REALNOM VREMENU

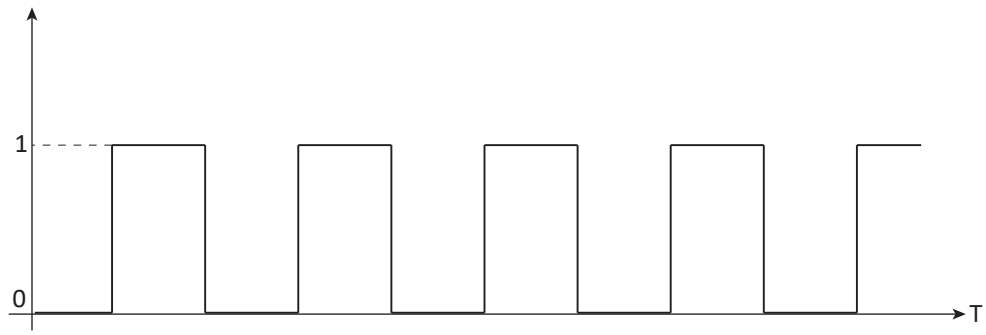


1. MERENJE VREMENA

Ispravno merenje protoka vremena je jako važno za ispravno funkcionisanje sistema za rad u realnom vremenu. Kao što je već ranije bilo reči, vreme se nalazi u samom nazivu i definiciji sistema za rad u realnom vremenu, i za njihov ispravan rad je potrebno ne samo da rezultati izračunavanja budu logički ispravni, već i da ti rezultati stignu u pravo vreme. Nepoštovanje zadatih vremenskih okvira može dovesti do teških posledica, a u nekim slučajevima čak i do uništenja sistema ili ljudskih žrtava. Zbog svega ovoga, svaki sistem za rad u realnom vremenu mora da sadrži časovnik realnog vremena (real time clock), koji omogućava precizno merenje proteklog vremena.

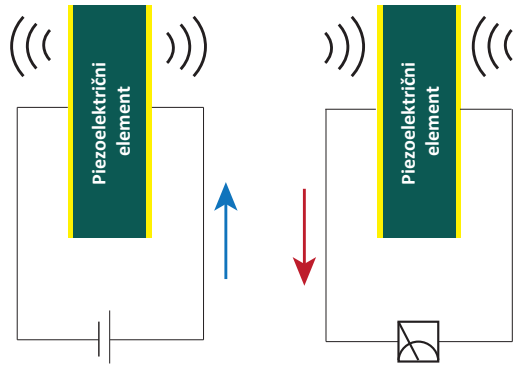
U računarskim sistemima, protok vremena se meri uz pomoć clock signala. To je signal koji upravlja radom procesora i sastoji se najčešće od niza pravougaonih naponskih pulseva jednake širine koja predstavlja i osnovnu periodu clock signala. Tipičan izgled clock signala prikazan je na sledećoj slici.

Slika 7: Tipičan izgled clock signala



Clock signal nastaje kao električni (naponski) signal na izlazu oscilatora koji osciluje fiksnom frekvencijom. Srce oscilatora koji generiše clock signal u računarskim sistemima je kvarcni kristal. Ovaj materijal ima piezoelektrične osobine. To znači da kada se na krajeve kristala primeni električni napon, kristal počne da osciluje. Sa druge strane, ukoliko je element od piezoelektričnog materijala pobuđen mehaničkim oscilacijama, na njegovim krajevima će se generisati električni napon. Ova svojstva piezoelektričnih materijala koriste se u konstrukciji zvučnika i mikrofona. Kod zvučnika, zvuk se generiše primenom napona na krajeve piezoelektričnih elemenata koji osciluju i stvaraju zvuk, dok kod mikrofona zvučni signal izaziva mehaničke oscilacije piezoelektričnih elemenata, koji se onda pretvara u analogni naponski signal, koji se potom diskretizuje i pretvara u digitalni signal. Ilustracija piezoelektričnog efekta, šematski prikaz kvarcnog kristala, kao i fotografija komercijalno dostupne komponente prikazani su na sledeće dve slike.

Slika 8: Ilustracija piezoelektričnog efekta

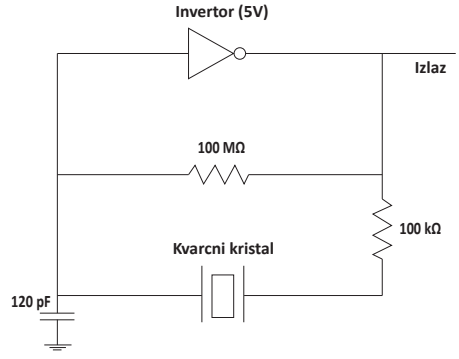


Slika 9: Šematski simbol i fizički izgled kvarcnog kristala



Povezivanjem kvarcnog kristala u paraleli sa invertorom, dobija se elektromehanički oscilator kod koga kvarcni kristal osciluje konstantnom frekvencijom, a na izlazu se dobija naponski signal sa pravougaonim pulsevima koji se pojavljuju sa frekvencijom jednakoj mehaničkoj frekvenciji oscilovanja kvarcnog kristala. Šematski prikaz ovakvog oscilatora dat je na sledećoj slici.

Slika 10: Šematski prikaz elektro-mehaničkog oscilatora za generisanje clock signal



Sistem može imati veći broj časovnika realnog vremena koji svi koriste clock signal. Svi ovi časovnici imaju osnovnu jedinicu za merenje vremena koja je ceo umnožak periode oscilovanja oscilatora koji generiše clock signal.

2. PREKIDI (INTERRUPTS)

Prekid (interrupt) je signal procesoru koji šalje neka hardverska komponenta ili softverski kod koji se trenutno izvršava i koji ukazuje da se desio događaj kome mora trenutno da se posveti pažnja. To podrazumeva da se mora prekinuti sa izvršavanjem programskog zadatka koji je do tada izvršavan i da se procesor mora posvetiti izvršavanju koda koji je vezan sa signalom prekida.

Funkcija koja se izvršava po prijemu prekida i koja sadrži kod koji treba da se izvrši u slučaju prekida naziva se servisna rutina prekida (Interupt Service Routine ISR). Po prijemu prekida, procesor zaustavlja izvršenje trenutnog zadatka i započinje izvršavanje ISR-a. Kada se ISR funkcija završi procesor se vraća izvršavanju zadatka koji je bio prekinut prijemom signala prekida.

Prekidi se prema izvoru koji ih generiše mogu podeliti na:

- **Hardverske**, koji su generisani od strane hardverskih uređaja koji traže trenutnu pažnju procesora. Tipičan primer hardverskih prekida su prekidi kod personalnih računara koji se javljaju usled pritiska na neku od dirki sa tastature. Svaki pritisak na neki od tastera podiže prekid koji zahteva identifikaciju koji taster je pritisnut.
- **Softverske**, koji su generisani od strane specijalne softverske instrukcije koja podiže prekid kada se izvrši. Softverski prekidi se često koriste za upravljanje greškama. Tipičan primer softverskog prekida je prekid koji se javlja pri pokušaju deljenja sa nulom.

Prema načinu generisanja, prekidi se dele na:

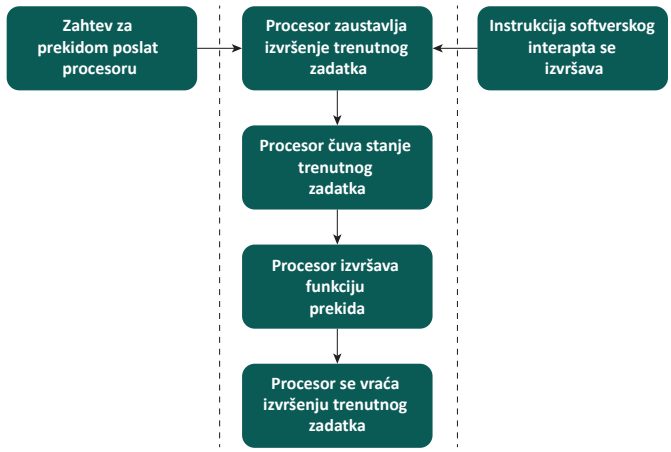
- **Prekide generisane naponskim nivoom (level-triggered)** koji se generišu sve dok je nivo naponskog signala koji je povezan sa prekidom nizak ili visok. Na primer, uzmimo prekid povezan sa naponskim nivoom na nekom od digitalnih ulaza za koji je povezan taster. Ukoliko se taster pritisne, na datom digitalnom ulazu će naponski nivo biti visok (na primer), što će generisati prekid. Ukoliko se taster drži pritisnut, naponski nivo na datom pinu će biti visok, pa će po izvršenju ISR biti ponovo generisan prekid i ponovljeno izvršavanje ISR. Prekidi će se neprestano javljati sve dok taster ne bude pušten i naponski nivo na datom digitalnom ulazu ne postane nizak.

- **Prekide generisane promenom naponskog nivoa (ili ivicom – edge trggered)**, koji se generišu ukoliko dođe do promene naponskog nivoa signala sa niskog na visoki (ili obrnuto). Uzimajući prethodni primer sa tasterom i digitalnim ulazom, kod ovakve vrste prekida, ISR bi se izvršila samo posle pritiska na taster. Ukoliko bi nastavili da držimo taster, ne bi došlo do generisanja novih prekida. Novi prekid bi mogao biti generisan samo puštanjem tastera i njegovim ponovnim pritiskanjem, što bi dovelo do promene naponskog nivoa na ulazu digitalnog pina sa niskog na visoki.
- **Hibridne prekide**, kod kojih je potrebno da se detektuje ivica signala, ali i da signal ostane na novom nivou određeno vreme.

Mehanizam obrade prekida se sastoji od generisanja (dizanja prekida) od strane hardverske komponente (hardverski prekid) ili specijalne softverske instrukcije (softverski prekid) i prijema prekidnog signala od strane procesora. Po prijemu ovog signala, procesor zaustavlja izvršavanje programskog zadatka koji se do tada izvršavao i čuva kontekst njegovog izvršavanja. Kontekst programskog zadatka koji se izvršava predstavljaju pokazivač na memorijsku lokaciju instrukcije koja sledeća treba da se izvrši i pokazivače na sve memorijske lokacije koje sadrže međurezultate izračunavanja koji su postojali u trenutku prekida. Ovi pokazivači se koriste kasnije za ponovo pokretanje prekinutog programskog zadatka. Po čuvanju konteksta prekinutog zadatka, procesor počinje sa izvršavanjem ISR povezane sa prekidom koji se obrađuje. Po izvršenju svih instrukcija u okviru ISR, procesor koristi sačuvani kontekst prekinutog zadatka i nastavlja sa izvršenjem programskog zadatka koji je bio prekinut. Opisani mehanizam obrade prekida prikazan je na grafičkoj šemi na sledećoj strani.

Često se u literaturi sreću različiti pojmovi za programski zadatak. Najčešće korišćeni pojmovi su zadatak, proces i thread. Iako postoje suptilne razlike između ovih pojmova, jer za izvršavanje programskog zadatka nije potrebno upravljanje memorijom, dok je ono potrebno za izvršavanje procesa, a thread je u suštini proces koji je prilagođen izvršavanju u računarskom sistemu sa više procesora. U ovom kursu ćemo sve ove pojmove smatrati sinonimima i gotovo uvek ćemo koristiti pojam programskog zadatka.

Slika 11: Proces obrade prekida

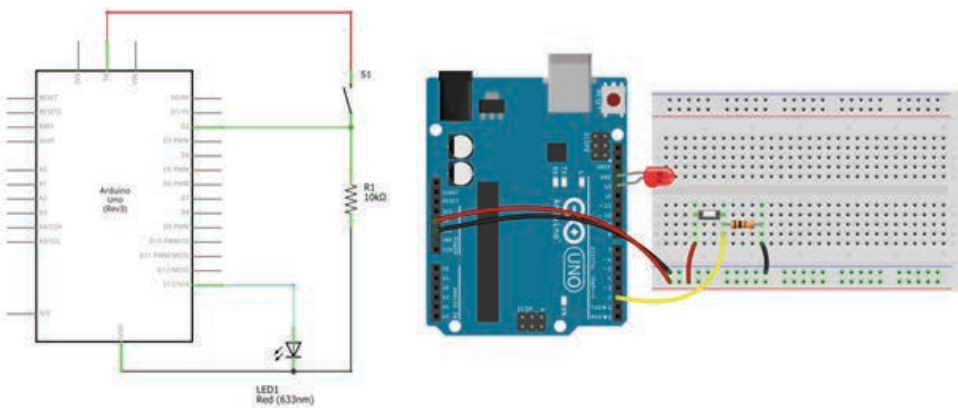


Jednostavan primer rada sa prekidima

Kao jednostavan primer rada sa prekidima, razmotrimo jedan od osnovnih zadataka sa Arduino platformom. Pretpostavimo da je LED dioda povezana na digitalni I/O pin 13 Arduina, a da je na digitalni I/O pin 2, koji omogućava spoljne prekide povezan taster čijim pritiskom nivo naponskog signala na pinu postaje jednak nuli (inače je naponski signal visok, jer je pin povezan na napon napajanja preko pull-up otpornika). Šematski prikaz ovakvog električnog kola dat je na slici ispod.

Potrebno je napisati kod za Arduino tako da dioda ima dva stanja (ili je upaljena ili ugašena), pri čemu će se stanje menjati pritiskom na taster.

Slika 12: Povezivanje komponenti za primer sa tasterom i diodom



Moguće rešenje ovog zadatka, koje koristi prekid generisan silaznom naponskom ivicom na ulaznom digitalnom pinu 2, dato je sledećim kodom.

```
1.  const int PIN_DIODE = 13; // Izabrani pin za diodu
2.  const int PIN_TASTERA = 2; // Izabrani pin za taster
3.  volatile byte Stanje = LOW; // Promenljiva za citanje stanja
                                   tastera
4.  void promena stanja() {
5.      Stanje = !Stanje; // Menja se stanje tastera
6.  }
7.  void setup() {
8.   pinMode(PIN_DIODE, OUTPUT); // Pin diode konfigurisan kao
                                  izlazni pin
9.    pinMode(PIN_TASTERA, INPUT_PULLUP); // Pin tastera konfigurisan
                                            kao ulazni pin
10.   attachInterrupt(digitalPinToInterrupt(PIN_TASTERA), promena_
          stanja, FALLING);
11.     // Ova komanda povezuje funkciju treptanje sa prekidom
12.     // koji generise silazna ivica na pinu 2 za koji je vezan taster
13. }
14. void loop() {
15.     digitalWrite(PIN_DIODE, Stanje);
16.     // Dioda se pali ili gasi u odnosu na stanje tastera
17. }
```

Kao što se može videti, program ne sadrži nikakav kod u glavnoj petlji, već samo instrukciju za promenu stanja diode u ISR-u povezanom sa padajućom naponskom ivicom na pinu 2. To znači da će se željeni deo koda izvršiti samo onda kada taster bude pritisnut, i nema potrebe za neprekidnim čitanjem stanja na pinu 2 da bi se detektovao pritisak na taster, jer će se ISR automatski izvršiti baš po pritisku na taster.

Analizirajmo sada uporedo i moguće alternativno rešenje koje ne koristi prekid.

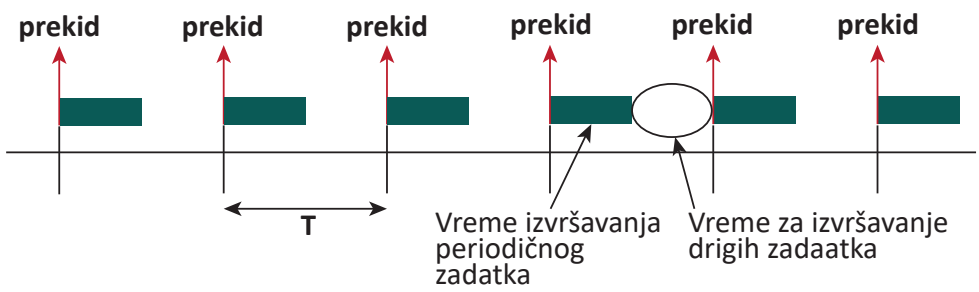
```
1.  const int PIN_DIODE = 13; //Izabrani pin za diodu
2.  const int PIN_TASTERA = 2; //Izabrani pin za taster
3.  volatile int Stanje = 0; //Promenljiva za citanje stanja tastera
4.
5.  void setup() {
6.    pinMode(PIN_DIODE, OUTPUT);//Pin diode konfigurisan kao izlazni pin
7.    pinMode(PIN_TASTERA, INPUT_PULLUP);//Pin tastera konfigurisan
                                          kao ulazni pin
8.  }
9.
10. void loop() {
11.   Stanje = digitalRead(PIN_TASTERA);//Citanje ulazne vrednosti tastera
12.   digitalWrite(PIN_DIODE, Stanje);
13.   //Dioda se pali ili gasi u odnosu na to da li je taster pritisnut
        ili ne
14. }
```

Poređenjem možemo zaključiti šta su osnovne prednosti prekida. U alternativnom rešenju u glavnoj petlji se stalno proverava naponski nivo na pinu i usled toga menja stanje. Ovakav pristup može da funkcioniše ukoliko je broj instrukcija u okviru glavne petlje mali. Međutim, ukoliko u glavnoj petlji imamo ogroman broj instrukcija, očitavanje stanja ulaznog pina neće moći da se radi sa velikom frekvencijom, pa se u takvoj situaciji može desiti da neki pritisak na taster ne bude registrovan. Druga mana ovakvog pristupa gde se ne koristi prekid je što se procesorsko vreme troši na proveravanje stanja pina, što kod implementacije sa prekidom nije slučaj.

Prekid generisan vremenom

I časovnik realnog vremena se može koristiti za generisanje prekida. Ovo omogućava da se prekidi generišu periodično sa izabranom periodom ponavljanja, što je za projektovanje sistema za rad u realnom vremenu veoma korisno. Tipični primeri gde se ovakvi prekidi koriste su u implementaciji upravljačkih algoritama, gde se upravljačke promenljive preračunavaju i primenjuju sa određenom zadatom periodom. Implementiranjem upravljačkih algoritama uz pomoć prekida koji su generisani časovnikom realnog vremena omogućava da se programski zadatak koji mora biti periodičan izvršava periodično, a da se u ostatku procesorskog vremena izvršavaju oni programski zadaci koji nisu vremenski kritični. Ilustracija ovakve implementacione logike data je na sledećoj slici.

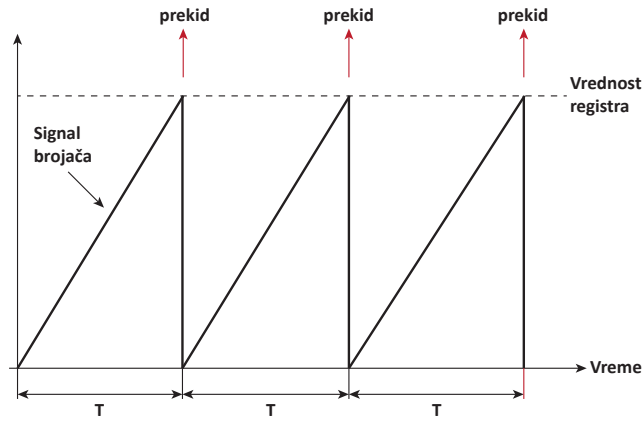
Slika 13: Periodično izvršavanje programskog zadatka uz pomoć prekida generisanih vremenom



Prekidi uzrokovani časovnikom realnog vremena mogu biti generisani sa proizvoljnom periodom koja je jednaka celobrojnom umnošku osnovne periode clock signala. Mehanizam za generisanje ovakvih prekida najčešće je realizovan uz pomoć poređenja vrednosti u registrima. Naime, mehanizam se sastoji iz dva registra u kojima se pohranjuju celi brojevi. U jednom registru nalazi se fiksni broj koji označava željeni broj celebrojnih umnožaka osnovne periode clock signala koji perioda pojave prekida treba da ima.

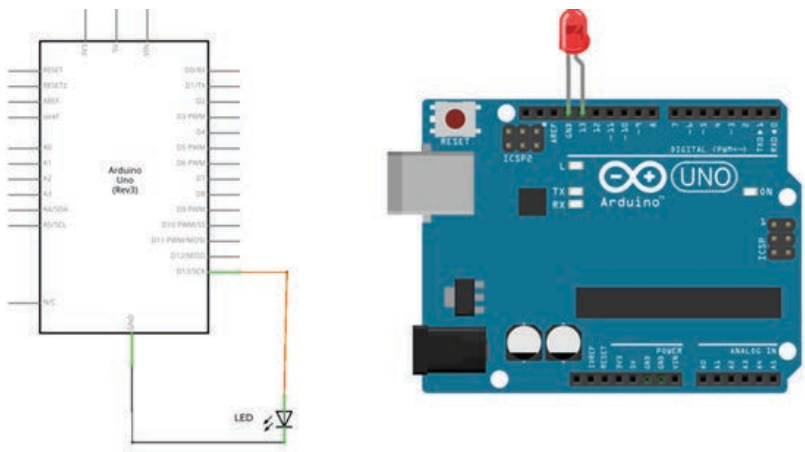
U drugom registru se smešta trenutni broj pulseva u clock signalu od poslednjeg prekida. Kada vrednosti u ova dva registra postanu jednake, generiše se signal prekida i vrednost registra koji sadrži broj pulseva se resetuje na nulu. Osnovna logika ovakvog mehanizma prikazana je na sledećoj slici

Slika 14: Mehanizam generisanja prekida uz pomoć časovnika realnog vremena



Kao ilustraciju primene prekida generisanih vremenom, razmotrimo kolo koje se sastoji od Arduina i led diode povezane na njegov pin 13. Zadatak je napisati program za Arduino koji obezbeđuje naizmenično paljenje i gašenje diode sa određenom zadatom periodom. Elektro šema koja se koristi za ovaj primer data je na sledećoj slici.

Slika 15: Šema hardvera za primer sa trepćućom diodom



Jedno od mogućih rešenja ovog zadatka dato je sledećim kodom koji koristi prekid generisan vremenom.

```
1.  #include <TimerOne.h>
2.  const int PIN_DIODE = 13; //Izabrani pin za prvu diodu
3.  volatile int StanjeDiode = LOW; //Promenljiva koja oznacava stanje
                                    diode
4.
5.  void prekid(){
6.    StanjeDiode = !StanjeDiode;//Dioda menja stanje
7.    digitalWrite(PIN_DIODE, StanjeDiode);
8.    //Dioda ce biti iskljucena, odnosno ukljucena u zavisnosti od
        stanja diode
9.  }
10. void setup() {
11.   pinMode(PIN_DIODE, OUTPUT);//Pin diode konfigurisan kao izlazni pin
12.   Timer1.initialize(500000);
13.   //Inicijalizacija Timer1 i podesavanje perioda trajanja prekida
        od pola sekunde
14.   Timer1.attachInterrupt(prekid);
15.   //Ova naredba vrsi prekid i poziva funkciju prekid
16. }
17.
18. void loop() {
19.   //Nema koda
20. }
```

U prethodnom rešenju komandom Timer1.initialize zadaje se broj osnovnih perioda clock signala koje čine period sa kojim će se javljati prekid. U ISR-u ovog prekida generisanog vremenom, nalaze se komande za promenu stanja diode i setovanje novog stanja. U glavnoj petlji nema nikakvog koda. To znači da će procesor biti slobodan gotovo sve vreme, samo će u određenom vremenskom intervalu promeniti stanje diode. To znači da se dodatno u glavnoj petlji mogu dodati (čak i veći broj) programskih naredbi koje izvršavaju neku korisnu funkciju. Radi poređenja, razmotrimo i alternativnu implementaciju koja ne koristi prekid generisan vremenom, već periodičnost promene stanja diode realizuje uz pomoć vremenskog čekanja.

```
1.  const int PIN_DIODE = 13; //Izabrani pin za prvu diodu
2.  const int TRAJANJE_STANJA = 500;
3.  volatile byte StanjeDiode = LOW; //Promenljiva koja oznacava
                                       stanje diode
4.
5.  void setup() {
6.    pinMode(PIN_DIODE, OUTPUT);//Pin diode konfigurisan kao izlazni pin
7.  }
8.
9.  void loop() {
10.   StanjeDiode = !StanjeDiode;//Dioda menja stanje
11.   digitalWrite(PIN_DIODE, StanjeDiode);
12.   //Dioda ce biti iskljucena, odnosno ukljucenja u zavisnosti od
        stanja diode
13.   delay(TRAJANJE_STANJA);
14. }
```

U slučaju ovakve implementacije u okviru glavne petlje ne bi trebalo dodavati dodatne komande, jer bi se u tom slučaju promenila i perioda sa kojom se menja stanje diode. Kao dodatni problem ove implementacije javlja se činjenica da većinu vremena u toku izvršavanja procesor ne radi ništa, ali je blokiran jer čeka da istekne određeni vremenski period u kome ne može raditi druge korisne stvari.

Prekidanje i onemogućavanje prekida

Čest je slučaj da u sistemu ne postoji samo jedan prekid, već da ih ima više. U tom slučaju se postavlja pitanje šta se dešava kada se generiše prekid dok procesor izvršava ISR nekog drugog prekida koji se ranije pojavio. Za ovakvu situaciju postoje dva rešenja i koje od njih se primenjuje zavisi od arhitekture sistema.

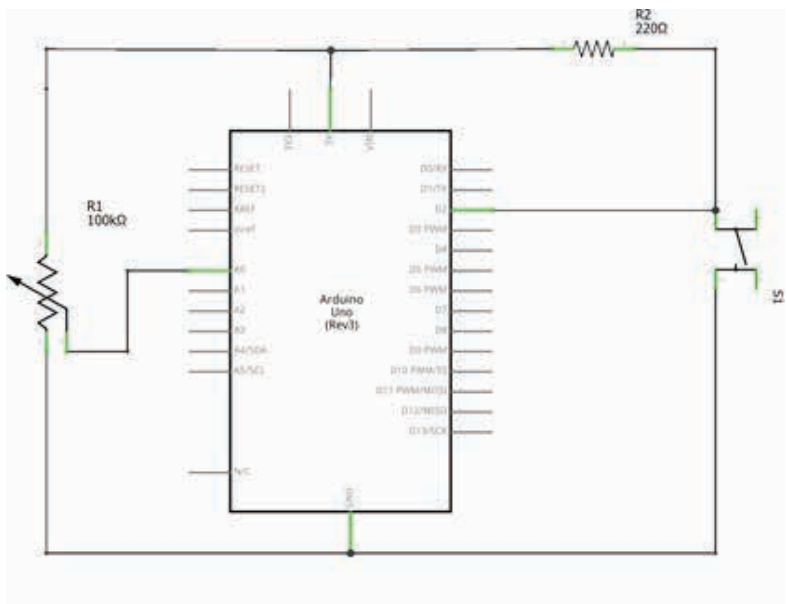
Jedno rešenje je da prekidanje ISR-ova koji se izvršavaju u procesoru ne bude moguće. U ovakvoj arhitekturi, prekid koji se generiše u toku izvršavanja ISR-a nekog drugog prekida biće registrovan i izvršavanje njegove ISR će početi čim se završi izvršenje trenutne. Ukoliko postoji više prekida na čekanju, izvršavaće se redom po prioritetu, a izvršavanje zadatka koji je prekinut od strane prvog prekida će se nastaviti tek kada se svi ISR-ovi na čekanju izvrše. Arduino, na primer, ima ovakav sistem organizovanja prekida.

Alternativno rešenje je dozvoljavanje takozvanog ugnježdavanja prekida, gde se dozvoljava da procesor prekida izvršavanje trenutnog ISR-a da bi se izvršio drugi, vezana za prekid većeg prioriteta. U situaciji kada se vraća u normalan tok rada, procesor u ovom slučaju uvek prvo izvršava zadatak koji je poslednji prekinut. Ovakva arhitektura omogućava postojanje većeg broja prekida od kojih su neki kritični i moraju biti obrađeni odmah po prispeću bez odlaganja (čekanja da se izvrši ISR nekog drugog – manje kritičnog prekida).

U situaciji kada postoji više prekida koji mogu međusobno da se prekidaju, a njihovi ISR-ovi dele određene promenljive ili hardverske resurse (npr. vrše upis na isti serijski port), ali i u situaciji kada postoji samo jedan prekid, a kod u glavnoj petlji i kod u okviru ISR-a dele iste resurse, mora se voditi računa da ne dođe do prekida u trenutku u kojem promena na deljenim resursima nije do kraja završena, što može dovesti do toga da rezultat izvršavanja ISR-a koji deli resurs da pogrešne i vrlo često nepredvidive rezultate. Takođe može se desiti da u okviru glavne petlje ili određenih ISR-ova postoje kritični delovi koda čije se izvršavanje iz određenih razloga ne sme prekidati. U cilju rešavanja ovakvih situacija, postoji mogućnost blokiranja prekida u delovima koda koji su kritični.

Kao primer situacije u kojoj se za deo koda mora blokirati prekid, uzimamo šemu sa Arduinom u kojoj imamo taster povezan na digitalni pin 2 i potenciometar povezan na analogni ulaz A0. Potrebno je da se vrednosti sa analognog ulaza čitaju i ispisuju na serijski port. Ukoliko se pritisne taster, potrebno je očitati vrednost na digitalnom ulazu 2 i nju ispisati na serijski port. Šematski prikaz hardvera korišćenog u ovom primeru dat je na sledećoj slici.

Slika 16: Šematski prikaz hardvera korišćenog u primeru koji ilustruje blokiranje prekida



Ovaj zadatak može se takođe rešiti upotrebom prekida, gde bi se u glavnoj petlji čitale vrednosti sa analognog izlaza i upisivale na serijski port, dok bi se čitanje vrednosti sa digitalnog ulaza i njegov upis na serijski port vršilo u okviru ISR-a koji bi bio povezan sa silaznom naponskom ivicom na digitalnom pinu 2. Primer Arduino koda koji odgovara ovakvoj implementaciji je sledeći.

```
1.  const byte interruptPin = 2; //Pin na kom ce se vrsiti prekid
2.
3.  void prikazStanjaDugmeta(){
4.    //Citanje stanja ulaznog pina:
5.    int stanjeDugmeta = digitalRead(interruptPin);
6.    //Ispis stanja na serijski port
7.    Serial.println(stanjeDugmeta);
8.  }
9.
10. void setup() {
11.   //Inicijalizacija serijske komunikacije na 9600 bita po sekundi
12.   Serial.begin(9600);
13.   //Specifikacija prekida
14.   pinMode(interruptPin, INPUT_PULLUP);
15.   attachInterrupt(digitalPinToInterrupt(interruptPin),prikazStanja
      Dugmeta, FALLING);
16. }
17.
18. void loop() {
19.   //Citanje input-a sa analognog pina 0
20.   int vrednostSaSenzora = analogRead(A0);
21.
22.   //Onemogucavanje pojave prekida
23.   noInterrupts();
24.
25.   //Ispisivanje ocitane vrednosti
26.   Serial.println(vrednostSaSenzora);
27.
28.   //Ponovo omogucavanje pojave prekida
29.   interrupts();
30. }
```

Ono što razlikuje ovaj kod od primera koje smo prethodno pokazali je to da u glavnoj petlji, pre komande za ispis na serijski port, postoji komanda *noInterrupts( )* koja onemogućava prekid programskog zadatka posle njenog izvršavanja. Posle komande za ispis na serijski port, nalazi se komanda *interrupts( )*, koja ponovo omogućava prekid programskog zadatka od strane ISR-a. Dakle komanda u glavnoj petlji za ispis na serijski port je zaštićena od prekidanja i u njeno izvršavanje ne može biti prekinuto, bez obzira što se u međuvremenu javio prekid. Ova zaštita od prekidanja je neophodna jer i ISR ima komandu za ispis na isti serijski port. Ukoliko komanda u glavnoj petlji ne bi bila zaštićena, moglo bi da dodje do delimičnog ispisa, zatim prekida, ispisa koji se vrši u ISR-u i ponovnog ispisa (do kraja) zadatka u glavnoj petlji koji je prekinut. Ukoliko bi došlo do ovakve situacije sadržaj ispisan na serijskom portu ne bi mogao da se tumači. Zbog toga je neophodno sprečiti istovremene pokušaje pisanja na isti serijski port.

Dobra praksa u radu sa prekidima

Kroz par prethodnih primera smo videli da korišćenje prekida može biti veoma korisno pri rešavanju raznih programskih zadataka koji se javljaju u implementaciji sistema za rad u realnom vremenu. Međutim pri korišćenju prekida, treba voditi računa o par pravila dobre prakse koja pomažu u debagovanju koda i ograničavanju njegove kompleksnosti.

Prvo pravilo je da bi kod u okviru ISR-a trebao da bude kratak, odnosno da bi servisiranje prekida trebalo da bude brzo, pogotovo ukoliko u sistemu postoji veći broj prekida, jer u suprotnom može doći do zagušenja i gomilanja velikog broja ISR-a koji čekaju na izvršenje.

Vremenski kritični zadaci se obično smeštaju u okvre ISR-a koje imaju najveći prioritet, dok se vremenski nekritični zadaci smeštaju u glavnu petlju.

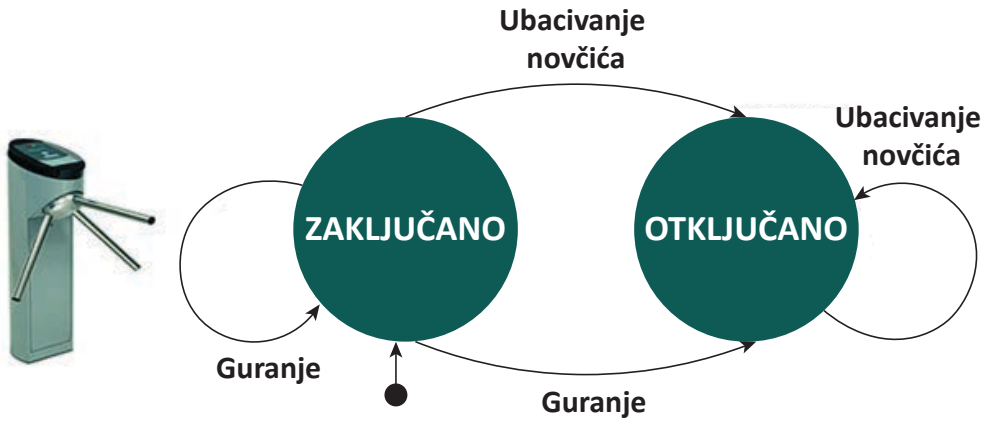
Isključenje prekida treba koristiti samo kada je to zaista neophodno, kako se kod i njegova struktura ne bi komplikovali. U slučaju da postoji veliki broj prekida i veliki broj mesta u kodu u kojima se prekidi blokiraju, tako da postaje veoma teško predvideti šta se kada dešava, treba razmisliti o korišćenju paralelnog (odnosno kvaziparalelnog) izvršavanja zadataka i korišćenju usluga real time operativnog sistema, o čemu će više reči biti nešto kasnije.

3. KONAČNI AUTOMATI

Konačni automat (na engleskom Finite State Machine ili samo State Machine) je matematički model izračunavanja koji se koristi za konstrukciju programa ili sekvencijalne logike. Konačni automat je abstraktna mašina koja se u jednom trenutku može naći u tačno jednom od konačno mnogo stanja. Stanje automata se menja usled nekog događaja (koji je najčešće signaliziran prekidom) ili ukoliko je ispunjen neki logički uslov. Konačni automat je potpuno definisan listom svih stanja i uslova za prelazak sa jednog stanja na drugo.

Kao jednostavan primer mašine stanja možemo razmatrati program za upravljanje kružne rampe koja se otključava ubacivanjem novčića. Ovaj sistem ima dva stanja Zaključano i Otključano koja redom označavaju stanje u kome je rampa zaključana i u kome je ona otključana. Početno je sistem u stanju Zaključano. Ukoliko se ubaci novčić sistem prelazi iz stanja Zaključano u stanje Otključano. Guranje rampre dovodi do prelaska stanja iz Otključanog u Zaključano. Kada je stanje sistema Otključano i ubacuje se novčič, stanje ostaje otključano. Ukoliko je stanje Zaključano i rampa se gura, stanje ostaje zaključano. Ovakav konačni automat može se grafički predstaviti na sledeći način.

Slika 17: Konačni automat kružne rampe



Alternativno konačni automat kružne rampe može se predstaviti i uz pomoć tabele.

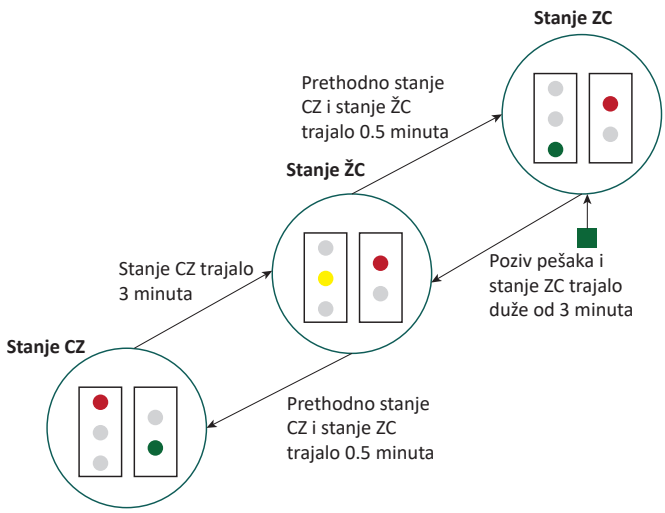
Tabela 1: Opis konačnog automata kružne rampe

Trenutno stanje	Događaj	Novo stanje	Aktivnost
Zaključano	Ubačen novčić	Otključano	Rampa se drži zaključana
	Guranje	Zaključano	
Otključano	Ubačen novčić	Otključano	Rampa se drži otključana
	Guranje	Zaključano	

Nešto složeniji primer konačnog automata je primer sistema koji se satoji od dva semafora, jednog za automobile i jednog za pešake. Semafor za pešake ima i taster uz pomoć koga pešaci mogu da traže paljenje zelenog svetla. Kada nema pešaka treba da bude zeleno za vozila i crveno za pešake. Po pritisku pešaka na taster pali se zuto za vozila, a zatim i crveno za vozila i zeleno za pešake. To traje 3 minuta i onda se pali crveno za pešake i žuto pa zeleno za vozila. Sledeći poziv pešaka biće uslišen tek pošto je zeleno svetlo za vozila trajalo duže od 3 minuta.

Ovakav sistem ima tri konačna stanja koja možemo obeležiti sa ZC (zeleno za vozila, crveno za pešake), ŽC (žuto za vozila, zeleno za pešake) i CZ (crveno za vozila i zeleno za pešake)

Slika 18: Konačni automat semafora



Konačni automati imaju veliku primenu u praksi i sreću se u softveru koji upravlja velikim brojem uređaja koji nas okružuju. Tipični primeri su automati za prodaju, sistem za naplatu parkinga i drugi sistemi sa rampama, liftovi, semafori, elektronske brave sa sigurnosnim kodom, komunikacijski protokoli.

4. PODELA POSLOVA NA PROGRAMSKE ZADATKE

Radi lakšeg obavljanja složenih upravljačkih poslova, najčešće je potrebno program koji se implementira podeliti na manje programske zadatke. Podela posla na zadatke nije uvek jednostavna i često zahteva određena iskustva i veštine od projektanta. Ne postoje fiksna pravila kako se neki posao može podeliti na programske zadatke. Pri podeli posla treba voditi računa o logičkoj celovitosti programskih zadataka i potrebi da sva upravljačka logika u okviru jednog programskog zadatka bude na jednom mestu.

Hardverske platforme koje se koriste u sistemima za rad u realnom vremenu ponekad imaju više procesora. U tom slučaju javlja se jasna potreba da se programski zadaci izvršavaju paralelno kako bi se maksimalno iskoristili svi procesori u sistemu. Međutim mnogo je češći slučaj u praksi da hardverske platforme sistema za rad u realnom vremenu imaju samo jedan procesor.

Pored toga ove platforme najčešće imaju vrlo oskudne ostale hardverske resurse. Kako je čest slučaj da pojedinačni zadaci pristupaju različitim hardverskim resursima, kada bi se zadaci izvršavali sekvencijalno (jedan po jedan), određeni hardverski uređaji dugo vremena ne bi bili iskorišćeni, pa bi iskorišćenost čitavog sistema bila mala. Zbog toga se često i kod hardverskih platformi sa jednim procesorom teži ostvarenju paralelnosti u izvršavanju programskih zadataka. Jasno je da se u ovom slučaju ne može ostvariti paralelnost u pravom smislu te reči, ali se može ostvariti privid paralelnosti ili kvaziparalelnost. Ona se ostvaruje stalnim dodeljivanjem procesora različitim programskim zadacima, gde se procesor dodeljuje jednom zadatku određeno vreme, onda se taj zadatak prekida i procesor se dodeljuje nekom drugom zadatku i tako redom. Ovakvim postupkom se ostvaruje mnogo bolja uposlenost hardverskih komponenti.

Naravno, organizacija postupka dodele procesora nije trivijalna. Ovaj postupak naziva se raspoređivanje ili engleski scheduling. Postoje različiti algoritmi i filozofije o tome kako treba vršiti raspoređivanje, odnosno dodelu procesora različitim programskim zadacima. Takođe, sa obzirom da se zadaci izvršavaju kvaziparalelno mora se voditi računa i o njihovoj sinhronizaciji i komunikaciji. U naredna dva poglavlja daćemo više detalja o raspoređivanju i sinhronizaciji i komunikaciji među procesima.

5. RASPOREĐIVANJE PROGRAMSKIH ZADATAKA

U sistemu u kome se programski zadaci izvršavaju kvaziparalelno, svaki programski zadatak se aktivira nekim događajem u sistemu. U slučaju periodičnih zadataka koji se izvršavaju sa tačno određenom periodom ponavljanja, signal za aktivaciju programskog zadatka je prekid koji dolazi sa tajmera. U slučaju aperiodičnih programskih zadataka, zadaci se aktiviraju uz pomoć hardverskih prekida i signala sa digitalnih ili analognih ulaza.

Sama aktivacija programskog zadatka ne znači nužno da će on odmah početi i da se izvršava. Zapravo u jednom trenutku može biti više programskih zadataka koji su aktivirani i koji čekaju na izvršenje. Dok čekaju na izvršenje programski zadaci se nalaze u redu čekanja. Svaki programski zadatak koji treba da se izvrši prolazi kroz nekoliko karakterističnih vremenskih trenutaka. To su trenutak aktivacije t_a u kome je stigao signal da dati zadatak treba da se aktivira, zatim trenutak početka izvršavanja u kome se zadatku prvi put dodeljuje procesor t_p, kao i trenutak u kome se izvršavanje zadatka završava t_z. U vremenskom periodu od t_p do t_z izvršavanje datog zadatka može biti prekidano jednom ili više puta kako bi procesor trenutno bio dodeljen programskim zadacima sa višim prioritetom. Ukupno vreme koje procesor provede izvršavajući dati zadatak označavamo sa *C*. Ovo vreme je uvek manje ili jednako od vremena koje protekne od početka do kraja izvršavanja zadatka:

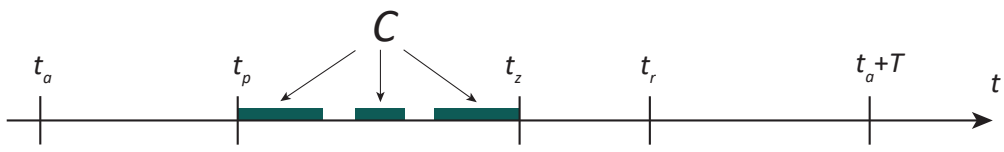
$$C \leq t_z - t_a$$

Ukoliko za dati zadatak postoji i rok izvršenja i taj trenutak označimo sa t_r, onda da bi program ispravno funkcionisao i da bi vremenski rok bio zadovoljem mora da važi:

$$t_z < t_r$$

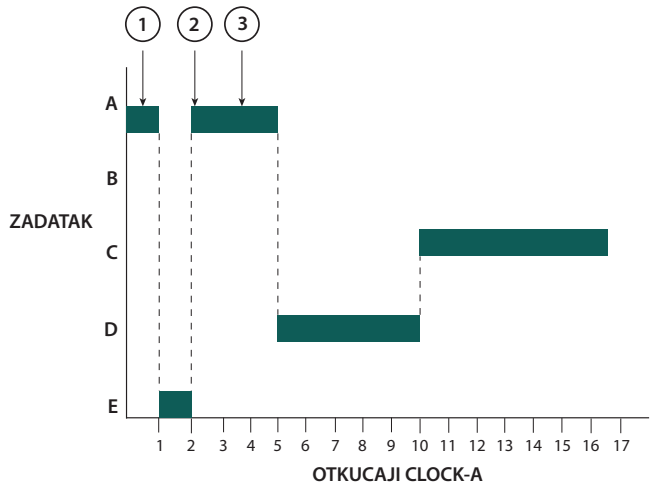
U slučaju da je programski zadatak periodičan, sledeći momenat njegove aktivacije biće t_a+T, gde je *T* perioda sa kojom se ponavlja izvršenje tog periodičnog programskog zadatka. Sledeća slika ilustruje opisane karakteristične vremenske trenutke jednog programskog zadatka.

Slika 19: Karakteristični vremenski trenuci pri izvršavanju programskog zadatka



Prema tome da li se dozvoljava prekidanje izvršavanja programskog zadatka koji jednom započne sa izvršavanjem raspoređivanje može biti premptivno (engleski **preemptive**) i ne-premptivno (non-**preemptive**). Kod preemptive raspoređivanja dozvoljava se prekidanje izvršavanja zadataka. Po ovom sistemu raspoređivanja, zadatak koji se trenutno izvršava biće prekinut ukoliko se u redu čekanja pojavi zadatak sa većim prioritetom od njega, kako bi se processor dodelio tom zadatku sa većim prioritetom. Kao ilustraciju ovog načina raspoređivanja možemo uzeti scenario prikazan na sledećoj slici.

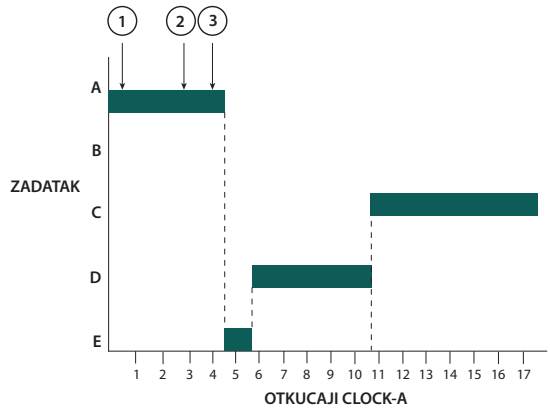
Slika 20: Primer preemptive raspoređivanja



U prikazanom scenariju izvršava se programski zadatak A. U trenutku 1, stiže signal za aktivaciju programskog zadatka E koji ima viši prioritet od programskog zadatka A. Usled toga se izvršavanje tog programskog zadatka prekida, i izvršava se programski zadatak E. Kada se završi izvršavanje ovog programskog zadatka, procesor se ponovo dodeljuje programskom zadatku A. U trenucima 2 i 3 stižu zahtevi za izvršavanje programskih zadataka C i D. U ovom slučaju zadatak A se izvršava pre nego što bude prekinut. Posle njega se izvršava prvo zadatak D kao zadatak sa većim prioritetom, a kada se on izvrši, izvršava se zadatak C. Pri izvršavanju zadataka D i C nema prekidanja, jer se ni u jednom trenutku u redu čekanja nije našao zadatak sa većim prioritetom zbog koga bi bilo potrebno prekinuti programski zadatak koji se trenutno izvršava.

Kod non-preemptive raspoređivanja nema prekidanja izvršavanja programskih zadataka, već se zadatak kada jednom počne njegovo izvršavanje izvršava do kraja. Sledeća slika prikazuje scenario non-preemptive raspoređivanja za istu situaciju kao i ranije.

Slika 21: Primer non-preemptive raspoređivanja



Prvo se izvršava zadatak A. U trenucima 1, 2 i 3 stižu zahtevi za aktivaciju zadataka E, C i D redom. Kako je raspoređivanje non-preemptive, izvršavanje zadatka A se ne prekida, već se ovaj zadatak izvršava do kraja. Po završetku njegovog izvršavanja, izvršava se zadatak E, pa zadaci D i C prema njihovim prioritetima.

Prema tome da li se odluke o raspoređivanju zadataka donose statički, pre početka izvršavanja programa, ili dinamički u toku samog izvršavanja, raspoređivanje može biti **statičko** ili **dinamičko**.

Statičko raspoređivanje

Kod statičkog raspoređivanja svakom programskom zadatku se u toku dizajniranja softvera dodeljuje fiksni prioritet koji se ne menja u toku samog izvršavanja programa. U toku izvršavanja programa, procesor se uvek prvo dodeljuje programskom zadatku sa najvećim prioritetom. Ukoliko u jednom trenutku u redu čekanja postoji više zadataka sa istim stepenom prioriteta, onda se raspoređivanje između tih zadataka vrši po nekom drugom kriterijumu. Na primer, čest je slučaj da pravo prvenstva dobije zadatak koji se najduže vremena nalazi u redu čekanja.

Kod statičkog raspoređivanja postoji više filozofija o tome kako treba dodeliti prioritete procesima u fazi dizajniranja softvera. Svaki od ovih načina dodele prioriteta mora da vodi računa o osnovnom zahtevu kod sistema za rad u realnom vremenu, a to je da se svi programski zadaci moraju završiti u datim vremenskim okvirima. Ovaj uslov se može matematički jednostavno predstaviti za program sa N programskih zadataka koji su svi periodični sa periodom T_i i trajanjem izvršavanja D_i. Za ovakav hipotetički program, uslov da svaki programski zadatak bude izvršen pre novog signala za njegovu aktivaciju dat je sledećom formulom.

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Svaki metod dodele prioriteta programskim zadacima koji vodi računa o ovom uslovu je validan. Jedan od metoda koji dokazano ispunjava ovaj uslov i koji se često koristi u praksi je dodela prioriteta prema očekivanoj učestalosti ponavljanja izvršenja programskog zadatka. Kod ovog načina dodele prioriteta, veći prioritet se dodeljuje onim zadacima koji se češće izvršavaju. Postoje i drugi kriterijumi za dodelu prioriteta. Jedan od njih je i dodela prioriteta prema trajanju izvršavanja zadataka, gde se veći prioritet dodeljuje programskim zadacima čije izvršavanje kraće traje. Postoji i mogućnost kombinovanja ova dva kriterijuma, gde bi najveći prioritet onda dobijali zadaci čije izvršavanje se često ponavlja i kratko traje. U praksi se međutim najčešće koristi dodeljivanje prioriteta prema učestanosti ponavljanja. Naravno, ova pravila se mogu delimično i modifikovati kada postoje neki veoma važni i kritični zadaci kojima se daje visok prioritet, iako bi prema ovim ustaljenim pravilima raspoređivanja trebalo da imaju nizak prioritet.

Dinamičko raspoređivanje

Kod dinamičkog raspoređivanja prioriteti zadataka nisu fiksirani tokom vremena, već se menjaju u toku izvršavanja programa. Pravila za dinamičku promenu prioriteta zadataka definišu se tako da ostvare raspoređivanje koje omogućava poštovanje svih zadatih vremenskih rokova. Najčešće korišćen metod dinamičkog dodeljivanja prioriteta je onaj po kome se najviši prioritet dodeljuje onom zadatku čiji je rok izvršenja vremenski najbliži.

Ovakvim načinom dodele prioriteta se postiže ispunjenje zadatih vremenskih rokova za sve vremenski kritične zadatke. Dinamičko raspoređivanje zahteva više softverskih resursa nego statičko. Algoritmi za dinamičko raspoređivanje zahtevaju više memorije i čine sistem generalno komplikovanijim.

Raspoređivanje vremenski nekritičnih zadataka

Kod vremenski nekritičnih zadataka, raspoređivanje se može vršiti na veliki broj različitih načina, jer pri ovom raspotređivanju ne mora da se vodi računa o ispunjenju vremenskih rokova. Filozofija raspoređivanja zavisi od toga šta je u konkretnom slučaju važnije. Neki od kriterijuma za raspoređivanje koji se koriste za vremenski nekritičke zadatke su:

- Pravedno raspoređivanje vremena svim zadacima, gde se svakom programskom zadatku koji se nalazi u listi čekanja dodeljuje isti deo procesorskog vremena.
- Favorizacija kraćih zadataka, gde je cilj da se u što kraćem vremenu završi izvršavanje što većeg broja programskih zadataka.
- Strategija kod koje se prvo izvršavaju programski zadaci koji najduže čekaju u redu čekanja. Cilj ovakve strategije je da vreme čekanja koje zadaci provedu u redu čekanja bude što manje.
- Strategija koja maksimizuje iskorišćenost procesora u toku izvršavanja različitih programskih zadataka.

Raspoređivanje u praksi

U praksi se ne koristi isključivo ni jedan od nabrojanih teoretskih principa raspoređivanja. Koriste se najčešće različite kombinacije nabrojanih principa raspoređivanja. Takođe, u praksi čisto teoretski principi raspoređivanja nisu dominantni, već se koriste heuristički principi koji su svoju valjanost potvrdili dugogodišnjim praktičnim korišćenjem. Procesom raspoređivanja programskih zadataka bavi se operativni sistem. Kasnije u kursu kao konkretan primer operativnog sistema učićemo Linux operativni sistem.

6. SINHRONIZACIJA I KOMUNIKACIJA MEĐU PROGRAMSKIM ZADACIMA

U situaciji u kojoj se programski zadaci izvršavaju kvaziparalelno, potrebno je da oni budu sinhronizovani, a često i da komuniciraju jedni sa drugima.

Sinhronizacija programskih zadataka je neophodna kada se mora voditi računa o pravima prvenstva među zadacima. To se dešava u situacijama kada je za izvršavanje neke akcije jednog programskog zadatka neophodno da se prvo završi akcija nekog drugog programskog zadatka.

Pod komunikacijom među programskim zadacima podrazumeva se razmena informacija među njima. Komunikacija se najčešće ostvaruje kroz deljenje promenljivih, gde jedan proces upisuje vrednost u određenu promenljivu, a drugi je čita i na taj način razmenuju informacije.

Mehanizam deljenja promenljivih može dovesti do problema ukoliko oba programska zadatka traže da istovremeno pristupaju istoj promenljivoj. Kako većina naredbi čitanja i pisanja nisu atomske (ne izvršavaju se kao jedna jedinstvena mašinska instrukcija, već kao niz mašinskih instrukcija), prekidanje izvršenja jednog zadatka u toku upisa u promenljivu i izvršenje dela drugog zadatka kojim se čita iz te promenljive ili u nju upisuje može dovesti do toga da pročitani ili upisani sadržaj bude pogrešan, što u sistemu može izazvati ozbiljne probleme. Zbog toga se u mehanizmu deljenja promenljivih mora voditi računa o tome da se ne dozvoli istovremeni pristup za više programskih zadataka. Ovo se postiže tako što se sprečava prekidanje takozvanog kritičnog dela koda koji vrši upis ili čitanje date promenljive zadatkom koji vrši pristup istoj promenljivoj. Ovakav mehanizam zaštite se naziva međusobno isključenje.

Jedan od načina da se ostvari međusobno isključenja je i korišćenje softverskih semafora. Semafori su jednostavan mehanizam koji se sastoji od celobrojne promenljive čije je inkrementiranje ili dekrementiranje atomska operacija (izvršava se uz pomoć jedne mašinske instrukcije). Osnovne dve komande koje semafor koristi su *Wait ( )* i *Signal ( )*. Pozivom funkcije *Wait (x)* vrednost celobrojne promenljive *x* smanjuje se za 1 ukoliko je ona veća od 0. Ukoliko je promenljiva *x* jednaka 0, ova funkcija čeka da promenljiva postane veća od nule da bi je dekrementirala. Funkcijom *Signal (x)* celobrojna promenljiva se povećava za 1 (inkrementira se). Promenljiva semafora se obično inicijalizuje na vrednost 1.

Kao primer korišćenja semafora za međusobno isključenje dva programska zadatka koja dele istu promenljivu, možemo uzeti sledeći primer:

Zadatak 1:	Zadatak 2:
Set komandi A	**Set komandi C**
Wait (x)	**Wait (x)**
Komande sa deljenom promenljivom	**Komande sa deljenom promenljivom**
Signal (x)	**Signal (x)**
Set komandi B	**Set komandi D**

Promenljiva *x* je inicijalizovana na 1. Prvi od zadataka koji izvrši funkciju *Wait (x)* smanjuje vrednost promenljive *x* na 0. Vrednost ove promenljive postaće ponovo 1, tek kada se izvrši komanda *Signal (x)* istog tog zadatka. Ukoliko u međuvremenu drugi zadatak pokuša da izvrši svoju funkciju *Wait (x)*, moraće da čeka dok vrednost promenljive *x* ponovo ne postane 1. Na ovaj način obezbeđuje se da delovi koda kojima se pristupa deljenoj promenljivoj ova dva programska zadatka ne mogu da prekidaju jedan drugi, čime se čuva integritet deljene promenljive.

Pored deljenja promenljivih, često je u praksi potrebno vršiti i uslovnu sinhronizaciju zadataka. Ova sinhronizacija je potrebna kada izvršavanje određene akcije u okviru nekog programskog zadatka ima smisla ili je bezbedno samo ukoliko su prethodno ispunjeni određeni uslovi. Uzmimo na primer čitanje i pisanje u neki bafer. Ukoliko je bafer prazan nema smisla vršiti čitanje iz njega. Da bi se iz bafera moglo čitati prethodno mora barem jedanput da se izvrši funkcija pisanja. Sa druge strane ukoliko je bafer pun, nema smisla vršiti upis u bafer. Da bi u bafer moglo dalje da se upisuje, neophodno je da se makar jedanput pre toga izvrši funkcija čitanja. Mehanizam semafora takođe se može koristiti i za ostvarivanje uslovne sinhronizacije. Primer dva programska zadatka koja koriste ovakav mehanizam je sledeći:

Zadatak 1:	Zadatak 2:
Set komandi A	**Set komandi B**
Wait (x)	**Signal (x)**
Komande koje se smeju izvršiti tek pošto zadatak 1 izvrši komande B	**Set komandi D**

U datom primeru u programskom zadatku 1 postoje komande koje se mogu izvršiti samo ukoliko se prethodno izvrši set komandi B programskog zadatka 2. U ovom slučaju potrebno je promenljivu *x* inicijalizovati na 0. Zbog prisustva komande *Wait (x)* programski zadatak 1 biće blokiran sve dok se ne izvrši komanda *Signal (x)* programskog zadatka 2, čime se postiže uslovno isključenje dela koda programskog zadatka 1.

Osnovna mana semafora je da pri njihovom korišćenju može doći do takozvanog deadlocka ili zaključavanja zadataka. Ovo je pojava u kojoj dva programska zadatka drže po jedan resurs, čekajući da upravo onaj drugi zadatak oslobodi svoj resurs. U takvoj situaciji ni jedan zadatak ne može dalje da se izvršava i oba ostaju zauvek zaključana. Primer dva programska zadatka kod kojih može doći do deadlocka je sledeći:

Zadatak 1:	Zadatak 2:
Set komandi A	**Set komandi C**
Wait (x)	**Wait (x)**
Wait (y)	**Wait (y)**
Set komandi B	**Set komandi D**

Kod ova dva programska zadatka ukoliko dođe do izvršavanja *Wait (x)* komande programskog zadatka 1, a onda izvršavanje ovog zadatka bude prekinuto da bi se izvršila *Wait (y)* naredba programskog zadatka 2, dolazi do deadlocka. Razlog je da programski zadatak 1 čeka da se inkrementira promenljiva *y*, a drugi programski zadata čeka da se inkrementira promenljiva *x*. Drugim rečima oba programska zadatka su blokirana i čekaju da onaj drugi odblokira njihov rad, što se nikada ne događa.

Zbog ove mane semafori se ne koriste toliko često u složenim sitemima. Kod jednostavnijih programa semafori se i dalje koriste u praksi. Alternativa semaforima su monitori kod kojih su kritični delovi zadataka enkapsulirani u procedure koje ne mogu da se prekidaju. Kod monitora deljenim promenljivama ne može se pristupati direktno, već preko procedura koje ne mogu da se prekidati.

7. MODELOVANJE SISTEMA ZA RAD U REALNOM VREMENU

Kod softverskih rešenja u sistemima za rad u realnom vremenu koji su složeni, model igra veoma važnu ulogu. Naime model je pojednostavljena apstrakcija sistema koja olakšava njegovo razumevanje i dozvoljava lakšu komunikaciju o njemu među inženjerima razvoja. Postoje različiti neformalni postupci za dokumentovanje i modelovanje sistema za rad u realnom vremenu. Neke od njih su skica principa funkcionisanja ili tekstualna specifikacija funkcionalnosti i svojstava sistema. Međutim ove metode modelovanja i dokumentacije nisu standardizovani i zbog toga mogu dovesti do zabuna ukoliko ljudi koji ih koriste imaju različito tumačenje. Zbog toga je poželjno da se koriste standardizovane metode modelovanja i dokumentovanja. Najčešće korišćena metoda u industriji danas su UML diagrami. UML je skraćenica od Unified Modeling Language ili objedinjeni jezik za modelovanje. UML standardizuje načine za vizuelni prikaz preko dijagrama organizacije softvera u smislu klasa i relacija među njima, zatim vizuelizaciju načina upotrebe sistema za rad u realnom vremenu, te sekvencijalnog izvršavanja pojedinih funkcionalnosti softvera, saradnje između različitih programskih zadataka ili vizuelizaciju konačnih automata.

UML strukturni dijagrami

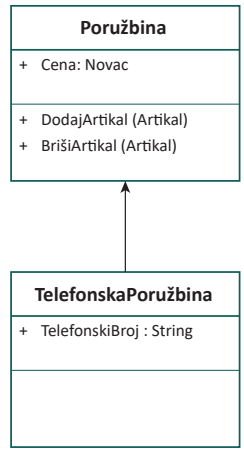
Strukturni UML dijagrami predstavljaju organizaciju koda. Oni predstavljaju sve klase u programu i njihove relacije. Svaka klasa je predstavljena kvadratom koji sadrži tri polja. Jedno polje sadrži ime klase, drugo sadrži sve atribute klase i njihove tipove, a treće specificira sve metode klase. Na sledećoj slici grafički je prikazana klasa *Porudzbina* koja ima atribut *Cena* tipa *Novac* i metode *DodajArtikal* i *BrisiArtikal*.

Slika 22: UML definicija klase



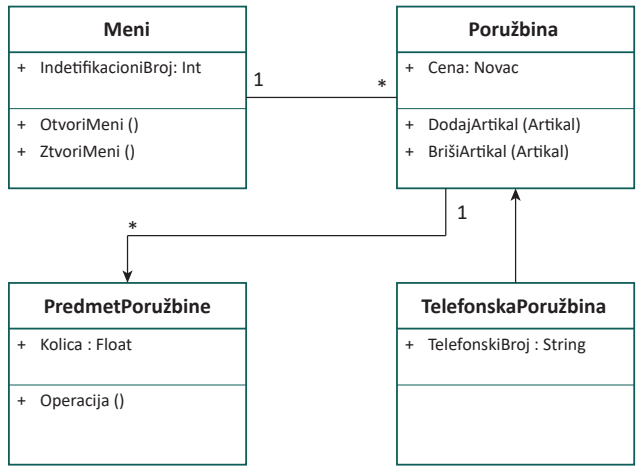
Nasleđivanje se u strukturnim dijagramima označava nepopunjenom strelicom koja ide iz klase koja se izvodi a završava se u osnovnoj klasi. Na sledećem dijagramu prikazana je već spomenuta klasa Porudzbina i iz nje izvedena klasa TelefonskaPorudzbina.

Slika 23: UML dijagram nasleđivanja



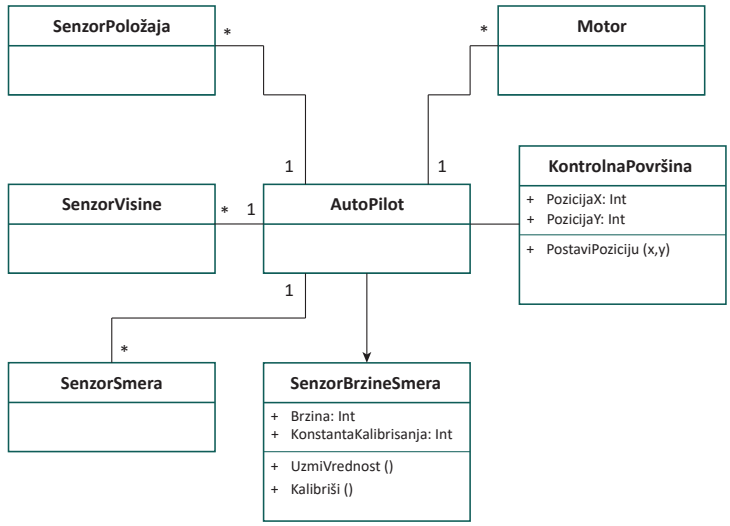
Asocijacija među klasama se označava punom linijom koja povezuje dijagrame klasa. Na svakom kraju te pune linije upisuje se broj instanci te klase koji je u nekoj vezi sa instancama druge klase. Zvezdica umesto broja označava veći broj. Usmerena strelica označava da instance klase u kojima strelica počinje sadrže instance klase u kojima se strelica završava, a da instance klase u kojim se strelica završava nemaju znanje da su deo te druge klase. Sledeći UML dijagram označava da jedana instanca klase *Meni* može da sadrži više instanci klase *Porudzbina*, a da više *Porudzbina* može da ima jedan *Meni*. Takodje jedna instanca klase *Porudzbina* može da sadrži više instanci klase *PredmetPorudzbine*.

Slika 24: UML strukturni dijagram



Sledeći UML dijagram prikazuje kod za realizaciju autopilota. Prema ovom dijagramu jedna instanca klase *Autopilot* sadrži više instanci klasa *SenzorPolozaja*, *SenzorVisine*, *SenzorSmera* i *Motor*. Klasa *Autopilot* je takođe povezana i sa klasom *KontrolnaPovrsina* i sadrži instance klase *SenzorBrzineSmera*.

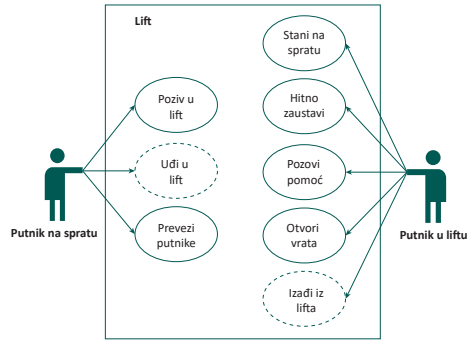
Slika 25: UML dijagram klasa za implementaciju autopilota



UML dijagram načina upotrebe

Ovi dijagrami prikazuju korisnike i punim linijama povezuju korisnike sa načinima korišćenja sistema. UML dijagram na sledećoj slici prikazuje način korišćenja lifta. Korisnici su putnik u liftu i putnik na spratu koji čeka list. Putnik u liftu može zahtevati otvaranje vrata lifta, tražiti hitno zaustavljanje lifta ili njegovo zaustavljanje na nekom spratu. Ovaj korisnik takođe može tražiti pomoć. On može izaći iz lifta, što je aktivnost koja je samo indirektno povezana sa načinom korišćenja lifta, zbog čega je i prikazana isprekidanom linijom. Putnik na spratu može pozvati lift i ući u lift (što opet nije aktivnost koja je direktno vezana sa načinom korišćenja lifta). Sam lift vrši funkciju prevoza putnika.

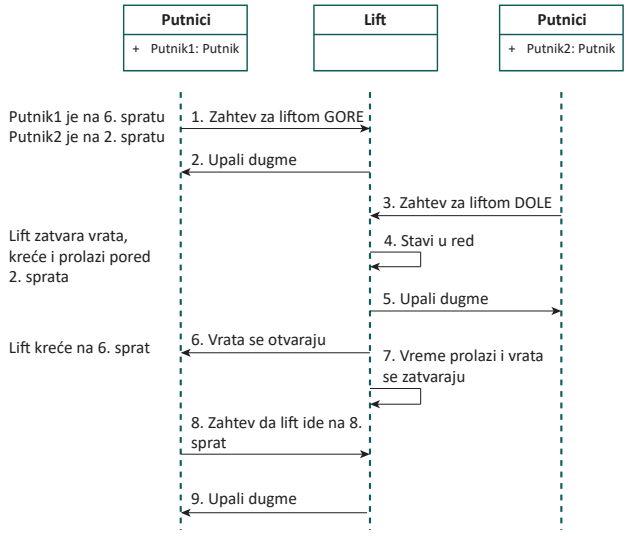
Slika 26: UML diagram upotrebe za primer lifta



UML sekvencijalni dijagram

Ova vrsta dijagrama služi za grafički prikaz sekvencijalnog sleda događaja u sistemu za rad u realnom vremenu. Oni najčešće služe za za prikaz vremenskog uređenja događaja i akcija. Ovi dijagrami se često koriste i kao deo dokumentacije kojim se daju početne specifikacije u projektovanju. U ovim dijagramima objekti su predstavljeni uz pomoć pravougaonika, a protok vremena isprekidanim linijama koje idu od objekata na dole. Komentari u vidu teksta stavljaju se na levu stranu dijagrama, a sekvencijalne poruke izmedju objekata prikazuju se usmerenim strelicama. Sekvencijalni dijagram za primer lifta dat je na sledećoj slici.

Slika 27: UML sekvencijalni dijagram za primer lifta

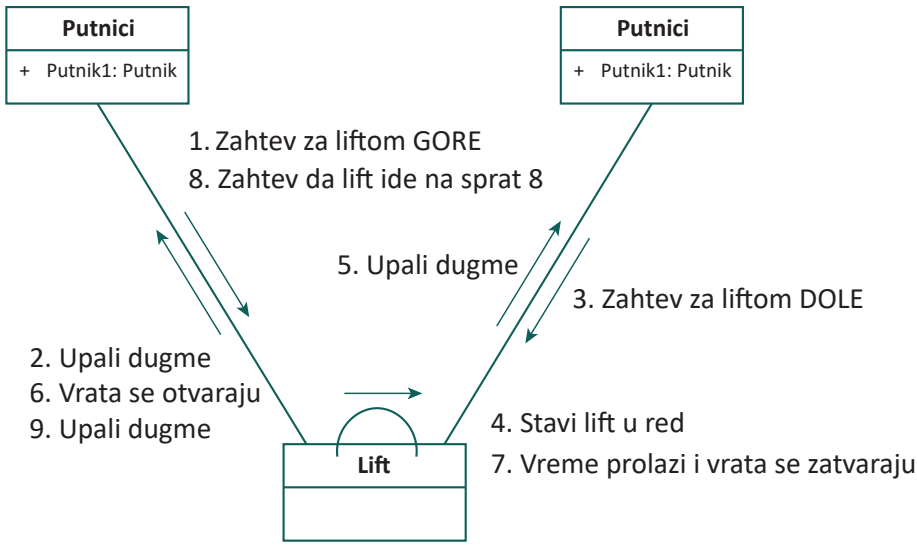


U ovom primeru postoje tri objekta *putnik1, putnik2* i *lift*. U početku *putnik1* zahteva kretanje lifta gore, a kao odgovor *lift* pali led diodu u dugmetu koje je pritisnuto i tako signalizira *putniku1* da je poziv primljen. Zatim *putnik2* šalje liftu poziv za kretanje na dole, lift ovaj poziv stavlja u red čekanja, a zatim pali led diodu u dugmetu koje je pritisnuto kao znak da je poziv prihvaćen. Sledeći događaj je otvaranje vrata za *putnika2* i čekanje lifta da se vrata zatvore. *Putnik1* izdaje zahtev da *lift* ide na sprat 2, a zatim *lift* pali led diodu koja potvrđuje da je ovaj poziv registrovan.

UML dijagram saradnje

Ovi dijagrami prikazuju kako različiti objekti u okviru sistema sarađuju. Objekti su prikazani u vidu pravougaonika, gde pune linije povezuju sve one objekte koji međusobno sarađuju. Poruke koje pojedini objekti šalju jedni drugima su u tekstualnom obliku ispisani pored tih linija i njihov smer je naznačen strelicama. Za primer semafora, dijagram saradnje u jednom slučaju korišćenja lifta dat je na sledećoj slici.

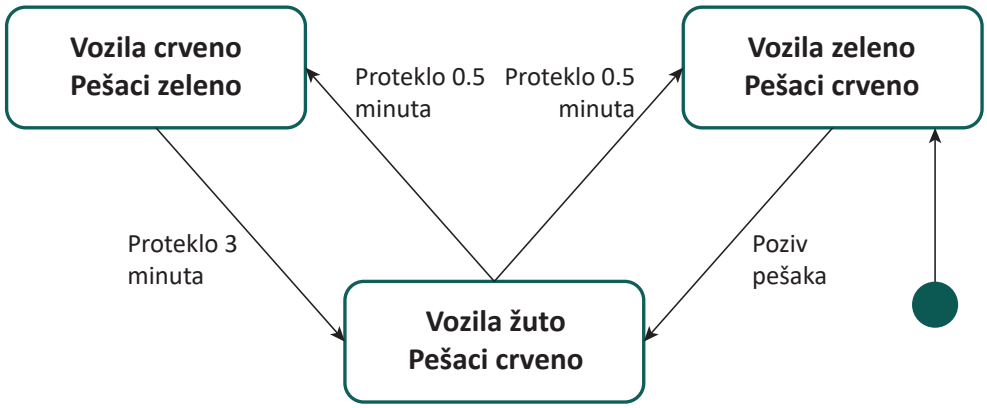
Slika 28: UML diagram saradnje za primer lifta



UML dijagram konačnog automata

Konačni automati o kojima je već ranije bilo reči takođe se mogu predstavljati uz pomoć UML dijagrama. Svako stanje automata je predstavljeno pravougaonikom, dok je prelaz sa jednog na drugo stanje prikazan strelicom i tekstom koji označava uslov za prelazak sa jednog stanja na drugo. UML dijagram konačnog automata semaforakoji je ranije opisan dat je na sledećoj slici.

Slika 29: UML dijagram konačnog automata semafor



III DEO

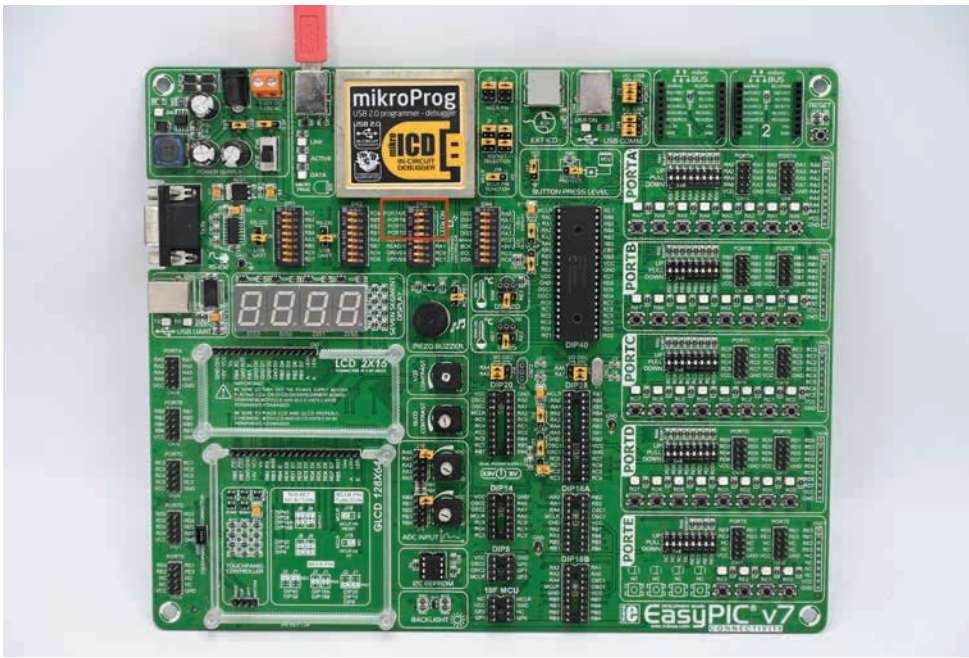
PRAKTIČNI PRIMERI

PRAKTIČNI PRIMERI

Najbolji način za potpuno usvajanje prethodno iznetih teoretskih postulata projektovanja sistema za rad u realnom vremenu jeste njihova praktična primena. U ovom delu knjige biće predstavljeni primeri implementacije osnovnih koncepata i principa projektovanja sistema za rad u realnom vremenu na razvojnoj ploči EasyPIC proizvođača Mikroelektronika. Vežbe se mogu raditi i na drugom razvojnim platformama uz određena prilagođenja.

1. TREPTANJE SVIH DIODA

Cilj ove vežbe je treptanje svih dioda na portovima A,B,C,D i E sa periodom od 1 sekunde. Na ploči je potrebno uključiti prekidače SW3.1., SW3.2., SW3.3. i SW3.4. Obratiti pažnju da se diode na pinovima RA6 i RA7 ne uključuju jer su ti pinovi rezervisani za oscilator kao i dioda na pinu RE3 jer taj pin može biti samo ulazni pin.

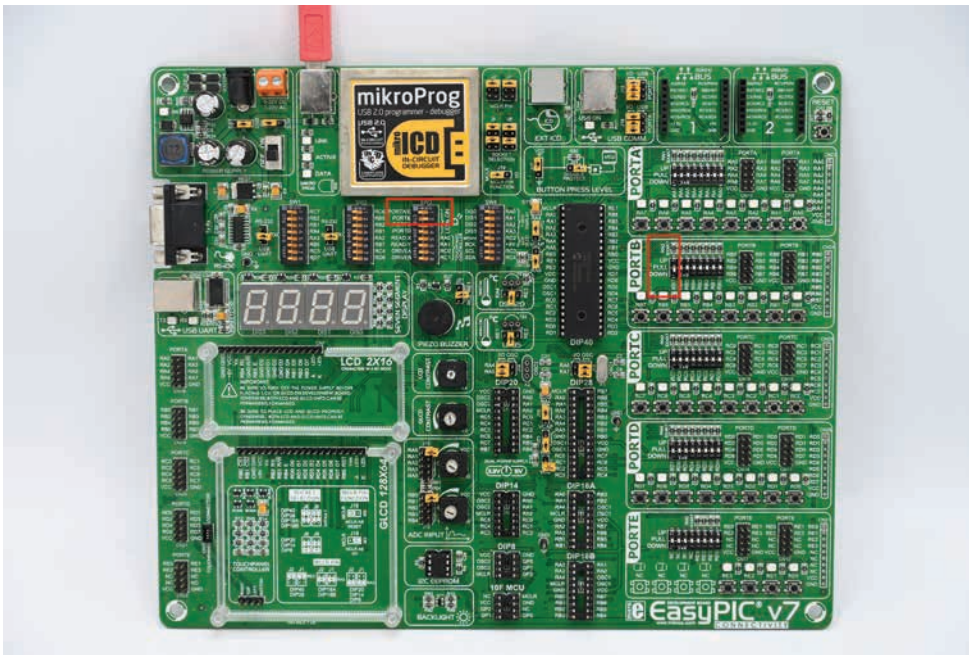


```
1.  void main() {
2.       TRISA = 0;
3.       TRISB = 0;
4.       TRISC = 0;
5.       TRISD = 0;
6.       TRISE = 0;
7.
8.       do {
9.        LATA = 0;
10.       LATB = 0;
11.       LATC = 0;
12.       LATD = 0;
13.       LATE = 0;
14.
15.       Delay_ms(1000);
16.
17.       LATA = 255;
18.       LATB = 255;
19.       LATC = 255;
20.       LATD = 255;
21.       LATE = 255;
22.
23.       Delay_ms(1000);
24.
25.      } while(1);
26. }
```

Postavljaju se svi pinovi PORTA, PORTB, PORTC, PORTD i PORTE kao izlazi koristeći registre TRISA, TRISB, TRISC, TRISD i TRISE. Ovo omogućava mikrokontroleru da kontroliše diode koje su povezane na ove pinove. U beskonačnoj petlji se izvršava sledeće: Svi pinovi se postavljaju na nisku logičku vrednost (0) koristeći registre LATA, LATB, LATC, LATD i LATE. Ovo isključuje sve diode. Zatim sledi pauza od 1000 milisekundi koristeći funkciju Delay_ms(1000). Ova funkcija čeka određeno vreme pre nego što se nastavi izvršavanje programa. Nakon pauze, svi pinovi se postavljaju na visoku logičku vrednost (1) koristeći registre LATA, LATB, LATC, LATD i LATE. Ovo uključuje sve diode. Ponavlja se pauza od 1000 milisekundi. Ova petlja se izvršava beskonačno, što znači da će diode neprekidno treptati u ciklusima od po jedne sekunde. Ukratko, ovaj kod stvara efekat treptanja svih dioda na mikrokontroleru tako što ih uključuje na jednu sekundu, zatim isključuje na jednu sekundu, i tako u beskonačnoj petlji.

2. KONTROLA DIODE PREKO DUGMETA

Cilj ove vežbe je kontrola diode na pinu 0 porta B pomoću dugmeta na pinu 7 porta B. Na ploči je potrebno uključiti prekidač SW3.2. , postaviti prekidač 7 na portu B u pull down položaj i uključiti Button biblioteku.



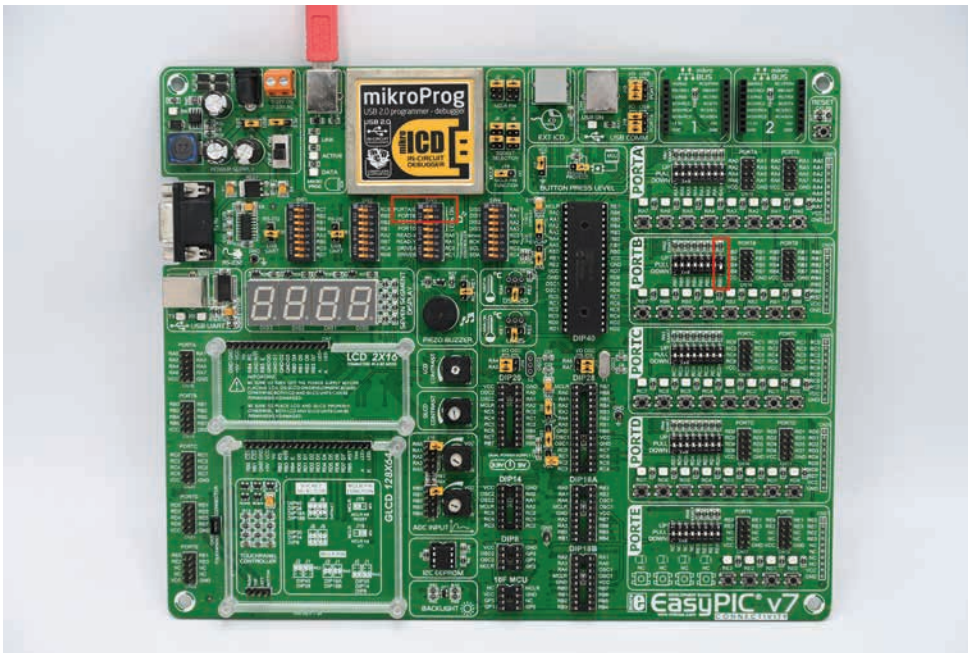

```
1.  bit oldstate;
2.
3.  void main() {
4.
5.      TRISB0_bit = 0;
6.      TRISB7_bit = 1;
7.
8.      LATB.F0 = 1;
9.      oldstate=0;
10.
11.     do {
12.         if (Button(&PORTB, 7, 1, 1)) {
13.             oldstate = 1;
14.         }
15.         if (oldstate && Button(&PORTB, 7, 1, 0)) {
```

```
16.             LATB.F0 = ~LATB.F0;
17.             oldstate = 0;
18.         }
19.     } while(1);
20. }
```

Na početku se postavlja pin RB0 kao izlaz (TRISB0_bit = 0) jer će na tom pinu biti spojena dioda koju želimo kontrolisati, dok se pin RB7 postavlja kao ulaz (TRISB7_bit = 1) jer će na tom pinu biti spojeno dugme, takođe postavlja se i početna vrednost za diodu - LATB.F0 se postavlja na 1, što znači da je dioda na početku uključena. Promenljiva oldstate se koristi za praćenje prethodnog stanja dugmeta. Na početku se postavlja na 0. Program ulazi u beskonačnu petlju koja izvršava sledeće: Proverava se da li je dugme pritisnuto koristeći funkciju Button() za koju je potrebno uključiti istoimenu biblioteku. Ako je dugme pritisnuto (tj. ako je na pinu RB7 detektovana visoka vrednost), oldstate se postavlja na 1. Ako je prethodno stanje bilo 1 (tj. ako je dugme bilo pritisnuto) i sada je dugme otpušteno (tj. ako je na pinu RB7 detektovana niska vrednost), to znači da je dugme pritisnuto i otpušteno. U tom slučaju, stanje diode se menja inverzijom trenutne vrednosti (koristeći ~LATB.F0), a zatim se oldstate resetuje na 0, spremajući se za sledeći pritisak dugmeta. Ova petlja se izvršava neprekidno, što nam omogućava da pritiskom na dugme kontrolišemo stanje diode. Ukratko, ovaj kod nam omogućava da pritiskom na dugme RB0 menjamo stanje diode između uključeno i isključeno.

3. KONTROLA DIODE PREKO DUGMETA SA PREKIDOM

Cilj ove vežbe je kontrola diode na pinu 7 porta B pomoću dugmeta na pinu 0 porta B koristeći prekid. Na ploči je potrebno uključiti prekidač SW3.2. Kao i postaviti prekidač 0 na portu B u pull down položaj.



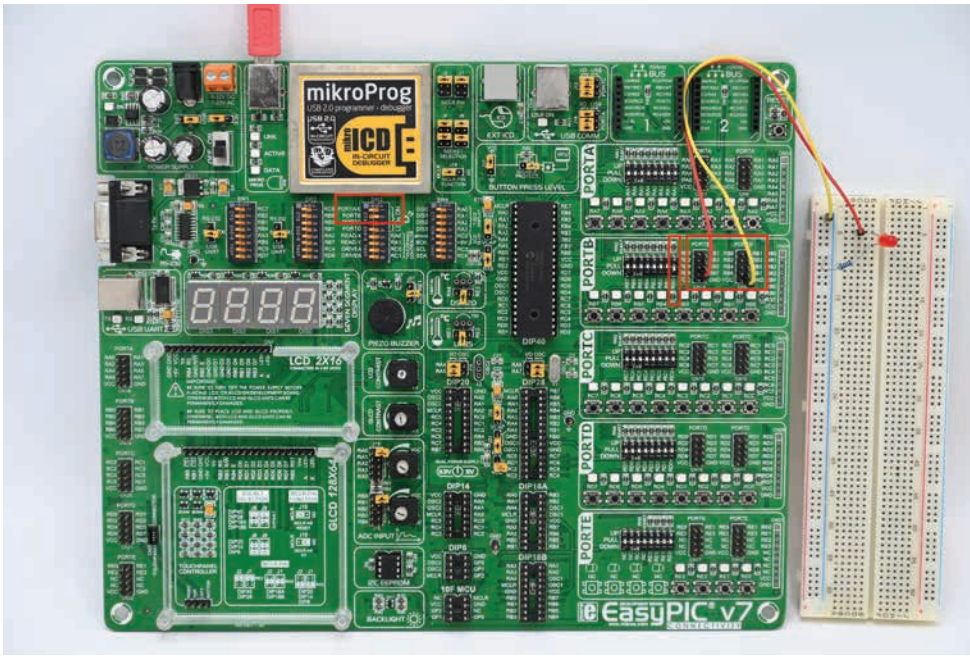

```
1.  void interrupt(){
2.   if(INT0IF_bit == 1 ) {
3.        INT0IF_bit = 0;
4.        LATB.F7 = ~LATB.F7;
5.   }
6.  }
7.  void main() {
8.   TRISB0_bit = 1;
9.   TRISB7_bit = 0;
10.  LATB.F7 = 1;
11.  ANSELB = 0;
12.
13.  INTEDG0_bit = 1;
14.  INT0IF_bit = 0;
```

```
15.  INT0IE_bit = 1;
16.  GIE_bit = 1;
17.  do {
18.  //empty
19.  } while(1);
20. }
```

Na početku konfigurišemo pinove, TRISB0_bit = 1; - Postavlja pin RB0 kao ulaz. Ovde je dugme spojeno. TRISB7_bit = 0; - Postavlja pin RB7 kao izlaz. Ovde je dioda spojena. LATB.F7 = 1; - Postavlja početno stanje diode na uključeno. ANSELB = 0; - Postavlja sve pinove na PORTB kao digitalne (isključuje analogne funkcije). Zatim konfigurišemo prekid, INTEDG0_bit = 1; - Postavlja prekid na rastuću ivicu signala na pinu INT0 (RB0). Prekid će se desiti kada se dugme pritisne i signal pređe iz niskog u visoko stanje. INT0IF_bit = 0; - Resetuje flag za prekid INT0, čime se osigurava da nema zaostalih prekida. INT0IE_bit = 1; - Omogućava prekid INT0. GIE_bit = 1; - Globalno omogućava prekide. Na kraju imamo rutinu prekida, void interrupt(){...} funkcija se izvršava kada dođe do prekida, if(INT0IF_bit == 1) { ... } - Proverava da li je prekid uzrokovan INT0 prekidom, INT0IF_bit = 0; - Resetuje INT0 flag kako bi se omogućilo prepoznavanje sledećeg prekida. LATB.F7 = ~LATB.F7; - Menja stanje diode na pinu RB7 ako se prekid desio(uključuje ili isključuje diodu). Ovaj kod nam omogućava da pritiskom na dugme povezano na pin RB0 menjamo stanje diode povezane na pin RB7. Kada se pritisne dugme, signal na pinu RB0 prelazi iz niskog u visoko stanje, što izaziva prekid. Rutina prekida menja stanje diode (uključuje je ili isključuje) svaki put kada se prekid desi. Korišćenje prekida nam donosi da mikrokontroler ne troši resurse na stalno proveravanje stanja dugmeta, već reaguje samo kada dođe do događaja (prekida) i takođe omogućava brži odziv odnosno brzu reakciju na pritisak dugmeta, što je korisno za aplikacije gde je važna brzina reakcije što je posebno bitno kada pričamo o sistemima za rad u realnom vremenu.

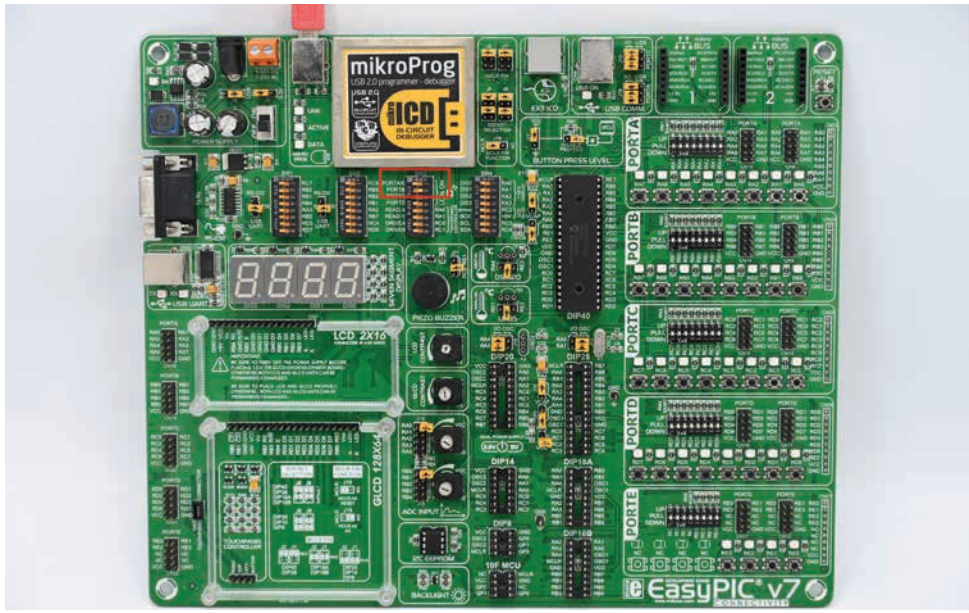
4. EKSTERNA DIODA

Cilj ove vežbe je kontrola eksterne diode povezane na pin 7 porta B pomoću dugmeta na pinu 0 porta B koristeći prekid. Potrebno je povezati diodu preko otpornika od 220 oma na pin RB7. Na ploči je potrebno uključiti prekidač SW3.2. Kao i postaviti prekidač 0 na portu B u pull down položaj. Programski kod je isti kao u prethodnom zadatku.



5. TREPTANJE DIODE – KAŠNJENJE

Cilj ove vežbe je napisati program kako bi dioda na pinu 7 porta B treptala sa određenom periodom od 250ms koristeći kašnjenja. Na ploči je potrebno uključiti prekidač SW3.2.

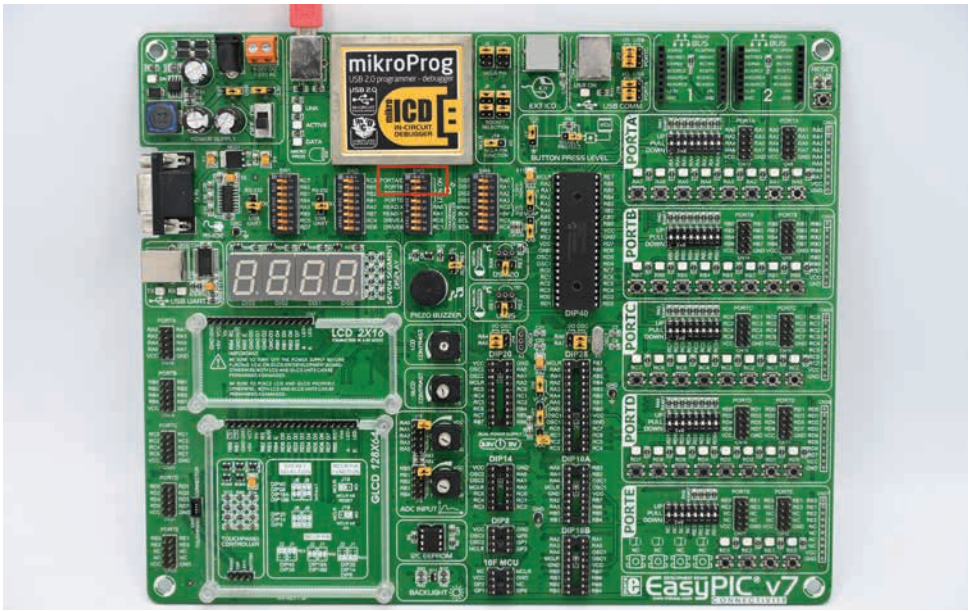



```
1.  void main() {
2.   TRISB7_bit = 0;
3.   LATB.F7 = 1;
4.   do {
5.      LATB.F7 = ~LATB.F7;
6.      Delay_ms(250);
7.   } while(1);
8.  }
```

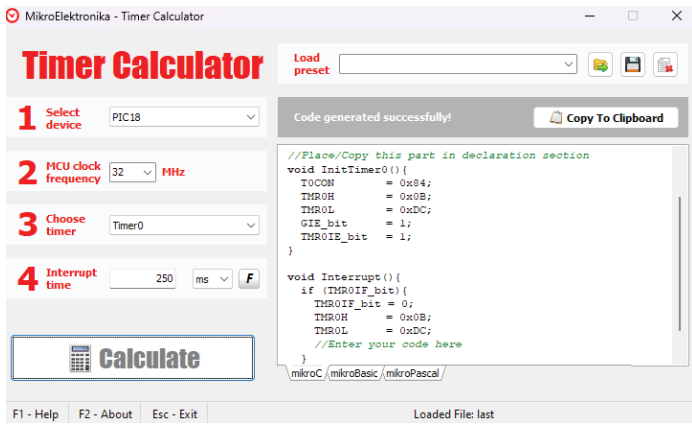
Prvo imamo konfiguraciju pina, postavlja se RB7 pin na kom koristimo diodu kao izlaz komandom TRISB7_bit = 0;. Nakon toga LATB.F7 = 1; - Postavlja početno stanje diode na uključeno. U beskonačnoj petlji omogućavamo kontinuirano treptanje diode, LATB.F7 = ~LATB.F7; - komanda invertuje trenutno stanje diode. Ako je dioda uključena, isključuje je, i obrnuto i takođe koristimo Delay_ms(250); - Ovo nam omogućava čekanje od 250 milisekundi pre nego što nastavi sa izvršavanjem sledeće instrukcije. Odnosno ova funkcija dodaje kašnjenje između promene stanja diode. Ovaj kod implementira treptanje diode povezane na pin RB7 sa kašnjenjem od 250 milisekundi između svake promene stanja. Dioda se neprekidno uključuje i isključuje u intervalu od 250 milisekundi.

6. TREPTANJE DIODE – SA TAJMEROM

Cilj ove vežbe je proširenje prethodnog zadatka koristeći tajmer za egzaktno generisanje prekida sa periodom od 250 milisekundi. Na ploči je potrebno uključiti prekidač SW3.2.



Generisanje koda za egzaktan tajmer sa periodom od 250ms može se izvršiti uspomoć Timer Calculator alata.

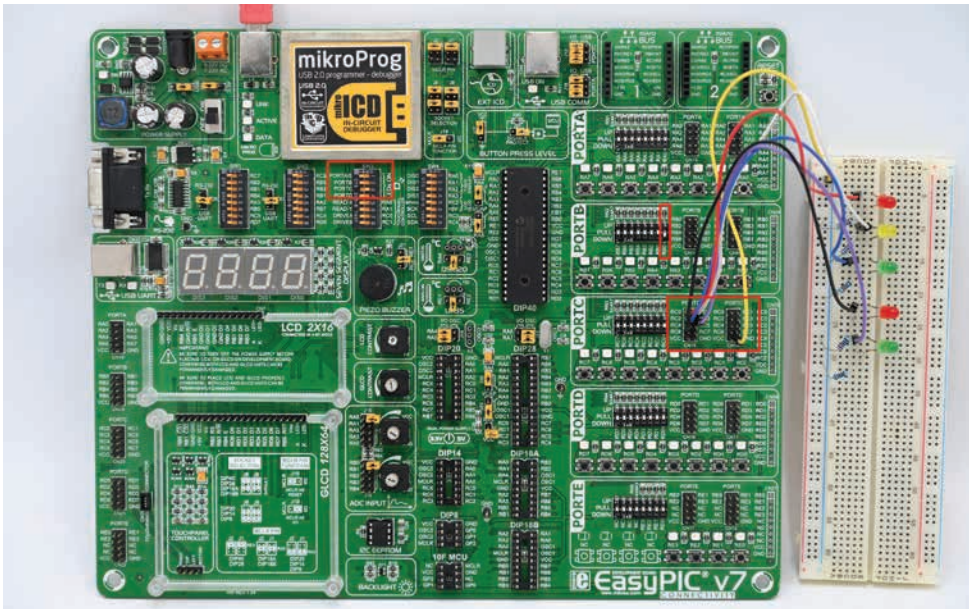


```
1.  void interrupt() {
2.   if (TMR0IF_bit) {
3.      TMR0IF_bit = 0;
4.      TMR0H = 0x0B;
5.      TMR0L = 0xDC;
6.      LATB.F7 = ~LATB.F7;
7.   }
8.  }
9.  void main() {
10.  TRISB7_bit = 0;
11.  LATB.F7 = 1;
12.  T0CON = 0x84;
13.  TMR0H = 0x0B;
14.  TMR0L = 0xDC;
15.  GIE_bit = 1;
16.  TMR0IE_bit = 1;
17.  do {
18.  } while(1);
19. }
```

Konfiguracija glavne funkcije: TRISB7_bit = 0; - Postavlja pin RB7 kao izlaz. Ovde je dioda spojena. LATB.F7 = 1; - Postavlja početno stanje diode na uključeno. GIE_bit = 1; - Globalno omogućava prekide. TMR0IE_bit = 1; - Omogućava prekid tajmera TMR0. Zatim pomoću aplikacije Timer Calculator možemo generisati kod za tajmer sa rutinom prekida od 250ms. InitTimer0 funkciju kopiramo u našu main funkciju dok Interrupt funkciju kopiramo u našu funkciju prekida. Funkcija prekida void interrupt() { ... } -, se poziva kada se desi prekid. Sa komandom if (TMR0IF_bit) { ... } - Proverava se da li je prekid uzrokovan tajmerom TMR0 (TMR0IF_bit). Ako jeste TMR0IF_bit = 0; - Resetuje se flag za prekid TMR0 kako bi se omogućilo prepoznavanje sledećeg prekida i izvršava se LATB.F7 = ~LATB.F7; - Menja se stanje diode na pinu RB7 (uključuje ili isključuje diodu). Ovaj kod omogućava treptanje diode povezane na pin RB7 koristeći tajmer TMR0 za generisanje prekida. Tajmer je konfigurisan da generiše prekid nakon vremenskog intervala od 250 milisekundi, a svaki put kada se prekid dogodi, stanje diode se menja (uključuje ili isključuje). Tajmer nam omogućava precizno određivanje vremenskih intervala za treptanje diode, oslobađa mikrokontroler od stalnog proveravanja stanja i omogućava automatsko reagovanje na događaje bez potrebe za stalnom pažnjom glavne petlje programa.

7. SEMAFOR

Cilj ove vežbe je realizacija prostog sistema semafora koji se sastoji od semafora za vozila (crvena, žuta i zelena dioda) i semafora za pešake (crvena i zelena dioda) i tastera za pešake kojim oni mogu da zatraže paljenje svetla za pešake. Ukoliko nema pešaka, treba da bude zeleno za vozila i crveno za pešake. Ukoliko neki pešak pritisne taster i zahteva prelaz ulice, potrebno je da se promeni svetlo za automobile tako što se prvo pali žuto svetlo u trajanju od 2 sekunde, a zatim i crveno. Tek po paljenju crvenog svetla za vozila, treba upaliti zeleno svetlo za pešake. Ovakvo stanje u kome se propuštaju pešaci treba da traje 10 sekundi, nakon čega se prvo pali crveno svetlo za pešake, a potom i prebacuje svetlo za vozila na zeleno, paljenjem žutog svetla u međuvremenu u trajanju od 2 sekunde. Novi zahtev za prelaz pešaka može izazvati promenu svetla tek nakon 10 sekundi od prethodno završene promene svetla. Zahtevi za prelazak pešaka koji nastaju dok je zeleno svetlo za pešake treba da se ignorišu. Zahtev pešaka treba da bude implementiran uz pomoć prekida, dok vremenska trajanja različitih svetala treba da budu implementirana uz pomoć tajmera. Na ploči je potrebno uključiti prekidač SW3.3. i postaviti prekidač 0 na portu B u pull down položaj. Diode je potrebno povezati putem otpornika od 220 oma na pinove RC0, RC1, RC2, RC3 i RC4.



```
1.  #define CARGR 0
2.  #define CARYL 1
3.  #define CARRD 2
4.  #define CARYLRD 3
5.
6.  bit flag;
7.  int count;
8.  int state;
9.
10. void interrupt() {
11.  if (TMR0IF_bit) {
12.     TMR0IF_bit = 0;
13.     TMR0H = 0x0B;
14.     TMR0L = 0xDC;
15.     count++;
16.  }
17.  if(INT0IF_bit == 1 ) {
18.     flag = 1;
19.     INT0IF_bit = 0;
20.  }
21. }
22.
23. void main() {
24.  TRISB0_bit = 1;
25.  TRISC = 0x00;
26.  LATC= 0x00;
27.  LATC.F0=1;
28.  ANSELB = 0;
29.  T0CON = 0x84;
30.  TMR0H = 0x0B;
31.  TMR0L = 0xDC;
32.  TMR0IE_bit = 1;
33.  INTEDG0_bit = 1;
34.  INT0IF_bit = 0;
35.  INT0IE_bit = 1;
36.  GIE_bit = 1;
37.
38.  state=CARGR;
39.  count=0;
40.  do {
41.
42.     switch (state){
43.        case CARGR :
44.            LATC.F0 = 1;
45.            LATC.F1 = 0;
46.            LATC.F2 = 0;
```

```
47.             LATC.F3 = 0;
48.             LATC.F4 = 1;
49.             if (flag && count>=10){
50.                 state=CARYL;
51.                 count = 0;
52.                 flag=0;
53.             }
54.         break;
55.         case CARYL:
56.             LATC.F3 = 1;
57.             LATC.F4 = 0;
58.             if (count>=2){
59.                 state = CARRD;
60.                 count=0;
61.             }
62.         break;
63.         case CARRD:
64.             LATC.F0 = 0;
65.             LATC.F1 = 1;
66.             LATC.F2 = 1;
67.             LATC.F2 = 1;
68.             LATC.F3 = 0;
69.             if (count>=10){
70.                 state = CARYLRD;
71.                 count=0;
72.             }
73.         break;
74.         case CARYLRD:
75.             LATC.F0 = 1;
76.             LATC.F1 = 0;
77.             LATC.F2 = 1;
78.             LATC.F3 = 1;
79.             if (count>=2){
80.                 state = CARGR;
81.                 count=0;
82.             }
83.         break;
84.     }
85.  } while(1);
86. }
```

Prikazani kod je implementacija semafora koji kontroliše svetlosne signale za vozila, koristeći prekide i tajmer na mikrokontroleru. Semafor ima četiri stanja: zeleno svetlo za vozila – crveno za pešake, žuto svetlo za vozila – crveno za pešake, crveno svetlo za vozila – zeleno svetlo za pešake i kombinaciju crvenog i žutog svetla za vozila – crveno za pešake. Na početku imamo definicije stanja i globalnih promenljivih:

#define CARGR 0 - Definiše stanje kada je zeleno svetlo za vozila

#define CARYL 1 - Definiše stanje kada je žuto svetlo za vozila.

#define CARRD 2 - Definiše stanje kada je crveno svetlo za vozila.

#define CARYLRD 3 - Definiše stanje kada su crveno i žuto svetlo za vozila upaljena.

bit flag; - Flag koji se postavlja kada se pritisne dugme (prekid INT0).

int count; - Brojač koji se koristi za tajming stanja.

int state; - Trenutno stanje semafora.

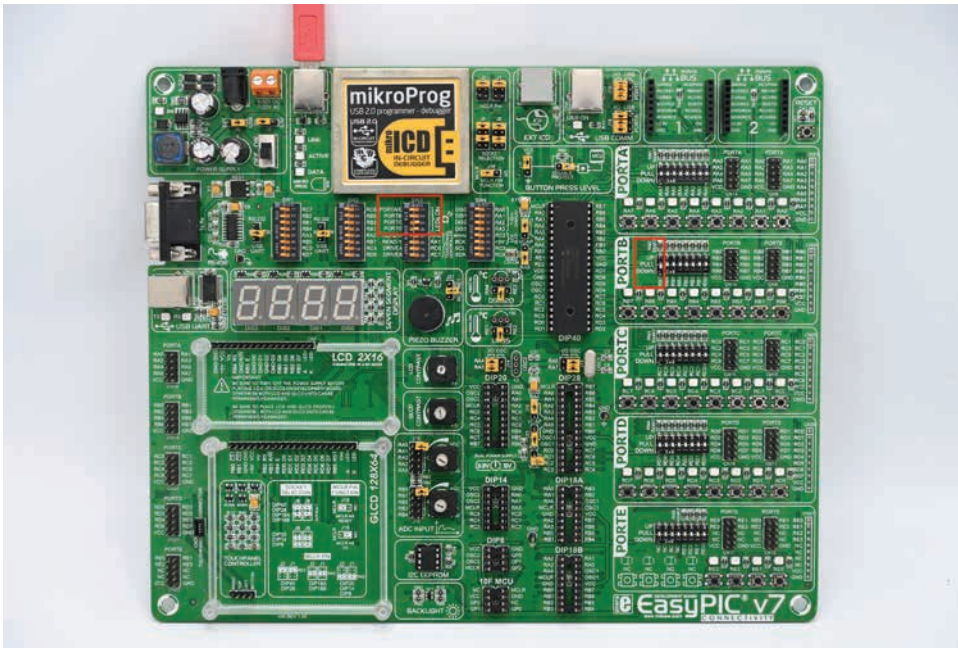
Rutina prekida void interrupt() { ... } koja se poziva kada se desi prekid, if (TMR0IF_bit) { ... } - Proverava se da li je prekid uzrokovan tajmerom TMR0, TMR0IF_bit = 0; - Ako jestre, resetuje se flag tajmera, TMR0H = 0x0B; TMR0L = 0xDC; - Postavljaju se vrednosti TMR0 registara za inicijalizaciju tajmera, count++; - Povećava se brojač. if(INT0IF_bit == 1) { ... } - Proverava se da li je prekid uzrokovan INT0 prekidom (dugme), Ako jeste, flag = 1; - Postavlja se flag za detekciju pritiska dugmeta, INT0IF_bit = 0; - Resetuje se flag INT0 prekida. U glavnoj funkciji konfigurisemo pinove, TRISB0_bit = 1; - Postavlja pin RB0 kao ulaz (dugme). TRISC = 0x00; - Postavlja sve pinove PORTC kao izlaze (za kontrolu svetala). LATC= 0x00; - Resetuje sve pinove PORTC na nisku vrednost. LATC.F0=1; - Uključuje zeleno svetlo za vozila kao početno stanje. ANSELB = 0; - Postavlja sve pinove PORTB kao digitalne.

Zatim imamo i konifguraciju tajmera i INT0 prekida, T0CON = 0x84; - Konfiguriše tajmer TMR0, TMR0IE_bit = 1; - Omogućava prekid tajmera TMR0. TMR0H = 0x0B; TMR0L = 0xDC; - Postavlja početne vrednosti tajmera uspomoć tajmer kalkulatora na 1 sekundu. INTEDG0_bit = 1; - Postavlja prekid na rastuću ivicu signala na pinu INT0 (RB0). INT0IF_bit = 0; - Resetuje flag za prekid INT0. INT0IE_bit = 1; - Omogućava prekid INT0. GIE_bit = 1; - Globalno omogućava prekide. Nakon konfiguracije, inicijalizacija promenljivih state = CARGR; postavlja početno stanje da bude zeleno svetlo za vozila i crveno za pešake i count = 0; resetuje brojač. U beskonačnoj petlji imamo switch koji menja stanje semafora na osnovu trenutnog stanja. CARGR: - Zeleno svetlo za vozila i crveno za pešake (LATC.F0 = 1), čekanje na pritisak dugmeta i prelaz na žuto svetlo za vozila ako je dugme pritisnuto i count >= 10 što označava da je prošlo 10 sekundi pošto nam se count promenljiva uvećava za 1 na osnovu tajmera odnosno označava da je prošla 1 sekunda. CARYL: - Žuto svetlo za vozila i crveno za pešake (LATC.F3 = 1), prelaz na crveno svetlo za vozila i zeleno za pešake nakon 2 brojača pošto ovo stanje traje 2 sekunde. CARRD: - Crveno svetlo za vozila i zeleno za pešake (LATC.F1 = 1 i LATC.F2 = 1), prelaz na kombinaciju crvenog i žutog svetla za vozila nakon 10 brojača. CARYLRD: - Crveno i žuto svetlo

za vozila i crveno za pešake (LATC.F2 = 1 i LATC.F3 = 1), prelaz na zeleno svetlo za vozila nakon 2 brojača. Ukratko, ovaj kod implementira jednostavan semafor koji koristi prekide i tajmer za upravljanje svetlosnim signalima za vozila. Semafor ima četiri stanja i menja svetla na osnovu tajmera i pritiska dugmeta. Tajmer generiše prekid svakih 1 sekundu, a dugme generiše prekid kada je pritisnuto. U glavnoj petlji, semafor menja stanje na osnovu brojača i flagova.

8. NAIZMENIČNO PALJENJE DIODA NA DVA PORTA

Cilj ove vežbe je naizmenično paljenje svih dioda na portu D i portu C pritiskom na dugme RB7, preciznije nakon otpuštanja dugmeta, drugim rečima detekcija silazne ivice. Na ploči je potrebno uključiti prekidače SW3.3. i SW3.4. i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.



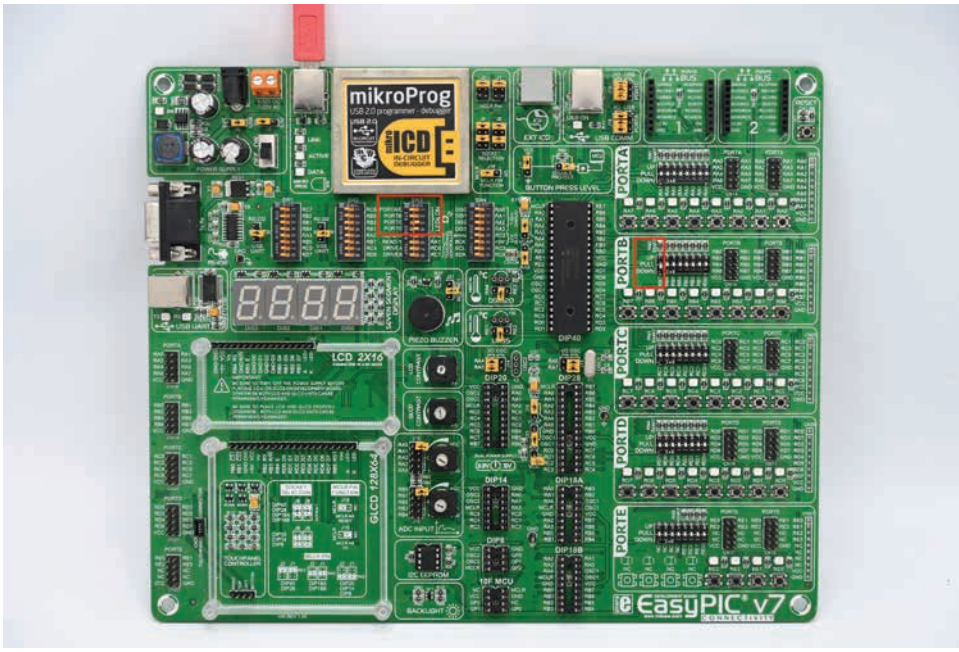

```
1.  bit oldstate;
2.
3.  void main() {
4.     TRISB7_bit = 1;
5.
6.     TRISD = 0x00;
7.     TRISC = 0x00;
8.     LATD = 0xFF;
9.     LATC = 0x00;
10.
11.    oldstate = 0;
12.
13.    do {
```

```
14.       if(Button(&PORTB, 7, 1, 1)) {
15.          oldstate = 1;
16.       }
17.       if(oldstate && Button(&PORTB, 7, 1, 0)){
18.          LATD = ~LATD;
19.          LATC = ~LATC;
20.          oldstate = 0;
21.       }
22.    } while(1);
23. }
```

Definisanje i inicijalizacija promenljivih: bit oldstate; - Definiše promenljivu koja će čuvati prethodno stanje tastera. Glavna funkcija: void main() { … } - Početak glavne funkcije programa. Konfiguracija pinova: TRISB7_bit = 1; - Postavlja pin RB7 kao ulaz (gde je povezan taster). TRISD = 0x00; - Postavlja sve pinove PORTD kao izlaze. TRISC = 0x00; - Postavlja sve pinove PORTC kao izlaze. LATD = 0xFF; - Postavlja početno stanje svih pinova PORTD na visoko (sve diode uključene). LATC = 0x00; - Postavlja početno stanje svih pinova PORTC na nisko (sve diode isključene). Inicijalizacija promenljive: oldstate = 0; - Inicijalizuje promenljivu oldstate na 0. Beskonačna petlja: do { ... } while(1); - Početak beskonačne petlje koja omogućava kontinuirano izvršavanje programa. Detekcija pritiska tastera: if(Button(&PORTB, 7, 1, 1)) { oldstate = 1; } - Proverava da li je taster na pinu RB7 pritisnut. Ako jeste, postavlja oldstate na 1. Detekcija otpuštanja tastera: if(oldstate && Button(&PORTB, 7, 1, 0)){ ... } - Proverava da li je taster otpušten. Ako je oldstate 1 (taster je bio pritisnut) i taster je sada otpušten, izvršava sledeće instrukcije: LATD = ~LATD; - Invertuje trenutno stanje svih pinova PORTD (ako su sve diode bile uključene, sada će biti isključene i obrnuto). LATC = ~LATC; - Invertuje trenutno stanje svih pinova PORTC (ako su sve diode bile isključene, sada će biti uključene i obrnuto). oldstate = 0; - Resetuje oldstate na 0, pripremajući se za sledeći ciklus detekcije pritiska tastera. Ukratko, ovaj kod omogućava naizmenično paljenje dioda na PORTD i PORTC koristeći taster povezan na pin RB7. Kada se taster pritisne i otpusti, diode na PORTD se isključuju (ako su bile uključene) i uključuju (ako su bile isključene), dok diode na PORTC rade suprotno.

9. NAIZMENIČNO PALJENJE DIODA NA DVA PORTA SA TREPTANJEM

Cilj ove vežbe je naizmenično paljenje svih dioda na portu D i portu C pritiskom na dugme RB7 sa treptanjem, preciznije nakon otpuštanja dugmeta, drugim rečima detekcija silazne ivice. Upaljeni port treba takođe da trepti, dok drugi port treba da bude ugašen. Na ploči je potrebno uključiti prekidače SW3.3. i SW3.4. i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  bit oldstate;
2.  bit activeport;
3.  void main() {
4.    TRISB7_bit = 1;
5.
6.    TRISD = 0x00;
7.    TRISC = 0x00;
8.    LATD = 0xFF;
9.    LATC = 0x00;
10.   activeport = 1;
11.
12.
```

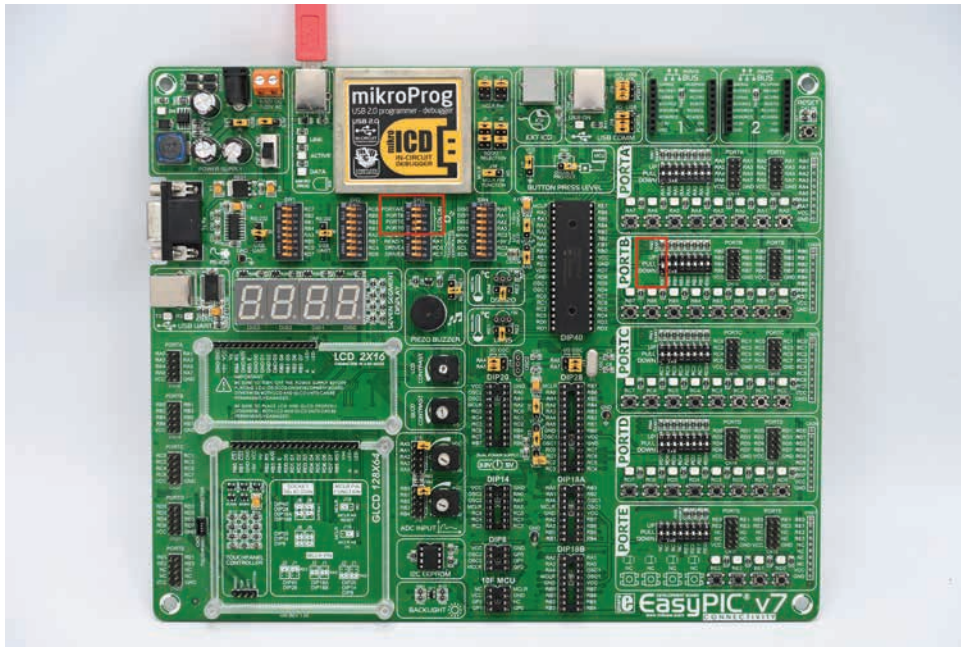
```
13.   oldstate = 0;
14.
15.   do {
16.     if(Button(&PORTB, 7, 1, 1)) {
17.       oldstate = 1;
18.     }
19.     if(oldstate && Button(&PORTB, 7, 1, 0)){
20.        if(activeport){
21.         LATD = 0x00;
22.         LATC = 0xFF;
23.        } else {
24.         LATC = 0x00;
25.         LATD = 0xFF;
26.        }
27.        activeport = ~activeport;
28.        oldstate=0;
29.     }
30.     if(activeport){
31.         LATD = ~LATD;
32.     }  else{
33.      LATC = ~LATC;
34.     }
35.     Delay_ms(50);
36.   } while(1);
37. }
```

Definisanje i inicijalizacija promenljivih: bit oldstate; - Definiše promenljivu koja će čuvati prethodno stanje tastera. bit activeport; - Definiše promenljivu koja označava koji port je trenutno aktivan. Glavna funkcija: void main() { ... } - Početak glavne funkcije programa. Konfiguracija pinova: TRISB7_bit = 1; - Postavlja pin RB7 kao ulaz (gde je povezan taster). TRISD = 0x00; - Postavlja sve pinove PORTD kao izlaze. TRISC = 0x00; - Postavlja sve pinove PORTC kao izlaze. LATD = 0xFF; - Postavlja početno stanje svih pinova PORTD na visoko (sve diode uključene). LATC = 0x00; - Postavlja početno stanje svih pinova PORTC na nisko (sve diode isključene). Inicijalizacija promenljivih: activeport = 1; - Postavlja početno aktivni port na PORTD. oldstate = 0; - Inicijalizuje promenljivu oldstate na 0. Beskonačna petlja: do { ... } while(1); - Početak beskonačne petlje koja omogućava kontinuirano izvršavanje programa. Detekcija pritiska tastera: if(Button(&PORTB, 7, 1, 1)) { oldstate = 1; } - Proverava da li je taster na pinu RB7 pritisnut. Ako jeste, postavlja oldstate na 1. Detekcija otpuštanja tastera: if(oldstate && Button(&PORTB, 7, 1, 0)){ ... } - Proverava da li je taster otpušten. Ako je oldstate 1 (taster je bio pritisnut) i taster je sada otpušten, izvršava sledeće instrukcije: if(activeport){ LATD = 0x00; LATC = 0xFF; } - Ako je PORTD bio aktivan, isključuje sve diode na PORTD i uključuje sve diode na PORTC. else { LATC = 0x00; LATD = 0xFF; } - Ako PORTD nije bio aktivan, isključuje sve diode na PORTC i uključuje sve diode na PORTD. activeport = ~activeport; - Invertuje promenljivu activeport kako bi označila promenjen aktivni port. oldstate = 0; - Resetuje oldstate na 0,

pripremajući se za sledeći ciklus detekcije pritiska tastera. Treptanje dioda: if(activeport) { LATD = ~LATD; } - Ako je PORTD aktivan, invertuje trenutno stanje svih pinova PORTD (diode trepere). else { LATC = ~LATC; } - Ako PORTC nije aktivan, invertuje trenutno stanje svih pinova PORTC (diode trepere). Delay_ms(50); - Zadržava trenutnu situaciju na 50 milisekundi kako bi se postigao efekat treptanja. Ukratko, ovaj kod omogućava naizmenično paljenje dioda na PORTD i PORTC koristeći taster povezan na pin RB7. Kada se taster pritisne i otpusti, diode na jednom portu se uključuju i trepere, dok se diode na drugom portu isključuju i obrnuto.

10. INKREMENTALNO PALJENJE KOLONA DIODA

Cilj ove vežbe je paljenje kolona pritiskom na dugme RB7, nakon svakog pritiska na dugme pali se sledeća kolona a gasi prethodna (PORT C – PORT D). Na ploči je potrebno uključiti prekidače SW3.3. i SW3.4. i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  bit oldstate;
2.  int counter;
3.
4.  void wait(){
5.     Delay_ms(50);
6.  }
7.  void main(){
8.     TRISB7_bit = 1;
9.
10.    TRISC = 0x00;
11.    TRISD = 0x00;
12.
13.    LATC = 0x00;
```

```
14.    LATD = 0x00;
15.    oldstate = 0;
16.    counter = 0;
17.
18.    do {
19.       if(Button(&PORTB, 7, 1, 1)){
20.          oldstate = 1;
21.       }
22.
23.       if(oldstate && Button(&PORTB, 7, 1, 0)){
24.          counter++;
25.          oldstate = 0;
26.          counter = counter%4;
27.       }
28.       LATC = 0x11 << counter;
29.       LATD = 0x11 << counter;
30.    } while(1);
31. }
```

Promenljiva counter je inicijalno 0, a uvecava se nakon svakog pritiska dugmeta i racuna se njegov ostatak pri celobrojnom deljenju sa brojem 4. Vrednosti portova su inicijalno, kada je counter 0, Ox 11 u heksadecimalnom zapisu, sto znaci da su diode 0 i 4 na svim portovima na visokom naponskom nivou, nakon toga svakim pritiskom na dugme, inicijalna vrednost se bitski pomera ulevo.

00010001 « 1 -> 00100010

00010001 « 2 -> 01000100

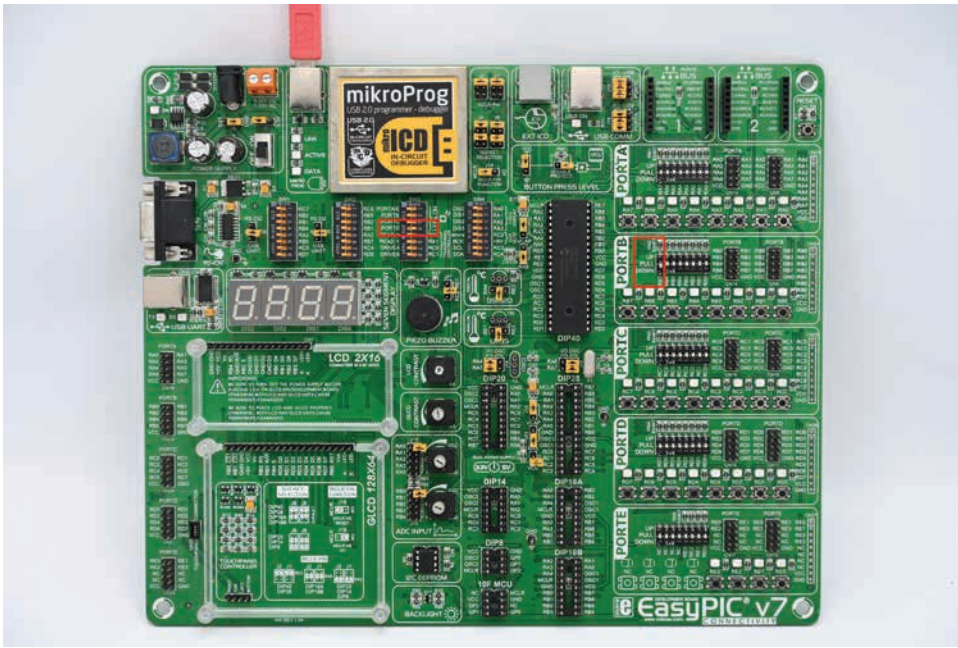
00010001 « 3 ->10001000

Definisanje i inicijalizacija promenljivih: bit oldstate; - Definiše promenljivu koja će čuvati prethodno stanje tastera. int counter; - Definiše brojač koji prati trenutnu poziciju aktivne kolone dioda. Glavna funkcija: void main(){ ... } - Početak glavne funkcije programa. Konfiguracija pinova: TRISB7_bit = 1; - Postavlja pin RB7 kao ulaz (gde je povezan taster). TRISC = 0x00; - Postavlja sve pinove PORTC kao izlaze. TRISD = 0x00; - Postavlja sve pinove PORTD kao izlaze. Inicijalizacija izlaznih registara i promenljivih: LATC = 0x00; - Postavlja početno stanje svih pinova PORTC na nisko (sve diode isključene). LATD = 0x00; - Postavlja početno stanje svih pinova PORTD na nisko (sve diode isključene). oldstate = 0; - Inicijalizuje promenljivu oldstate na 0. counter = 0; - Inicijalizuje brojač counter na 0. Beskonačna petlja: do { ... } while(1); - Početak beskonačne petlje koja omogućava kontinuirano izvršavanje programa. Detekcija pritiska tastera: if(Button(&PORTB, 7, 1, 1)){ oldstate = 1; } - Proverava da li je taster na pinu RB7 pritisnut. Ako jeste, postavlja oldstate na 1. Detekcija otpuštanja tastera: if(oldstate && Button(&PORTB, 7, 1, 0)){ ... } - Proverava da li je taster otpušten. Ako je oldstate 1 (taster je bio pritisnut) i taster je sada otpušten, izvršava sledeće instrukcije: counter++; - Povećava vrednost brojača

za 1. oldstate = 0; - Resetuje oldstate na 0, pripremajući se za sledeći ciklus detekcije pritiska tastera. counter = counter % 4; - Ograničava vrednost brojača na opseg od 0 do 3, kako bi se obezbedilo kružno pomeranje. Pomeranje kolona dioda: LATC = 0x11 << counter; - Pomeranje kolone dioda na PORTC. Vrednost 0x11 (binarno 00010001) se pomera ulevo za counter mesta, što rezultira aktivacijom određene kolone. LATD = 0x11 << counter; - Pomeranje kolone dioda na PORTD na isti način kao i za PORTC. Ukratko, ovaj kod omogućava inkrementalno pomeranje kolona dioda na PORTC i PORTD koristeći taster povezan na pin RB7. Kada se taster pritisne i otpusti, kolona dioda se pomera za jedno mesto dok se ostale diode isključuju.

11. INKREMENTALNO PALJENJE KOLONA DIODA NA JEDNOM PORTU KORISTEĆI DUGME

Cilj ove vežbe je a paljenje određene diode na portu D, shodno pritiskanju tastera RB7, drugim rečima, pritiskom na dugme pali se sledeća dioda u nizu na portu D i tako u krug. Na ploči je potrebno uključiti prekidač SW3.4. i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.



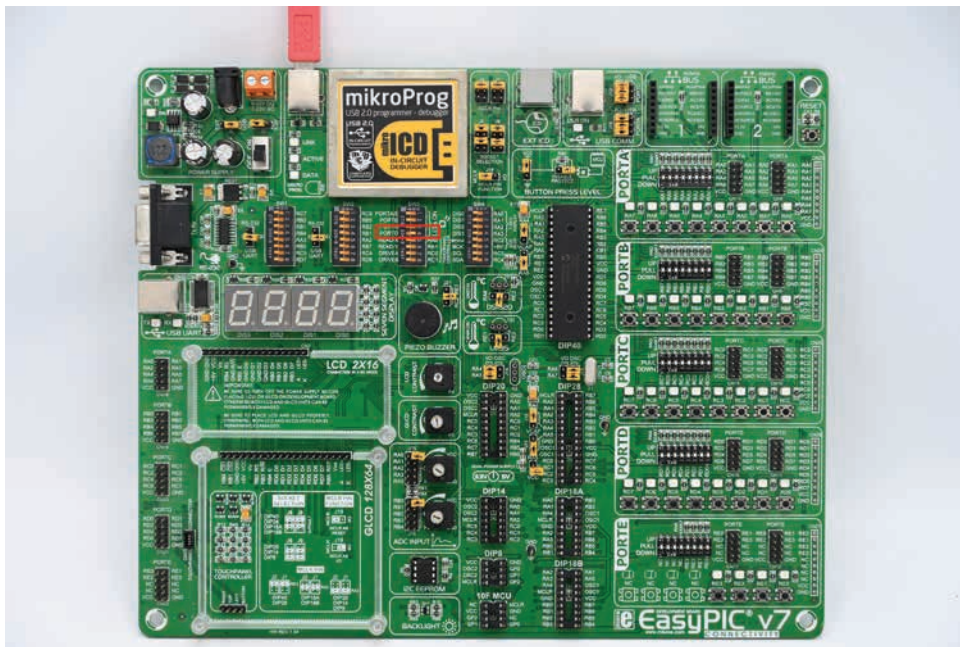

```
1.  bit oldstate;
2.  int counter;
3.
4.  void wait(){
5.     Delay_ms(100);
6.  }
7.  void main(){
8.     TRISB7_bit = 1;
9.
10.    TRISD = 0x00;
11.    LATD = 0x00;
```

```
12.
13.    oldstate = 0;
14.    counter = 0;
15.
16.    do {
17.       if(Button(&PORTB, 7, 1, 1)){
18.          oldstate = 1;
19.       }
20.
21.       if(oldstate && Button(&PORTB, 7, 1, 0)){
22.          counter++;
23.          oldstate = 0;
24.          counter = counter%8;
25.       }
26.       LATD = 0x01 << counter;
27.    } while(1);
28. }
```

Definisanje i inicijalizacija promenljivih: bit oldstate; - Promenljiva koja beleži prethodno stanje tastera, kako bi se detektovalo kada je taster otpušten. int counter; - Brojač koji prati trenutno aktivnu poziciju svetleće diode na portu D. Glavna funkcija: void main(){ ... } - Ovo je glavna funkcija programa gde se inicijalizuju portovi i postavlja beskonačna petlja za kontinuiranu detekciju pritiska tastera. Konfiguracija pinova: TRISB7_bit = 1; - Postavlja pin RB7 kao ulaz (na koji je povezan taster). TRISD = 0x00; - Postavlja sve pinove porta D kao izlaze (gde su povezane LED diode). Inicijalizacija izlaznog registra i promenljivih: LATD = 0x00; - Svi pinovi na portu D su postavljeni na nisko, što znači da su sve LED diode isključene. oldstate = 0; - Postavlja početno stanje tastera na 0 (nepritisnuto). counter = 0; - Brojač je inicijalizovan na 0, što znači da će prva dioda na portu D biti uključena. Beskonačna petlja: do { ... } while(1); - Beskonačna petlja u kojoj program kontinuirano prati stanje tastera i upravlja LED diodama. Detekcija pritiska tastera: if(Button(&PORTB, 7, 1, 1)){ oldstate = 1; } - Proverava da li je taster na pinu RB7 pritisnut. Ako jeste, postavlja oldstate na 1, što signalizira da je taster bio pritisnut. Detekcija otpuštanja tastera i ažuriranje brojača: if(oldstate && Button(&PORTB, 7, 1, 0)){ ... } - Proverava da li je taster otpušten. Ako je taster bio pritisnut (oldstate == 1) i sada je otpušten, tada se izvršava sledeći deo koda: counter++; - Povećava brojač za 1, pomerajući aktivnu svetleću diodu na sledeći pin. oldstate = 0; - Resetuje oldstate na 0, pripremajući se za sledeći ciklus detekcije tastera. counter = counter%8; - Ograničava vrednost brojača na opseg od 0 do 7, što omogućava kružno pomeranje svetleće diode. Uključivanje odgovarajuće diode: LATD = 0x01 << counter; - Pomera binarnu vrednost 0x01 (binarno 00000001) ulevo za broj mesta definisan counter promenljivom. Ovo aktivira odgovarajući pin na portu D, uključujući svetleću diodu na toj poziciji. Ukratko ovaj kod omogućava inkrementalno pomeranje svetleće diode na portu D korišćenjem jednog tastera. Svaki put kada pritisnete i otpustite taster, svetleća dioda se pomera za jedno mesto udesno. Kada svetleća dioda dostigne poslednji pin (krajnju poziciju), ciklus se ponavlja od prvog pina.

12. LED CHASER – KNIGHT RIDER

Cilj ove vežbe je efekat plivajuće diode, diode na portu D treba da se sekvencijalno pale, diode RD7- RD0, nakon toga RD0- RD7. Na ploči je potrebno uključiti prekidač SW3.4.

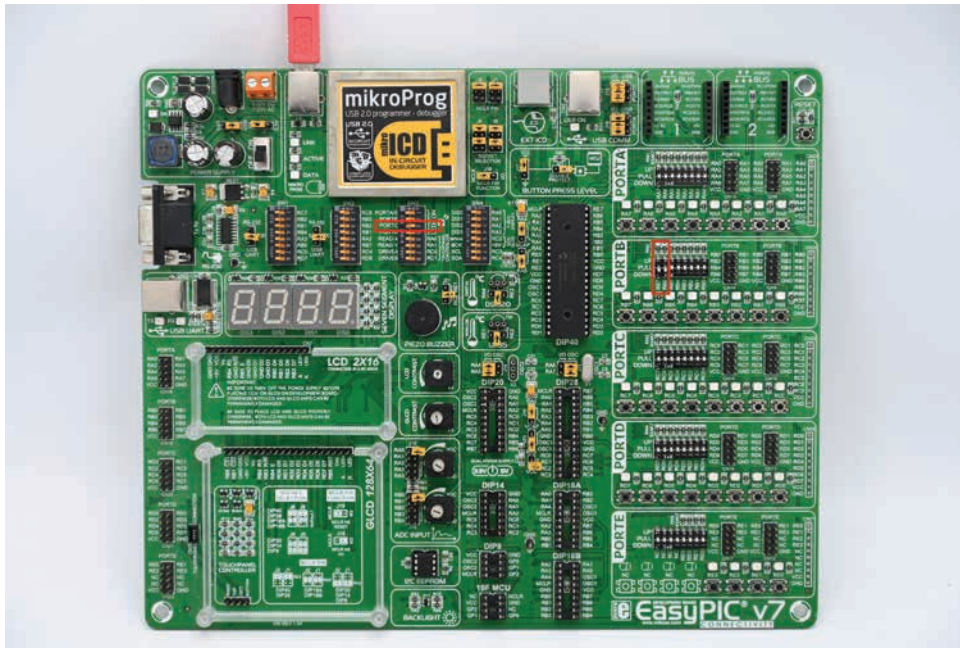



```
1.  char counter;
2.
3.  void main(){
4.      TRISD = 0x00;
5.      LATD = 0x00;
6.      counter = 0;
7.
8.      do {
9.          for(counter=0; counter<7; counter++){
10.             LATD = 0x80 >> counter;
11.             delay_ms(40);
12.         }
13.         for(counter=0; counter<7; counter++){
14.             LATD = 0x01 << counter;
15.             delay_ms(40);
16.         }
17.     } while(1);
18. }
```

Definisanje promenljive: char counter; - Ovo je promenljiva koja se koristi kao brojač za iteraciju kroz LED diode na portu D. Inicijalizacija portova: TRISD = 0x00; - Svi pinovi porta D su postavljeni kao izlazi. To znači da su svi pinovi spremni da kontrolišu LED diode. LATD = 0x00; - Svi pinovi na portu D su inicijalno postavljeni na nisko (svi LED-ovi su isključeni). Beskonačna petlja: do { ... } while(1); - Petlja se izvršava beskonačno, omogućavajući stalno ponavljanje efekta. Prvi for ciklus - LED dioda se pomera s leva na desno: for(counter=0; counter<7; counter++){ ... } - Ovaj ciklus prolazi kroz pinove na portu D od najvišeg (pin 7) ka najnižem (pin 0). LATD = 0x80 >> counter; - U ovom izrazu, binarna vrednost 0x80 (binarno 10000000) se pomera udesno za broj mesta definisan promenljivom counter. Ovo znači da će svetleti LED dioda koja je inicijalno na krajnjem levom mestu, a zatim se pomera ka desnom. delay_ms(40); - Pauza od 40 milisekundi između promena stanja LED dioda, kako bi se postigao vidljiv efekat “kretanja” dioda. Drugi for ciklus - LED dioda se pomera s desna na levo: for(counter=0; counter<7; counter++){ ... } - Ovaj ciklus prolazi kroz pinove na portu D od najnižeg (pin 0) ka najvišem (pin 7). LATD = 0x01 << counter; - U ovom izrazu, binarna vrednost 0x01 (binarno 00000001) se pomera ulevo za broj mesta definisan promenljivom counter. Ovo znači da će svetleti LED dioda koja je inicijalno na krajnjem desnom mestu, a zatim se pomera ka levom. delay_ms(40); - Pauza od 40 milisekundi između promena stanja LED dioda, kako bi se postigao vidljiv efekat “kretanja” dioda. Ukratko, ovaj program simulira efekat “Knight Rider” pomoću LED dioda priključenih na port D. LED diode se pale redom sa leva na desno, a zatim se vraćaju nazad sa desna na levo, stvarajući efekat kretanja dioda u oba smera.

13. KONTROLA DIODE PREKO PWM SIGNALA

Cilj ove vežbe je da se pritiskom na dugme RB7 pojačava intenzitet svetlosti dioda na portu C a pritiskom na dugme RB6 smanjuje. Na ploči je potrebno uključiti prekidač SW3.3. i postaviti prekidače 6 i 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button i PWM12 biblioteke.




```
1.  unsigned short current_duty, old_duty, current_duty1, old_duty1;
2.  bit oldstate7,oldstate6;
3.  void InitMain() {
4.       ANSELB = 0;
5.       TRISB7_bit = 1;
6.       TRISB6_bit = 1;
7.       TRISC = 0;
8.       LATC = 0x00;
9.       PWM1_Init(5000);
10. }
11. void main() {
12.      InitMain();
13.      current_duty = 16;
```

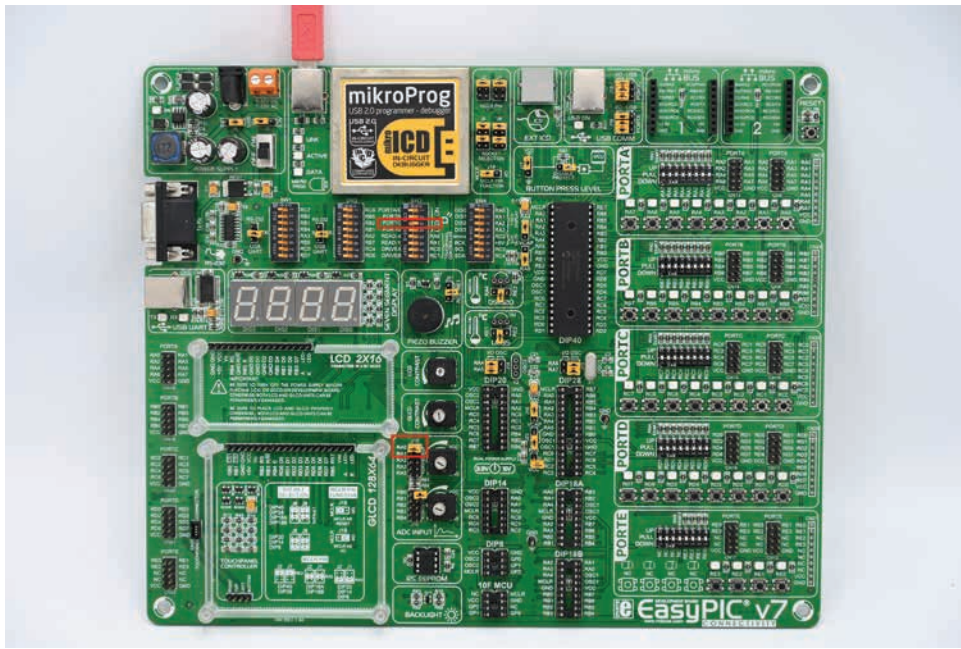
```
14.     PWM1_Start();
15.     PWM1_Set_Duty(current_duty);
16.     while (1) {
17.         if (Button(&PORTB, 7, 1, 1)) {
18.             oldstate7 = 1;
19.         }
20.         if (oldstate7 && Button(&PORTB, 7, 1, 0)) {
21.             current_duty+=10;
22.         if (current_duty > 255) current_duty = 0;
23.             PWM1_Set_Duty(current_duty);
24.             oldstate7 = 0;
25.         }
26.         if (Button(&PORTB, 6, 1, 1)) {
27.             oldstate6 = 1;
28.         }
29.         if (oldstate6 && Button(&PORTB, 6, 1, 0)) {
30.             current_duty-=10;
31.         if (current_duty <0) current_duty = 255;
32.             PWM1_Set_Duty(current_duty);
33.             oldstate6 = 0;
34.         }
35.         //Delay_ms(5);
36.     }
37. }
```

Deklaracija promenljivih: unsigned short current_duty, old_duty, current_duty1, old_duty1; - Promenljive za čuvanje trenutne i prethodne vrednosti širine impulsa (duty cycle) PWM signala. bit oldstate7, oldstate6; - promenljive za praćenje stanja tastera povezanih na portove B7 i B6. Funkcija InitMain: ANSELB = 0; - Konfiguriše port B kao digitalni ulaz/izlaz (isključuje analogne funkcije). TRISB7_bit = 1; i TRISB6_bit = 1; - Konfiguriše B7 i B6 pinove kao ulaze (za tastere). TRISC = 0; - Konfiguriše sve pinove porta C kao izlaze. LATC = 0x00; - Postavlja sve izlazne pinove porta C na nisku vrednost (isključuje sve diode). PWM1_Init(5000); - Inicijalizuje PWM modul sa frekvencijom od 5 kHz. Glavna funkcija main: InitMain(); - Poziva inicijalizaciju portova i PWM modula. current_duty = 16; - Postavlja početnu vrednost širine impulsa PWM signala. PWM1_Start(); - Pokreće PWM modul. PWM1_Set_Duty(current_duty); - Postavlja početnu vrednost širine impulsa na current_duty. Beskonačna petlja while(1): Program u petlji konstantno proverava stanje tastera i menja osvetljenje LED diode u zavisnosti od pritisnutih tastera. Kontrola osvetljenja: Povećanje osvetljenja: if (Button(&PORTB, 7, 1, 1)) { oldstate7 = 1; } - Proverava da li je taster na B7 pritisnut i postavlja promenljivu oldstate7 na 1. if (oldstate7 && Button(&PORTB, 7, 1, 0)) { ... } - Ako je taster prethodno bio pritisnut i sada je pušten, povećava vrednost current_duty za 10. if (current_duty > 255) current_duty = 0; - Ako current_duty pređe vrednost od 255, vraća se na 0 (osvetljenje se resetuje). PWM1_Set_Duty(current_duty); - Postavlja novu vrednost širine impulsa na PWM signalu.

oldstate7 = 0; - Resetuje stanje tastera B7. Smanjenje osvetljenja: if (Button(&PORTB, 6, 1, 1)) { oldstate6 = 1; } - Proverava da li je taster na B6 pritisnut i postavlja promenljivu oldstate6 na 1. if (oldstate6 && Button(&PORTB, 6, 1, 0)) { ... } - Ako je taster prethodno bio pritisnut i sada je pušten, smanjuje vrednost current_duty za 10. if (current_duty < 0) current_duty = 255; - Ako current_duty pređe vrednost od 0, vraća se na 255 (osvetljenje se resetuje). PWM1_Set_Duty(current_duty); - Postavlja novu vrednost širine impulsa na PWM signalu. oldstate6 = 0; - Resetuje stanje tastera B6. Ukratko, ovaj program omogućava kontrolu osvetljenja LED diode korišćenjem PWM signala. Korisnik može povećavati ili smanjivati osvetljenje LED diode pomoću dva tastera. Kada je jedan taster pritisnut, osvetljenje se postepeno povećava, a kada je drugi taster pritisnut, osvetljenje se smanjuje. PWM signal se koristi za preciznu kontrolu intenziteta svetlosti diode, što omogućava gladak prelaz između različitih nivoa osvetljenja.

14. ANALOGNO ČITANJE VREDNOSTI KORISTEĆI POTENCIOMETAR P1

Cilj ove vežbe je čitati analognu vrednost sa pina RA0 i upisivati tu vrednost na port C, tj bitsku vrednost prikazati na portu C. Na ploči je potrebno uključiti prekidač SW3.3. i postaviti jumper J15 na RA0 kako bi koristili potenciometar P1. Takođe je potrebno uključiti ADC biblioteku.

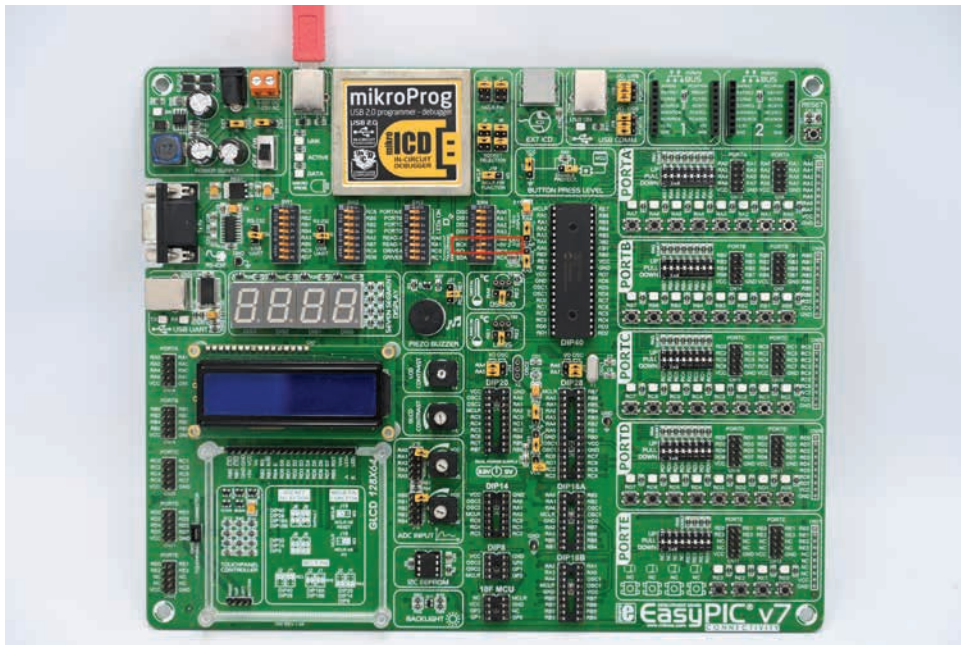



```
1.  #include <built_in.h>
2.  unsigned int adc_rd;
3.  void main() {
4.      ANSELA = 1;
5.      TRISA0_bit = 1;
6.      TRISC = 0x00;
7.      ADC_Init();
8.      while (1) {
9.          adc_rd = ADC_Get_Sample (0);
10.         LATC = adc_rd >> 2;
11.     }
12. }
```

Uključivanje biblioteka: #include <built_in.h> - Uključuje biblioteku koja omogućava korišćenje ugrađenih funkcija poput one za inicijalizaciju ADC modula. Deklaracija promenljive: unsigned int adc_rd; - Promenljiva adc_rd koristi se za skladištenje očitane analogne vrednosti sa A0 pina. Funkcija main: Inicijalizacija: ANSELA = 1; - Konfiguriše port A kao analogni (omogućava rad ADC-a na ovom portu). TRISA0_bit = 1; - Konfiguriše pin A0 kao ulaz (gde je povezan potenciometar). TRISC = 0x00; - Konfiguriše port C kao izlaz (za LED diode). ADC_Init(); - Inicijalizuje ADC modul mikrokontrolera koji omogućava analogno-digitalnu konverziju. Beskonačna petlja while(1): Čitanje analogne vrednosti: adc_rd = ADC_Get_Sample(0); - Očitava analognu vrednost sa pina A0 koristeći ADC i smešta tu vrednost u promenljivu adc_rd. ADC vrši konverziju analogne vrednosti (koja može biti u rasponu od 0 do 1023) u digitalnu vrednost. Postavljanje vrednosti na port C:LATC = adc_rd >> 2; - Pomera očitanu vrednost u desno za 2 bita (deljenje sa 4) i postavlja tu vrednost na port C. Pošto je port C 8-bitni, a ADC daje 10-bitni rezultat, deljenjem sa 4 dobija se vrednost u rasponu od 0 do 255 koja može biti direktno prikazana na portu C.

15. LCD ISPIS

Cilj ove vežbe je ispisati tekst Primer 1 a zatim Singidunum MikroElektronika na LCD displeju. Kako bi LCD funkcionisao, na ploči je potrebno uključiti prekidač SW4.6. za LCD. Takođe je potrebno uključiti Button i Lcd biblioteke.



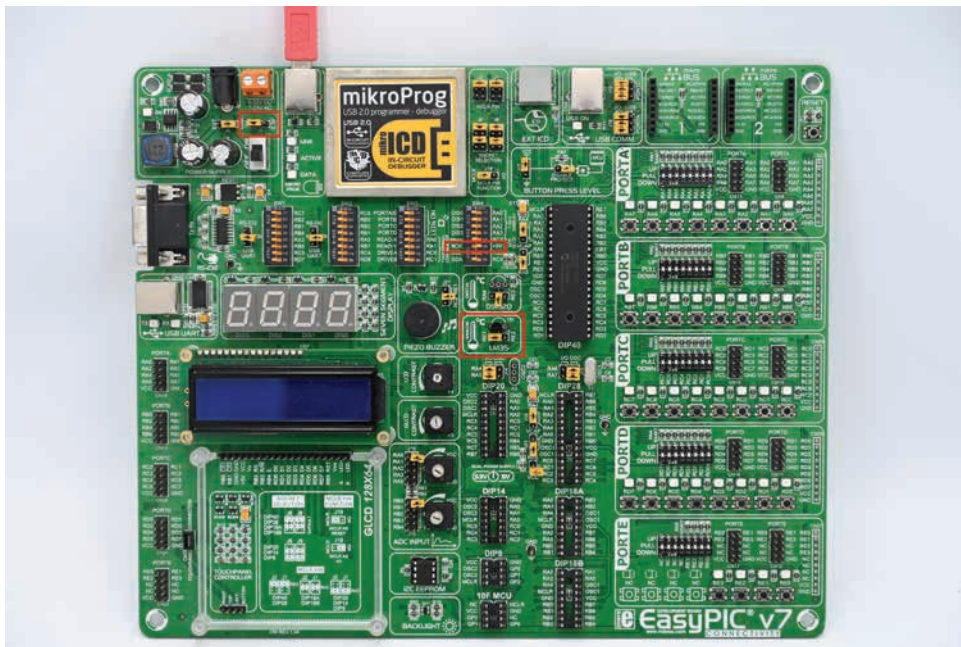

```
1.  sbit LCD_RS at LATB4_bit;
2.  sbit LCD_EN at LATB5_bit;
3.  sbit LCD_D4 at LATB0_bit;
4.  sbit LCD_D5 at LATB1_bit;
5.  sbit LCD_D6 at LATB2_bit;
6.  sbit LCD_D7 at LATB3_bit;
7.
8.  sbit LCD_RS_Direction at TRISB4_bit;
9.  sbit LCD_EN_Direction at TRISB5_bit;
10. sbit LCD_D4_Direction at TRISB0_bit;
11. sbit LCD_D5_Direction at TRISB1_bit;
12. sbit LCD_D6_Direction at TRISB2_bit;
13. sbit LCD_D7_Direction at TRISB3_bit;
14.
15. char txt1[] = "Singidunum";
16. char txt2[] = "MikroElektronika";
```

```
17. char txt3[] = “Primer”;
18. char txt4[] = “1”;
19.
20. void main() {
21.     ANSELB = 0;
22.     do{
23.         Lcd_Init();
24.         Lcd_Cmd(_LCD_CLEAR);
25.         Lcd_Cmd(_LCD_CURSOR_OFF);
26.         Lcd_Out(1,6,txt3);
27.         Lcd_Out(2,8,txt4);
28.         Delay_ms(2000);
29.         Lcd_Cmd(_LCD_CLEAR);
30.         Lcd_Out(1,1,txt1);
31.         Lcd_Out(2,1,txt2);
32.         Delay_ms(2000);
33.         Lcd_Cmd(_LCD_CLEAR);
34.     } while(1);
35. }
```

Definisanje LCD pinova: Redovi 1-13: Ovde se definišu pinovi na mikrokontroleru koji će biti povezani sa LCD displejem. To uključuje RS (Register Select), EN (Enable), i data linije D4-D7. Inicijalizacija promenljivih sa tekstom: Redovi 15-18: Definišu se tekstualni nizovi (txt1, txt2, txt3, txt4) koji će biti prikazani na LCD-u. char txt1[] = “Singidunum”; - Prvi tekst za prikaz. char txt2[] = “MikroElektronika”; - Drugi tekst za prikaz. char txt3[] = “Primer”; - Treći tekst za prikaz. char txt4[] = “1”; - Četvrti tekst za prikaz. Funkcija main: Inicijalizacija: ANSELB = 0; - Konfiguriše port B kao digitalni (za rad sa LCD-om). Beskonačna petlja do...while(1): Inicijalizacija LCD-a: Lcd_Init(); - Inicijalizuje LCD displej. Lcd_Cmd(_LCD_CLEAR); - Briše sadržaj LCD-a. Lcd_Cmd(_LCD_CURSOR_OFF); - Isključuje kursor na LCD-u. Prikaz prvog seta podataka: Lcd_Out(1,6,txt3); - Ispisuje tekst “Primer” u prvom redu, na šestoj poziciji. Lcd_Out(2,8,txt4); - Ispisuje tekst “1” u drugom redu, na osmoj poziciji. Delay_ms(2000); - Čeka 2000 milisekundi (2 sekunde) pre nego što pređe na sledeći korak. Briše ekran i prikazuje drugi set podataka: Lcd_Cmd(_LCD_CLEAR); - Ponovo briše sadržaj LCD-a. Lcd_Out(1,1,txt1); - Ispisuje tekst “Singidunum” u prvom redu, na prvoj poziciji. Lcd_Out(2,1,txt2); - Ispisuje tekst “MikroElektronika” u drugom redu, na prvoj poziciji. Delay_ms(2000); - Čeka još 2 sekunde pre nego što se proces ponovi. Beskonačna petlja: while(1); - Ova petlja osigurava da se proces neprekidno ponavlja, prikazujući naizmenično oba seta teksta na LCD displeju. Ukratko, ovaj program demonstrira osnovno korišćenje LCD displeja za prikazivanje tekstualnih poruka. Tekstovi se prikazuju naizmenično, sa pauzom između prikaza.

16. OČITAVANJE TEMPERATURE SA LM35 I ISPIS NA LCD

Cilj ove vežbe je merenje i prikazivanje trenutne temperaturu na LCD displeju koristeći LM35 senzor. Na ploči je potrebno postaviti LM35 senzor na odgovarajuće mesto i postaviti jumper J25 na RE1 kao i uključiti prekidač SW4.6. za LCD. Obratiti pažnju na jumper J5 odnosno na koju voltažu je podešen jer će biti potrebna za izračunavanje temperature. Takođe je potrebno uključiti ADC, Conversions, C_String i Lcd biblioteke.




```
1.  sbit LCD_RS at LATB4_bit;
2.  sbit LCD_EN at LATB5_bit;
3.  sbit LCD_D4 at LATB0_bit;
4.  sbit LCD_D5 at LATB1_bit;
5.  sbit LCD_D6 at LATB2_bit;
6.  sbit LCD_D7 at LATB3_bit;
7.
8.  sbit LCD_RS_Direction at TRISB4_bit;
9.  sbit LCD_EN_Direction at TRISB5_bit;
10. sbit LCD_D4_Direction at TRISB0_bit;
11. sbit LCD_D5_Direction at TRISB1_bit;
```

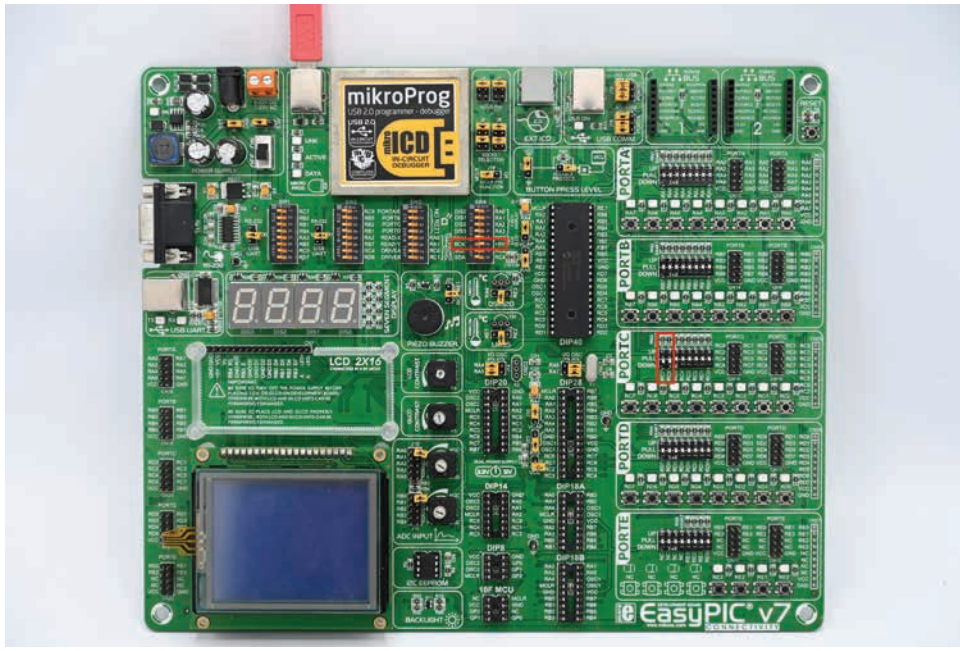
```
12. sbit LCD_D6_Direction at TRISB2_bit;
13. sbit LCD_D7_Direction at TRISB3_bit;
14.
15. const unsigned short VREF = 5;
16.
17. unsigned int temp_res = 0;
18. float temp;
19. char txt[15];
20.
21. void main() {
22.   ANSELB = 0;
23.   ANSELE = 0x02;
24.   TRISE1_bit = 1;
25.
26.   ADC_Init();
27.
28.   Lcd_Init();
29.   Lcd_Cmd(_LCD_CLEAR);
30.   Lcd_Cmd(_LCD_CURSOR_OFF);
31.
32.   Lcd_Out(1, 1, "Temperature :");
33.   Lcd_Chr(2,8,223);
34.   Lcd_Chr(2,9,'C');
35.
36.   temp_res = 0;
37.
38.   do {
39.     temp_res = ADC_Get_Sample(6);
40.     temp = (temp_res * VREF)/10.240;
41.
42.     FloatToStr(temp, txt);
43.     txt[4] = 0;
44.     Lcd_Out(2,3,txt);
45.     Delay_ms(300);
46.   } while(1);
47. }
```

Definisanje LCD pinova: Redovi 1-13: Definišu se pinovi mikrokontrolera koji su povezani sa LCD displejem. Inicijalizacija promenljivih: Red 15: const unsigned short VREF = 5; - Definiše referentni napon ADC-a kao 5V. Redovi 17-19: Definišu se promenljive: temp_res - Služi za čuvanje rezultata očitavanja ADC-a. temp - Čuva izračunatu vrednost temperature. txt - Niz karaktera za prikazivanje temperature na LCD-u. Inicijalizacija u funkciji main: Redovi 22-24: ANSELB = 0; - Port B je konfigurisan kao digitalni. ANSELE = 0x02; - Pin RE1 se koristi kao analogni ulaz za očitavanje temperature. TRISE1_bit = 1; - Pin RE1 se postavlja kao ulazni. Red 26: ADC_Init(); - Inicijalizuje se ADC modul.

Redovi 28-30: Inicijalizuje se LCD i briše se početni ekran. Red 32: Lcd_Out(1, 1, “Temperature :”); - Prikazuje tekst “Temperature :” u prvom redu LCD-a. Redovi 33-34: Prikazuje znak za stepen i oznaku “C” za Celzijuse u drugom redu na LCD-u. Petlja za očitavanje i prikaz temperature: Red 39: temp_res = ADC_Get_Sample(6); - Očitava se vrednost sa ADC kanala 6 (gde je povezan LM35). Red 40: temp = (temp_res * VREF)/10.240; - Preračunava očitanu vrednost iz ADC-a u temperaturu u stepenima Celzijusa. LM35 daje 10 mV po stepenu Celzijusa, pa se ADC rezultat množi sa referentnim naponom (5V) i deli sa faktorom 10.240 (što je ekvivalentno deljenju sa 1024 i pomnoženo sa 100 da se dobije temperatura u stepenima). Red 42: FloatToStr(temp, txt); - Pretvara vrednost temperature u tekstualni niz. Red 43: txt[4] = 0; - Skracuje tekst na 4 karaktera (za prikaz temperature sa jednom decimalom). Red 44: Lcd_Out(2,3,txt); - Prikazuje temperaturu u drugom redu LCD-a. Red 45: Delay_ms(300); - Zadržava prikaz za 300 milisekundi pre sledećeg očitavanja. Beskonačna petlja: Red 46: Petlja se ponavlja neprestano, osvežavajući vrednost temperature na LCD-u. Ukratko, ovaj program omogućava korisniku da u realnom vremenu prati temperaturu očitanu sa senzora LM35 i prikaže je na LCD displeju. Program demonstrira korišćenje ADC modula mikrokontrolera i prikaz podataka na LCD-u, što je korisno za aplikacije koje zahtevaju praćenje temperature i prikazivanje rezultata na jednostavan način.

17. OSNOVA SNAKE IGRICE

Cilj ove vežbe je kreiranje osnovnog kretanja zmije na GLCD ekranu. Zmiju treba kontrolisati dugmićima RC7 i RC6, dok portovi B i D služe za upravljanje GLCD ekranom. Potrebno je takođe implementirati i mogućnost da zmija prolazi kroz zidove. Zmija je uvek konstantne dužine 20 piksela i ne raste niti jede kockice koje se pojavljuju kao što je to običaj u klasičnoj snake igrici. Program treba samo da implementira kretanje zmije po ekranu uz pomoć tastera. Kako bi GLCD funkcionisao, na ploči je potrebno uključiti prekidač SW4.6. za GLCD kao i postaviti prekidače 6 i 7 na portu C u pull down položaj. Egzaktan timer konfigurisati uz pomoć Timer calculator alata. Takođe je potrebno uključiti Button i Glcd biblioteke.




```
1.  #define UP 0
2.  #define RIGHT 1
3.  #define DOWN 2
4.  #define LEFT 3
5.
6.  #define ARENA_X 128
7.  #define ARENA_Y 64
8.
9.  char GLCD_DataPort at PORTD;
```

```
sbit GLCD_CS1 at LATB0_bit;
sbit GLCD_CS2 at LATB1_bit;
sbit GLCD_RS at LATB2_bit;
sbit GLCD_RW at LATB3_bit;
sbit GLCD_EN at LATB4_bit;
sbit GLCD_RST at LATB5_bit;
sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;

int snakeDir = RIGHT;
int arena[ARENA_X][ARENA_Y];
int snakeX[20];
int snakeY[20];
bit oldstate7, oldstate6,flag7,flag6;
int flag,left,right;
int direction;
int i,k;
void interrupt() {
 if (TMR0IF_bit) {
    TMR0IF_bit = 0;
    TMR0H = 0x0B;
    TMR0L = 0xDC;
    flag=1;
 }
}
void arena_clear(){
    Glcd_Fill(0x00);
    for ( i = 19; i > 0; i-- ){
        snakeX[i]=0;
        snakeY[i]=0;
    }
    snakeX[0]=0;
    snakeY[0]=0;
    direction=RIGHT;
}
 void main() {
    ANSELB = 0;
    ANSELC = 0;
    ANSELD = 0;
    TRISC=0xFF;
    T0CON = 0x84;
    TMR0H = 0x0B;
```

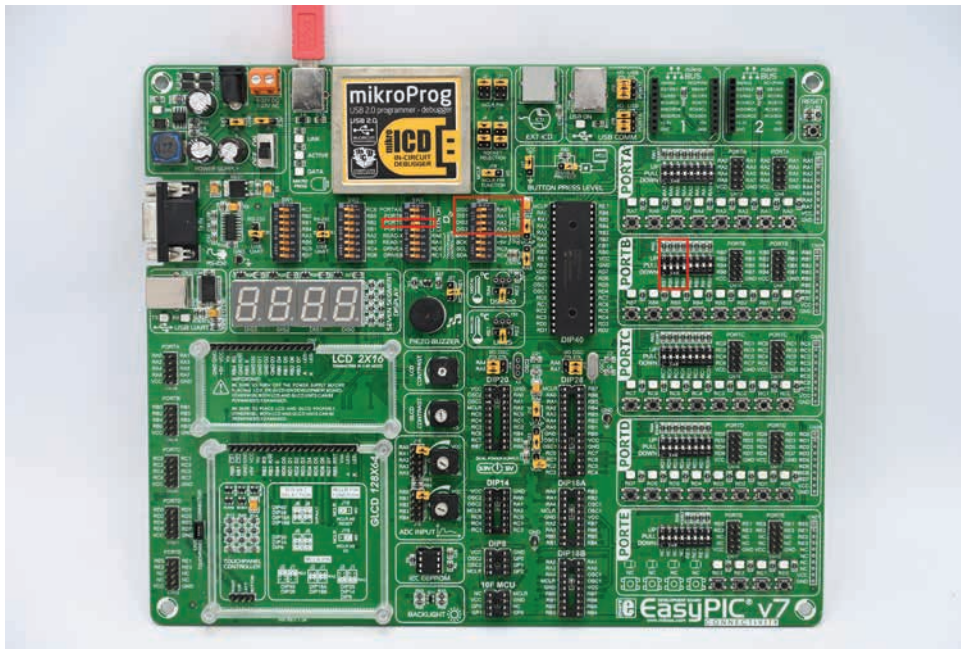
```
56.     TMR0L = 0xDC;
57.     TMR0IE_bit = 1;
58.     INTEDG0_bit = 1;
59.     GIE_bit = 1;
60.     Glcd_Init();
61.     arena_clear();
62.     flag7=0;
63.     flag6=0;
64.     do {
65.         if (Button(&PORTC, 7, 1, 1)) {
66.             oldstate7 = 1;
67.         }
68.         if (oldstate7 && Button(&PORTC, 7, 1, 0)) {
69.             flag7 =1;
70.             oldstate7 = 0;
71.         }
72.         if (Button(&PORTC, 6, 1, 1)) {
73.             oldstate6 = 1;
74.         }
75.         if (oldstate6 && Button(&PORTC, 6, 1, 0)) {
76.             flag6 =1;
77.             oldstate6 = 0;
78.         }
79.         if (flag){
80.             Glcd_Dot(snakeX[19], snakeY[19], 0);
81.             for ( i = 19; i > 0; i-- ){
82.               if(snakeX[0]==snakeX[i] && snakeY[0]==snakeY[i]){
83.                 arena_clear();
84.                 }
85.                 snakeX[i]=snakeX[i-1];
86.                 snakeY[i]=snakeY[i-1];
87.             }
88.             switch (direction){
89.                 case UP:
90.                     snakeY[0]--;
91.                     if (snakeY[0]<0) snakeY[0]=63;
92.                     if (flag7){
93.                         direction = LEFT;
94.                         flag7=0;
95.                     }
96.                     if (flag6){
97.                         direction = RIGHT;
98.                         flag6 = 0;
99.                     }
100.                   break;
101.                  case DOWN:
```

```
102.                         snakeY[0]++;
103.                         if (snakeY[0]>63) snakeY[0]=0;
104.                         if (flag7){
105.                             direction = RIGHT;
106.                             flag7=0;
107.                         }
108.                         if (flag6){
109.                             direction = LEFT;
110.                             flag6 = 0;
111.                         }
112.                     break;
113.                     case RIGHT:
114.                         snakeX[0]++;
115.                         if (snakeX[0]>127) snakeX[0]=0;
116.                         if (flag7){
117.                             direction = UP;
118.                             flag7=0;
119.                         }
120.                         if (flag6){
121.                             direction = DOWN;
122.                             flag6 = 0;
123.                         }
124.                     break;
125.                     case LEFT:
126.                         snakeX[0]--;
127.                         if (snakeX[0]<0) snakeX[0]=127;
128.                         if (flag7){
129.                             direction = DOWN;
130.                             flag7=0;
131.                         }
132.                         if (flag6){
133.                             direction = UP;
134.                             flag6 = 0;
135.                         }
136.                     break;
137.                 }
138.                 flag=0;
139.                 for ( k = 0; k < 20; k++ ){
140.                     Glcd_Dot(snakeX[k], snakeY[k], 1);
141.                 }
142.             }
143.         } while(1);
144.     }
```

Definisanje konstanti: Redovi 1-4: Definišu se konstante za pravce kretanja zmije: gore (UP), desno (RIGHT), dole (DOWN), i levo (LEFT). Redovi 6-7: Definišu se dimenzije arene (ekrana) sa širinom od 128 piksela (ARENA_X) i visinom od 64 piksela (ARENA_Y). Povezivanje pinova za GLCD: Redovi 9-21: Ovi redovi povezuju portove i signale mikrokontrolera sa grafičkim LCD displejom (GLCD). Deklaracija promenljivih: Red 23: snakeDir je promenljiva koja čuva trenutni pravac kretanja zmije (inicijalno postavljena na RIGHT). Red 24: arena je dvodimenzionalni niz koji predstavlja arenu po kojoj se zmija kreće. Redovi 25-26: snakeX i snakeY su nizovi koji čuvaju koordinate svakog segmenta zmije po osi X i Y. Redovi 27-29: oldstate7, oldstate6, flag7, flag6, flag, left, right, i direction su promenljive koje se koriste za praćenje stanja tastera i pravca kretanja. Redovi 30-31: i i k su brojači koji se koriste u petljama. Interrupt rutine: Redovi 31-37: interrupt() funkcija se koristi za rukovanje tajmerom (TMR0). Kada tajmer istekne (TMR0IF_bit je postavljen), postavlja se flag na 1, signalizirajući da je vreme za ažuriranje zmije. Funkcija za čišćenje arene: Redovi 39-48: arena_clear() funkcija briše ekran (Glcd_Fill(0x00)) i resetuje pozicije zmije. Takođe, postavlja početni pravac zmije na RIGHT. Glavna funkcija main(): Redovi 49-63: Ovaj deo koda inicijalizuje portove i tajmer, postavlja interrupt rutine, inicijalizuje GLCD i postavlja početne vrednosti za igru. Redovi 64-78: Ovaj deo koda obrađuje pritiske tastera: Ako je pritisnut taster povezan na PORTC7 (Button(&PORTC, 7, 1, 1)), flag7 se postavlja na 1, što će kasnije promeniti pravac kretanja zmije. Slično, ako je pritisnut taster povezan na PORTC6 (Button(&PORTC, 6, 1, 1)), flag6 se postavlja na 1. Logika igre: Redovi 79-142: Ovaj deo koda upravlja kretanjem zmije i ažuriranjem ekrana: Redovi 80-87: Prvi korak u ažuriranju zmije je brisanje zadnjeg segmenta zmije sa ekrana (Glcd_Dot(snakeX[19], snakeY[19], 0)) i pomeranje svih segmenata zmije napred (snakeX[i]=snakeX[i-1]). Redovi 88-136: Zmija menja pravac u skladu sa trenutnim pravcem (UP, DOWN, RIGHT, LEFT). Ako je pritisnut odgovarajući taster (flag7 ili flag6), pravac se menja. Ako zmija pređe granice ekrana, pojavljuje se sa suprotne strane (npr. ako snakeY[0]<0, vraća se na donji deo ekrana). Red 138: flag=0 se postavlja na 0 nakon svakog pomeranja, čekajući da interrupt signalizuje sledeće ažuriranje. Redovi 139-141: Nakon što je zmija pomerena, svi segmenti zmije se ponovo crtaju na ekranu (Glcd_Dot(snakeX[k], snakeY[k], 1)). Beskonačna petlja: Red 143: do-while petlja koja se beskonačno ponavlja, omogućavajući kontinuirano kretanje zmije i reakcije na pritiske tastera. Ukratko, ovaj kod implementira osnovu za igricu “Snake” na grafičkom LCD displeju, gde se zmija kreće po ekranu i menja pravac kretanja na osnovu pritiska tastera. Kada zmija udari u sebe, igra se resetuje.

18. SIMON SAYS

Cilj ove vežbe je kreiranje „Simon says“ igrice, gde je potrebno tasterima na portu B (RB7-RB4) ponoviti sekvencu prikazanu paljenjem dioda na portu C (RC7-RC4). Svakim uspešnim unosom sekvence potrebno je dodati još jedan član na kraj sekvence i tako sve dok korisnik ne unese pogrešnu sekvencu. Takođe potrebno je meriti vreme potrebno za reprodukciju sekvence i ispisati ga na 4 sedmosegmentna displeja u formatu [sec sec. msec msec]. U slučaju da je korisnik uneo pogrešnu sekvencu treba ispisati „LOSE“ na sedmosegmentnom displeju. Kako bi 7-segmentni displej funkcionisao, na ploči je potrebno uključiti prekidače SW4.1., SW4.2., SW4.3., SW4.4. i SW3.3. kao i postaviti prekidače 4,5,6 i 7 na portu B u pull down položaj. Takođe je potrebno uključiti C_Stdlib biblioteku.




```
1.  #include “Display_Utils.h”
2.  #define FIRST 1
3.  #define SECOND 2
4.  #define THIRD 3
5.  #define FOURTH 4
6.  #define BLANK 5
7.  #define DISPLAY 1
8.  #define ANSWER 2
9.  #define END 3
10. #define IDLE 4
```

```
unsigned short shifter, portd_index;
unsigned int digit, sec, msec;
unsigned short portd_array[4];
int seq[30];
int level,state;
int pressed;
bit flag7,flag6,flag5,flag4;
int cnt,i,position;
int times;
void interrupt() {
    if(TMR0IF_bit){
        msec++;
        LATA = 0;
        LATD = portd_array[portd_index];
        LATA = shifter;

        shifter <<= 1;
        if (shifter > 8u)
        shifter = 1;

        portd_index++ ;
        if (portd_index > 3u)
        portd_index = 0;
        TMR0IF_bit = 0;
        TMR0L = 0x06;
    }
    if(RBIF_bit == 1 ) {
        RBIF_bit = 0;
        if (PORTB.F7 == 0 && flag7){
            flag7=0;
            pressed = FIRST;
        } else if(PORTB.F7 == 1) flag7=1;
        if (PORTB.F6 == 0 && flag6){
            pressed = SECOND;
            flag6=0;
        } else if(PORTB.F6 == 1) flag6=1;
        if (PORTB.F5 == 0 && flag5){
            pressed = THIRD;
            flag5=0;
        } else if(PORTB.F5 == 1) flag5=1;
        if (PORTB.F4 == 0 && flag4){
            pressed = FOURTH;
            flag4=0;
        } else if(PORTB.F4 == 1) flag4=1;
    }
}
```

```
57. void display(int cur){
58.     switch (cur){
59.         case FIRST :
60.             LATC = 0x80;
61.             delay_ms(400);
62.             LATC = 0x00;
63.             delay_ms(200);
64.         break;
65.         case SECOND :
66.             LATC = 0x40;
67.             delay_ms(400);
68.             LATC = 0x00;
69.             delay_ms(200);
70.         break;
71.         case THIRD :
72.             LATC = 0x20;
73.             delay_ms(400);
74.             LATC = 0x00;
75.             delay_ms(200);
76.         break;
77.         case FOURTH :
78.             LATC = 0x10;
79.             delay_ms(400);
80.             LATC = 0x00;
81.             delay_ms(200);
82.         break;
83.         case BLANK :
84.             LATC = 0x00;
85.         break;
86.     }
87. }
88. int generate(){
89.     int k;
90.     k = rand() & 0x03;
91.     return k;
92. }
93. void start_time(){
94.     digit = (sec / 10u) % 10u;
95.     portd_array[3] = mask(digit);
96.     digit = sec % 10u;
97.     portd_array[2] = mask(digit);
98.     portd_array[2] |= 0x80;
99.     digit = (msec / 100u) % 10u;
100.          portd_array[1] = mask(digit);
101.          digit = (msec / 10u) % 10u;
102.          portd_array[0] = mask(digit);
```

```
103.          if (msec >= 1000){
104.              sec++ ;
105.          if (sec == 60u)sec = 0;
106.              msec=0;
107.          }
108.      }
109.      void lose(){
110.          portd_array[3] = 0x38;
111.          portd_array[2] = 0x3F;
112.          portd_array[1] = 0x6D;
113.          portd_array[0] = 0x79;
114.      }
115.      void main() {
116.          TRISB7_bit = 1;
117.          ANSELA = 0;
118.          ANSELC = 0;
119.          ANSELD = 0;
120.          TRISC = 0;
121.          LATC = 0;
122.          TRISA = 0;
123.          LATA = 0;
124.          TRISD = 0;
125.          LATD = 0;
126.          TRISB = 0xFF;
127.          RBIE_bit = 1;
128.          IOCB4_bit = 1;
129.          IOCB5_bit = 1;
130.          IOCB6_bit = 1;
131.          IOCB7_bit = 1;
132.          RBIF_bit = 0;
133.          T0CON = 0xC4;
134.          TMR0L = 0x06;
135.          GIE_bit = 1;
136.          TMR0IE_bit = 1;
137.          level = 1;
138.          digit = 0;
139.          portd_index = 0;
140.          shifter = 1;
141.          times = 0;
142.          msec = 00;
143.          sec = 00;
144.          state = DISPLAY;
145.          portd_array[3] = mask(digit);
146.          portd_array[2] = mask(digit);
147.          portd_array[1] = mask(digit);
148.          portd_array[0] = mask(digit);
```

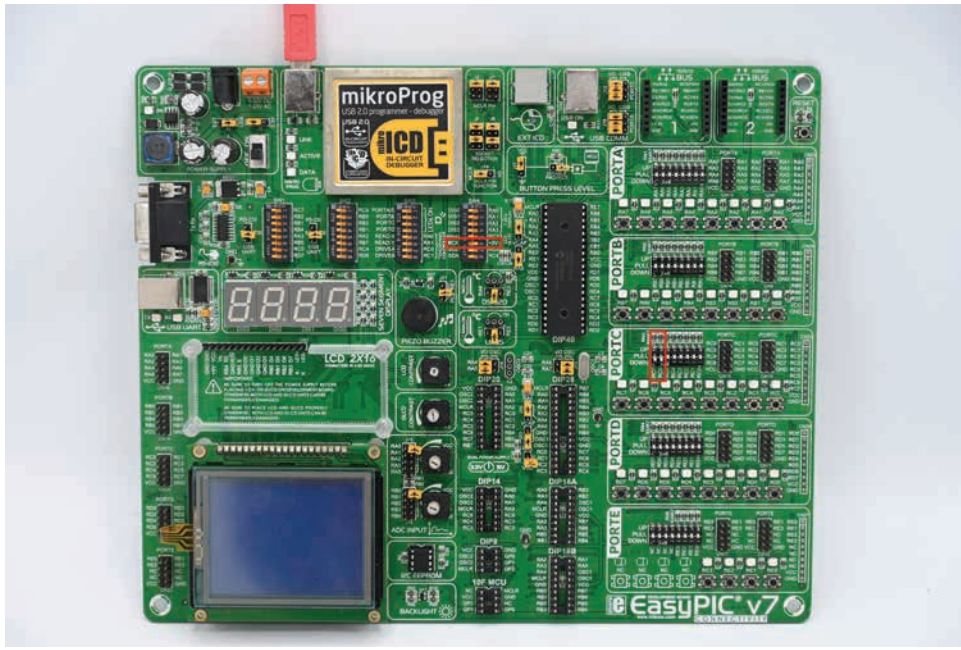
```
149.            //delay_ms(1000);
150.            pressed = BLANK;
151.            position=0;
152.            seq[0] = (int)generate()+1;
153.            do {
154.                if (times) start_time();
155.                switch (state) {
156.                    case DISPLAY:
157.                        times = 0;
158.                        for (i= 0;i<level;i++){
159.                            display(seq[i]);
160.                        }
161.                        state = ANSWER;
162.                        position = 0;
163.                        times=1;
164.                    break;
165.                    case ANSWER:
166.                        if (seq[position] == pressed){
167.                            position++;
168.                            pressed = BLANK;
169.                        } else if (pressed<5){
170.                            state = END;
171.                        }
172.                        if (position == level){
173.                            level++;
174.                            seq[level-1]=(int)generate()+1;
175.                            position = 0;
176.                            sec=0;
177.                            msec =0;
178.                            cnt= 0;
179.                            state = DISPLAY;
180.                        }
181.                    break;
182.                    case END:
183.                        times = 0;
184.                        lose();
185.                    break;
186.                }
187.            } while(1);
188.        }
```

Definisanje konstanti: Redovi 2-10: Definišu se konstante za LED diode i stanja igre. Na primer, FIRST, SECOND, THIRD, i FOURTH predstavljaju različite LED diode, dok BLANK označava da nijedna dioda nije pritisnuta. Takođe, postoje konstante koje predstavljaju različite faze igre, poput DISPLAY, ANSWER, END, i IDLE. Deklaracija promenljivih: Redovi 11-19: Ovde su deklarisane promenljive koje se koriste u igri. seq[] je niz koji

čuva sekvencu svetlećih dioda koju igrač treba da ponovi. level označava trenutni nivo igre, dok state označava trenutnu fazu igre. pressed beleži koju je diodu igrač pritisnuo, a times kontroliše merenje vremena. Interrupt rutine: Redovi 20-36: Funkcija interrupt() obrađuje dva glavna događaja: Timer 0 interrupt: Ova rutina upravlja prikazivanjem LED dioda na 7-segmentnom displeju i meri proteklo vreme u milisekundama (msec) i sekundama (sec). Interrupt sa PORTB: Ova rutina detektuje pritisak tastera povezanih na PORTB, i na osnovu toga postavlja vrednost pressed, koja označava koju je diodu igrač pritisnuo. Funkcija za prikazivanje LED dioda: Redovi 57-86: Funkcija display(int cur) koristi se za prikazivanje svetlećih dioda na osnovu vrednosti cur. Svaka LED dioda (npr. FIRST, SECOND, itd.) osvetljava se na određeno vreme (delay_ms(400)), a zatim se gasi (delay_ms(200)). Generisanje sekvence: Redovi 88-91: Funkcija generate() koristi se za generisanje nasumične vrednosti između 0 i 3, koja se zatim koristi za odabir sledeće diode u sekvenci. Prikaz vremena: Redovi 93-107: Funkcija start_time() koristi se za prikazivanje vremena na 7-segmentnom displeju. Ova funkcija osvežava vrednosti sekundi i milisekundi, kao i prikazivanje tih vrednosti na displeju. Prikaz poruke o gubitku: Redovi 109-113: Funkcija lose() koristi se za prikazivanje poruke “LOSE” na 7-segmentnom displeju kada igrač napravi grešku. Glavna funkcija main(): Redovi 115-187: Ovaj deo koda inicijalizuje portove, postavlja početne vrednosti za igru i upravlja glavnom logikom igre: Redovi 137-144: Inicijalizacija igre: postavljanje početnog nivoa, postavljanje vremena na nulu, i generisanje prve sekvence. Redovi 153-185: Glavna petlja igre upravlja različitim fazama igre: DISPLAY: Prikazuje sekvencu LED dioda koju igrač treba da ponovi. ANSWER: Proverava da li je igrač pravilno pritisnuo taster koji odgovara trenutnoj diodi u sekvenci. Ako jeste, igra nastavlja; ako nije, prelazi u stanje END. END: Prikazuje poruku o gubitku i završava igru. Ukratko, ovaj kod implementira igru “Simon Says” gde mikrokontroler nasumično osvetljava LED diode koje igrač treba da ponovi pritiskom na odgovarajuće tastere. Ako igrač napravi grešku, igra se završava i prikazuje se poruka o gubitku.

19. IGRA HVATANJA

Cilj ove vežbe je kreiranje igre hvatanja koristeći GLCD. Potrebno je grafički implementirati korpu za hvatanje na dnu GLCD ekrana koju korisnik pomera koristeći tastere RC7 i RC6 za pomeranje ulevo i udesno. Sa vrha GLCD ekrana treba da padaju elementi koje korisnik hvata sa korpom, svakim uspešnim hvatanjem, uvećava se rezultat i ubrzava se padanje elemenata. Na kraju igre, kada korisnik promaši element, potrebno je ispisati konačan rezultat. Kako bi GLCD funkcionisao, na ploči je potrebno uključiti prekidač SW4.6. za GLCD kao i i postaviti prekidače 6 i 7 na portu C u pull down položaj. Takođe je potrebno uključiti Button, C_Stdlib i Glcd biblioteke.




```
1.  #define GENERATE 0
2.  #define DROP 1
3.  #define EAT 2
4.  #define MISSED 3
5.  #define IDLE 4
6.
7.  #define ARENA_X 128
8.  #define ARENA_Y 64
9.
10. char GLCD_DataPort at PORTD;
11. sbit GLCD_CS1 at LATB0_bit;
```

```
sbit GLCD_CS2 at LATB1_bit;
sbit GLCD_RS at LATB2_bit;
sbit GLCD_RW at LATB3_bit;
sbit GLCD_EN at LATB4_bit;
sbit GLCD_RST at LATB5_bit;
sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;

int arena[ARENA_X][ARENA_Y];
const int BASKETX[20] ={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
    15,16,17,18,19};
const int BASKETY[20]
={58,59,60,61,62,63,63,63,63,63,63,63,63,63,63,62,61,60,59,58};
const int FOODX[6] = {0,1,2,3,4,5};
const int FOODY[6] = {0,1,2,3,4,5};
bit oldstate7, oldstate6,flag7,flag6;
int flag,left,right;
int move_offset, food_posX, food_posY, cnt, score,state;
int i,k,j,flag_eat;
void interrupt() {
    if (TMR0IF_bit) {
        TMR0IF_bit = 0;
        TMR0L = 0x06;
        cnt++;
        flag;
    }
}
void draw(int color){
    for ( k = 0; k < 20; k++ ){
       Glcd_Dot(BASKETX[k]+move_offset, BASKETY[k], color);
    }
}
void draw_food(int color){
    for ( k = 0; k < 6; k++ ){
        for (j=0; j<6; j++){
          Glcd_Dot(FOODX[k]+food_posX,FOODY[j]+food_posY, color);
        }
    }
}
void arena_clear(){
    Glcd_Fill(0x00);
    move_offset=0;
}
int generate(){
```

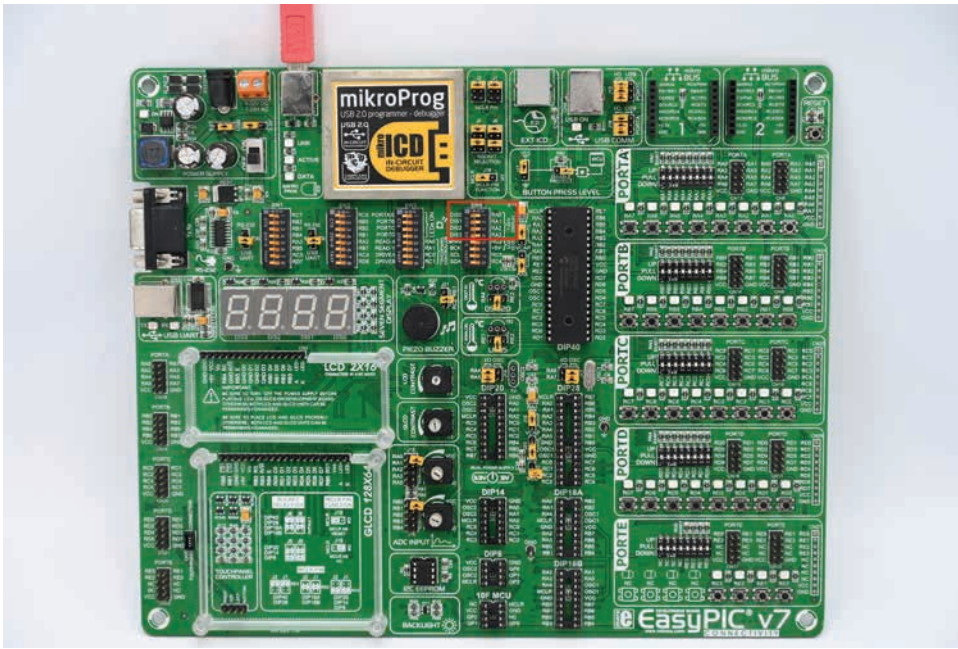
```
59.     int k;
60.     k = rand() & 0x39;
61.     return k*2;
62. }
63. void game_over(){
64.     unsigned int short ONES;
65.     unsigned int short TENS;
66.     draw(0);
67.     Glcd_Write_Text("KRAJ", 50, 2, 1);
68.     Glcd_Write_Text("Rezultat:", 40, 4, 1);
69.     TENS = (score/10)%10 + 48;
70.     ONES = score%10 + 48;
71.     Glcd_Write_char(TENS, 95, 4, 1);
72.     Glcd_Write_char(ONES, 101, 4, 1);
73. }
74. void main() {
75.     ANSELB = 0;
76.     ANSELC = 0;
77.     ANSELD = 0;
78.     TRISC=0xFF;
79.     T0CON = 0xC4;
80.     TMR0L = 0x06;
81.     TMR0IE_bit = 1;
82.     INTEDG0_bit = 1;
83.     GIE_bit = 1;
84.     Glcd_Init();
85.     arena_clear();
86.     flag7=0;
87.     flag6=0;
88.     state = GENERATE;
89.     score=0;
90.     flag_eat=0;
91.     do {
92.         draw(1);
93.         if (Button(&PORTC, 7, 1, 1)) {
94.             oldstate7 = 1;
95.         }
96.         if (oldstate7 && Button(&PORTC, 7, 1, 0)) {
97.             draw(0);
98.             move_offset-=11;
99.         if (move_offset < 0) move_offset=0;
100.                 oldstate7 = 0;
101.             }
102.             if (Button(&PORTC, 6, 1, 1)) {
103.                 oldstate6 = 1;
104.             }
105.             if (oldstate6 && Button(&PORTC, 6, 1, 0)) {
106.                 draw(0);
```

```
107.                move_offset+=11;
108.            if (move_offset > 107) move_offset=107;
109.                oldstate6 = 0;
110.            }
111.            switch (state){
112.                case GENERATE:
113.                    food_PosX = generate();
114.                    food_PosY=0;
115.                    draw_food(1);
116.                    state = DROP;
117.                break;
118.                case DROP:
119.                    if ((cnt+score)>=400){
120.                     draw_food(0);
121.                     food_PosY+=5;
122.                     cnt=0;
123.                     draw_food(1);
124.                    }
125.                    if (food_PosY>=52){
126.                     draw_food(0);
127.                     for (i=0;i<20;i++){
128.                    if((BASKETX[i]+move_offset)==food_
                          PosX){
129.                            flag_eat=1;
130.                        }
131.                     }
132.                     if(flag_eat){
133.                         state=EAT;
134.                         flag_eat=0;
135.                     }
136.                     else{
137.                         state=MISSED;
138.                     }
139.                    }
140.                break;
141.                case EAT:
142.                    score++;
143.                    state=GENERATE;
144.                break;
145.                case MISSED:
146.                    game_over();
147.                    state=IDLE;
148.                break;
149.            }
150.            flag=0;
151.        } while(state!=IDLE);
152.    }
```

Definisanje konstanti: Redovi 1-9: Definišu se konstante koje predstavljaju različite faze igre (GENERATE, DROP, EAT, MISSED, IDLE) i dimenzije arene (ARENA_X, ARENA_Y). Inicijalizacija GLCD portova: Redovi 10-22: Ovi redovi postavljaju portove mikrokontrolera za komunikaciju sa grafičkim LCD ekranom. Deklaracija promenljivih: Redovi 24-33: Deklarišu se globalne promenljive koje se koriste u igri: arena[][]: Dvodimenzionalni niz koji predstavlja arenu (ekran). BASKETX[] i BASKETY[]: Nizovi koji definišu poziciju korpe na ekranu. FOODX[] i FOODY[]: Nizovi koji definišu poziciju hrane koja pada. flag, left, right, move_offset: Promenljive koje kontrolišu pomeranje korpe. food_posX, food_posY: Promenljive koje definišu trenutnu poziciju hrane na ekranu. cnt, score, state: Promenljive za brojanje, rezultat i praćenje trenutnog stanja igre. flag_eat: Promenljiva koja označava da je hrana uhvaćena. Interrupt rutina: Redovi 34-40: Ova rutina se izvršava kada tajmer (Timer0) generiše prekid. Ona jednostavno inkrementira brojač cnt, koji se koristi za kontrolu brzine padanja hrane. Funkcije za crtanje: Redovi 42-52: Postoje dve glavne funkcije za crtanje: draw(int color): Crta korpu na ekranu na osnovu njene trenutne pozicije (move_offset). draw_food(int color): Crta hranu na ekranu na osnovu njene trenutne pozicije (food_posX, food_posY). Funkcija za čišćenje arene: Redovi 54-57: Funkcija arena_clear() čisti ekran i postavlja move_offset na početnu poziciju. Generisanje hrane: Redovi 58-62: Funkcija generate() koristi se za generisanje nasumične X-koordinate (food_posX) na kojoj će hrana početi da pada. Prikaz kraja igre: Redovi 63-72: Funkcija game_over() se koristi za prikazivanje poruke “KRAJ” na ekranu i rezultata igrača nakon što igrač propusti hranu. Glavna funkcija main(): Redovi 74-151: Ova funkcija inicijalizuje portove i tajmer, postavlja početne vrednosti za igru, i upravlja glavnom logikom igre kroz petlju do-while: Redovi 88-90: Inicijalizacija igre i postavljanje početnog stanja (state na GENERATE). Redovi 91-150: Glavna petlja igre: Pomeranje korpe levo/desno: Pritiskom na tastere PORTC.7 i PORTC.6, korpa se pomera levo ili desno. Funkcija draw(0) briše staru poziciju korpe, a draw(1) crta korpu na novoj poziciji. GENERATE: Generiše novu hranu na vrhu ekrana i postavlja igru u stanje DROP. DROP: Hrana počinje da pada prema korpi. Ako hrana dođe do dna ekrana (food_PosY >= 52), proverava se da li je uhvaćena (EAT) ili promašena (MISSED). EAT: Ako je hrana uhvaćena, rezultat (score) se povećava i igra se vraća u stanje GENERATE. MISSED: Ako je hrana promašena, prikazuje se kraj igre i igra prelazi u stanje IDLE. Ukratko, ovaj kod implementira jednostavnu igru “Hvatanje”, u kojoj igrač pomera korpu levo i desno kako bi uhvatio hranu koja pada sa vrha ekrana. Ako igrač propusti hranu, igra se završava i prikazuje se konačni rezultat.

20. 7-SEGMENTNI DISPLEJ

Cilj ove vežbe je implementirati 4-cifreni brojač na sedmosegmentnom displeju. Na svakom prekidu, osvežava se jedna cifra, dok se broj postepeno povećava od 0000 do 9999 i kad dođe do kraja kreće ispočetka. Na ploči je potrebno uključiti prekidače SW4.1., SW4.2., SW4.3. i SW4.4. za sedmosegmentni displej.




```
1.  #include “Display_Utils.h”
2.
3.  unsigned short shifter, portd_index;
4.  unsigned int   digit, number;
5.  unsigned short portd_array[4];
6.
7.  unsigned short mask(unsigned short num) {
8.    switch (num) {
9.      case 0 : return 0x3F;
10.     case 1 : return 0x06;
11.     case 2 : return 0x5B;
12.     case 3 : return 0x4F;
13.     case 4 : return 0x66;
14.     case 5 : return 0x6D;
15.     case 6 : return 0x7D;
```

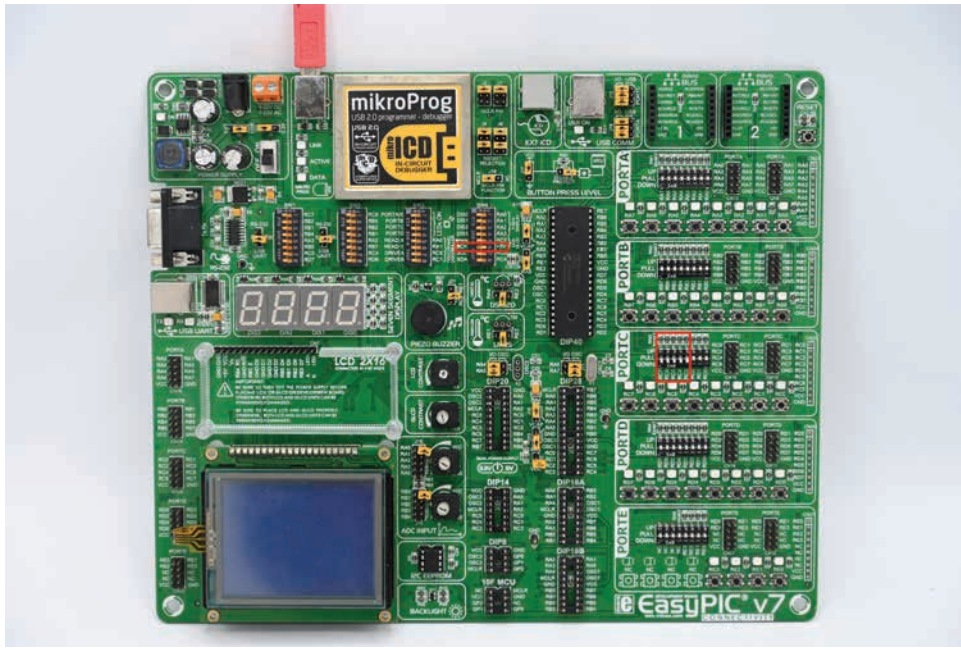
```
16.     case 7 : return 0x07;
17.     case 8 : return 0x7F;
18.     case 9 : return 0x6F;
19.   }
20. }
21.
22. void interrupt() {
23.   LATA = 0;
24.   LATD = portd_array[portd_index];
25.   LATA = shifter;
26.
27.   shifter <<= 1;
28.   if (shifter > 8u)
29.     shifter = 1;
30.     portd_index++ ;
31.   if (portd_index > 3u)
32.     portd_index = 0;
33.     TMR0L  =   0;
34.     TMR0IF_bit = 0;
35. }
36.
37. void main() {
38.   ANSELA = 0;
39.   ANSELD = 0;
40.
41.   TRISA = 0;
42.   LATA  = 0;
43.   TRISD = 0;
44.   LATD  = 0;
45.
46.   T0CON = 0xC4;
47.   TMR0L = 0;
48.   digit = 0;
49.   portd_index = 0;
50.   shifter = 1;
51.
52.   number = 0000;
53.   GIE_bit = 1;
54.   TMR0IE_bit = 1;
55.
56.   do {
57.     digit = number / 1000 ;
58.     portd_array[3] = mask(digit);
59.     digit = (number / 100) % 10;
60.     portd_array[2] = mask(digit);
61.     digit = (number / 10) % 10;
```

```
62.     portd_array[1] = mask(digit);
63.     digit = number % 10;
64.     portd_array[0] = mask(digit);
65.
66.     number++ ;
67.     if (number > 9999u)
68.       number = 0000;
69.
70.   } while(1);
71. }
```

Uključivanje biblioteka: Red 1: Uključuje se datoteka “Display_Utils.h”, koja sadrži definicije i funkcije za rad sa 7-segmentnim displejom. Deklaracija promenljivih: Red 3: shifter je promenljiva koja se koristi za izbor cifre koja će biti prikazana; portd_index drži trenutni indeks cifre; digit je promenljiva koja čuva trenutnu cifru koja se obrađuje; number je promenljiva koja sadrži broj koji se prikazuje. Red 5: portd_array je niz koji čuva vrednosti koje će biti prikazane na svakom segmentu (od cifara na displeju). Funkcija mask: Redovi 7-20: Funkcija mask pretvara cifre (0-9) u odgovarajuće binarne kodove za 7-segmentni displej. Svaka cifra od 0 do 9 ima svoj odgovarajući binarni kod koji se koristi za osvetljavanje odgovarajućih segmenata displeja. Na primer, 0x3F predstavlja cifru 0, gde su svi segmenti osim jednog osvetljeni. Interrupt rutina: Redovi 22-35: Interrupt rutina koja se izvršava svaki put kada tajmer istekne. Red 23: LATA = 0; - Isključuje sve cifre na displeju. Red 24: LATD = portd_array[portd_index]; - Postavlja vrednost trenutne cifre na odgovarajući port. Red 25: LATA = shifter; - Uključuje cifru koja je trenutno odabrana. Redovi 27-32: Pomeranje shifter varijable levo za jedan bit (tj. prelazak na sledeću cifru). Ako je pomerenje veće od 8 (sve 4 cifre su prikazane), resetuje se na 1. portd_index se koristi za indeksiranje cifre koja se prikazuje, i resetuje se na 0 kada sve cifre budu prikazane. Red 33: TMR0L = 0; - Resetuje tajmer. Red 34: TMR0IF_bit = 0; - Resetuje interrupt flag. Glavna funkcija main: Redovi 37-44: Inicijalizacija portova ANSELA, ANSELD, TRISA, TRISD i njihovih izlaznih vrednosti (LATA, LATD). Red 46: T0CON = 0xC4; - postavlja tajmer TMR0 u konfiguraciju gde se koristi kao izvor vremena za interrupt. Red 47: TMR0L = 0; - Resetuje vrednost tajmera. Redovi 48-50: Postavljaju se inicijalne vrednosti za digit, portd_index i shifter. Red 52: number = 0000; - Postavlja početni broj na 0000. Redovi 53-54: Aktiviraju se globalni interrupt (GIE_bit = 1) i interrupt za tajmer (TMR0IE_bit = 1). Glavna petlja: Redovi 56-70: Petlja koja se beskonačno ponavlja i ažurira vrednosti za prikazivanje na displeju. Red 57: Izračunava se vrednost prve (hiljadite) cifre (digit = number / 1000). Red 58: Ova vrednost se pretvara u odgovarajući binarni kod i smešta u portd_array[3]. Redovi 59-64: Isti postupak se ponavlja za sve ostale cifre (stotine, desetine, jedinice). Red 66: number++ ; - Povećava se vrednost broja za 1. Red 67: Ako broj pređe 9999, resetuje se na 0000. Ukratko, ovaj kod implementira sistem za prikazivanje četvorocifrenog broja na 7-segmentnom displeju. Kroz upotrebu tajmera i interrupt rutine, sve četiri cifre se brzo osvežavaju, stvarajući iluziju da su sve cifre prikazane istovremeno. Broj se automatski povećava, a nakon što dostigne 9999, ponovo se resetuje na 0000.

21. GLCD CRTANJE

Cilj ove vežbe je kreiranje programa za iscrtavanje na GLCD ekranu. Potrebno je implementirati kontrolu kursora pomoću 4 dugmeta kojima kontrolišemo kursor (kretanje gore, dole, levo i desno), dok peto dugme služi za izmenu stanja polja na kome se kursor nalazi – pritiskom na ovaj taster polje na kome se kursor nalazi se zatamnjuje. Pošto se polje zatamni, kursor treba da pređe na najbliže slobodno polje. Kako bi GLCD ekran funkcionisao, na ploči je potrebno uključiti prekidač SW4.6. kao i postaviti prekidače za dugmiće 3,4,5,6 i 7 na portu C u pull down položaj. Takođe je potrebno uključiti Button i Glcd biblioteke.




```
#define UP      0
#define RIGHT   1
#define DOWN    2
#define LEFT    3
#define IDLE    4

char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at LATB0_bit;
sbit GLCD_CS2 at LATB1_bit;
sbit GLCD_RS  at LATB2_bit;
```

```
sbit GLCD_RW  at LATB3_bit;
sbit GLCD_EN  at LATB4_bit;
sbit GLCD_RST at LATB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction  at TRISB2_bit;
sbit GLCD_RW_Direction  at TRISB3_bit;
sbit GLCD_EN_Direction  at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;

int posX;
int posY;
bit color_off;

bit oldstate7, oldstate6, oldstate5, oldstate4, oldstate3;
int flag,left,right;
int state;


void arena_clear(){
  Glcd_Fill(0x00);
}

void main() {

  ANSELB = 0;
  ANSELC = 0;
  ANSELD = 0;
  TRISC=0xFF;

  posX=1;
  posY=1;
  color_off = 1;

  Glcd_Init();
  Glcd_Fill(0x00);
  Glcd_Rectangle(posX-1, posY-1, posX, posY, 1);

  do {
    switch (state){
        case UP:
            if (color_off){
             Glcd_Rectangle(posX-1, posY-1, posX, posY, 0);
            }
            posY--;
```

```
            if (posY<1) posY=1;
            state = IDLE;
            Glcd_Rectangle(posX-1, posY-1, posX, posY, 1);
        break;
        case DOWN:
            if (color_off){
             Glcd_Rectangle(posX-1, posY-1, posX, posY, 0);
            }
            posY++;
            if (posY>63) posY=63;
            state = IDLE;
            Glcd_Rectangle(posX-1, posY-1, posX, posY, 1);
        break;
        case RIGHT:
            if (color_off){
             Glcd_Rectangle(posX-1, posY-1, posX, posY, 0);
            }
            posX++;
            if (posX>127) posX=127;
            state = IDLE;
            Glcd_Rectangle(posX-1, posY-1, posX, posY, 1);
        break;
        case LEFT:
             if (color_off){
             Glcd_Rectangle(posX-1, posY-1, posX, posY, 0);
             }
             posX--;
             if (posX<1) posX=1;
             state = IDLE;
            Glcd_Rectangle(posX-1, posY-1, posX, posY, 1);
        break;
        case IDLE:
        break;
    }

    if (Button(&PORTC, 7, 1, 1)) {
      oldstate7 = 1;
    }
    if (oldstate7 && Button(&PORTC, 7, 1, 0)) {
      state=UP;
      oldstate7 = 0;
    }
          if (Button(&PORTC, 6, 1, 1)) {
            oldstate6 = 1;
          }
          if (oldstate6 && Button(&PORTC, 6, 1, 0)) {
```

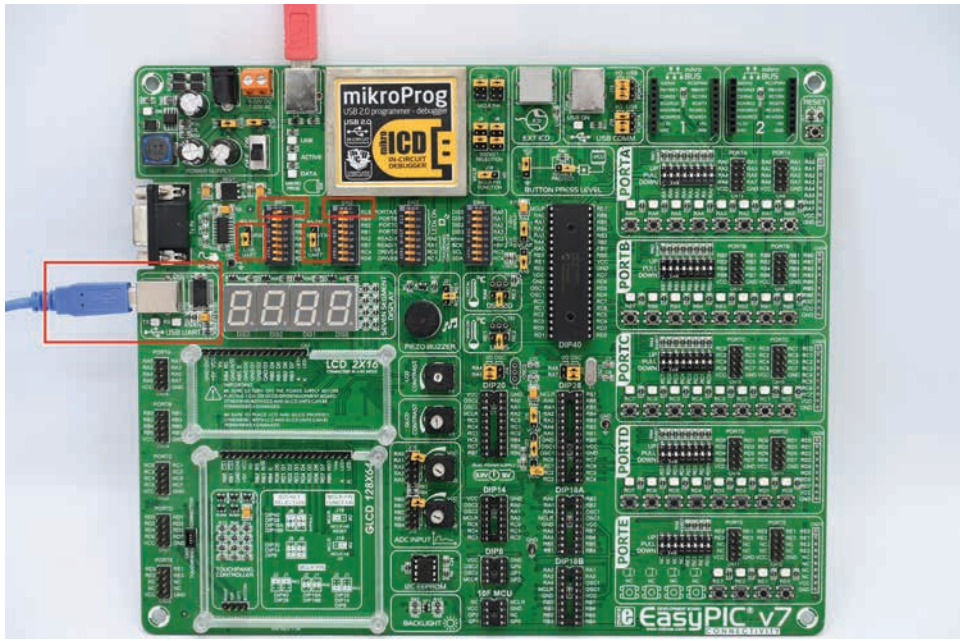
```
104.            state=DOWN;
105.            oldstate6 = 0;
106.          }
107.          if (Button(&PORTC, 5, 1, 1)) {
108.            oldstate5 = 1;
109.          }
110.          if (oldstate5 && Button(&PORTC, 5, 1, 0)) {
111.            state=LEFT;
112.            oldstate5 = 0;
113.          }
114.          if (Button(&PORTC, 4, 1, 1)) {
115.            oldstate4 = 1;
116.          }
117.          if (oldstate4 && Button(&PORTC, 4, 1, 0)) {
118.            state=RIGHT;
119.            oldstate4 = 0;
120.          }
121.          if (Button(&PORTC, 3, 1, 1)) {
122.            oldstate3 = 1;
123.          }
124.          if (oldstate3 && Button(&PORTC, 3, 1, 0)) {
125.            color_off=~color_off;
126.            oldstate3 = 0;
127.          }
128.        } while(1);
129.      }
```

Definisanje konstanti: Redovi 1-5: Definišu se konstante za pravce kretanja markera: gore (UP), desno (RIGHT), dole (DOWN), levo (LEFT), i stanje mirovanja (IDLE). Povezivanje pinova za GLCD: Redovi 7-21: Ovi redovi povezuju portove i signale mikrokontrolera sa grafičkim LCD displejom (GLCD). Deklaracija promenljivih: Red 23-24: posX i posY su promenljive koje čuvaju trenutne koordinate markera na GLCD ekranu. Red 25: color_off je bit promenljiva koja određuje da li će marker trenutno crtati (uključeno) ili brisati (isključeno) piksele. Redovi 27-29: oldstate7, oldstate6, oldstate5, oldstate4, oldstate3, flag, left, right, i state su promenljive koje se koriste za praćenje stanja tastera i pravca kretanja. Funkcija za čišćenje arene: Redovi 32-34: arena_clear() funkcija briše ekran (Glcd_Fill(0x00)), uklanjajući sve crtane elemente. Glavna funkcija main(): Redovi 36-41: Ovaj deo koda inicijalizuje portove i postavlja početne vrednosti za igru. Redovi 43-45: Postavljaju se početne koordinate markera na poziciju (1,1) i inicijalno stanje boje markera (color_off = 1), što znači da će marker crtati piksele. Red 47: Inicijalizuje se GLCD. Red 48: Briše se ekran na početku (Glcd_Fill(0x00)). Red 49: Postavlja se početni kvadrat na GLCD na poziciji (1,1). Logika kretanja markera: Redovi 52-90: Ovaj deo koda upravlja kretanjem markera i ažuriranjem ekrana: Redovi 53-60: Ako je trenutni pravac UP, marker se pomera gore. Ako je opcija color_off aktivna, prethodna pozicija markera se briše (Glcd_Rectangle(posX-1, posY-1, posX, posY, 0)). Zatim se pozicija markera ažurira i crta

na novoj poziciji. Redovi 62-69: Ako je trenutni pravac DOWN, marker se pomera dole, uz sličnu logiku kao za UP. Redovi 71-78: Ako je trenutni pravac RIGHT, marker se pomera desno, uz sličnu logiku. Redovi 80-87: Ako je trenutni pravac LEFT, marker se pomera levo, uz sličnu logiku. Red 89: Ako je stanje IDLE, marker ne menja svoju poziciju. Obrada pritiska tastera: Redovi 93-127: Ovaj deo koda obrađuje pritiske tastera: Redovi 93-99: Ako je pritisnut taster povezan na PORTC7, marker menja stanje u UP. Redovi 100-106: Ako je pritisnut taster povezan na PORTC6, marker menja stanje u DOWN. Redovi 107-113: Ako je pritisnut taster povezan na PORTC5, marker menja stanje u LEFT. Redovi 114-120: Ako je pritisnut taster povezan na PORTC4, marker menja stanje u RIGHT. Redovi 121-127: Ako je pritisnut taster povezan na PORTC3, menja se stanje color_off (tj. menja se između crtanja i brisanja piksela). Beskonačna petlja: Red 128: do-while petlja koja se beskonačno ponavlja, omogućavajući kontinuirano kretanje markera i reakcije na pritiske tastera. Ukratko, ovaj kod omogućava korisniku da kontroliše kretanje markera na GLCD ekranu pomoću tastera i da crta ili briše piksele na ekranu. Korisnik može da se kreće u četiri pravca (gore, dole, levo, desno) i da menja stanje markera (crtanje ili brisanje).

22. USB UART – HELLO WORLD

Cilj ove vežbe je uspostavljanje UART komunikacije sa računarom i slanje poruke “Hello World” svakih 1 sekundu. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Takođe treba omogućiti UART komunikaciju na ploči tako što se uključe prekidači na SW1.1. i SW2.1. označeni kao RC7 i RC6, tj. TX i RX linije UART-a kao i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti UART biblioteku.

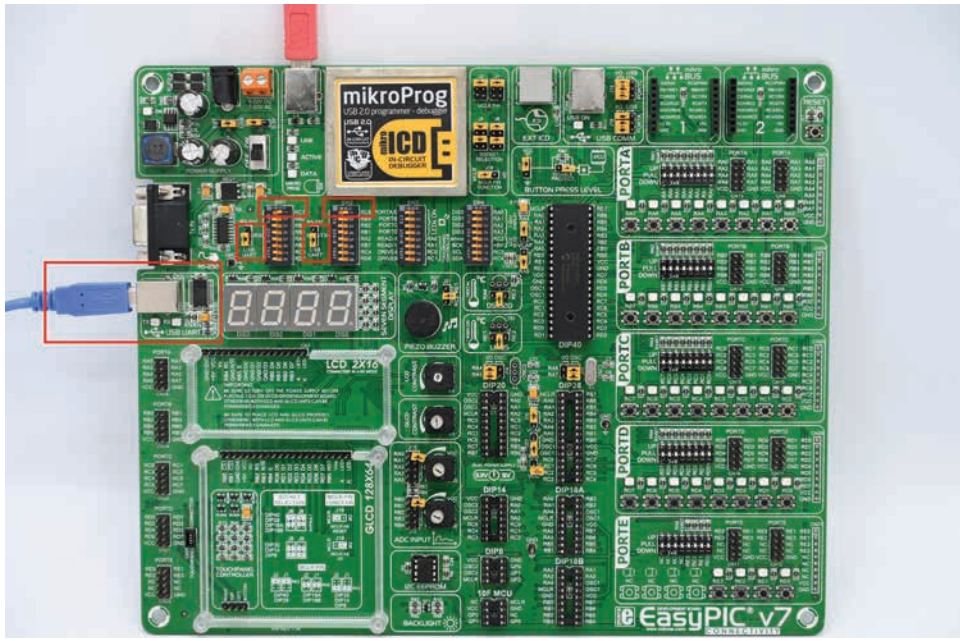



```
1.   void main() {
2.    ANSELC = 0;
3.
4.    UART1_Init(9600);
5.    Delay_ms(100);
6.
7.    UART1_Write_Text(“Start”);
8.    UART1_Write(13);
9.    UART1_Write(10);
10.
11.   while (1) {
12.     UART1_Write_Text(“Hello World “);
13.     Delay_ms(1000);
14.   }
15. }
```

Glavna funkcija main(): Red 1: Počinje definicija glavne funkcije koja se izvršava kada se mikrokontroler uključi. Red 2: ANSELC = 0; postavlja sve pinove na portu C kao digitalne, pošto se ovde ne koristi analogni mod. Inicijalizacija UART-a: Red 4: UART1_Init(9600); inicijalizuje UART1 modul sa baud brzinom od 9600. Ovo omogućava komunikaciju između mikrokontrolera i drugih uređaja (kao što je računar) putem serijskog interfejsa. Red 5: Delay_ms(100); dodaje kratko kašnjenje od 100 milisekundi da bi se UART stabilizovao pre slanja prvih podataka. Slanje poruke "Start": Red 7: UART1_Write_Text("Start"); šalje tekstualnu poruku "Start" preko UART-a. Redovi 8-9: UART1_Write(13); i UART1_Write(10); šalju redom ASCII kodove za povratak na početak linije (CR) i prelazak u novi red (LF), kako bi poruka "Start" bila pravilno prikazana u terminalskom programu. Beskonačna petlja: Redovi 11-14: U beskonačnoj while petlji: Red 12: UART1_Write_Text("Hello World "); šalje poruku "Hello World" preko UART-a. Red 13: Delay_ms(1000); dodaje kašnjenje od 1 sekunde pre nego što se poruka ponovo pošalje, što znači da će poruka "Hello World" biti poslata svake sekunde. Ukratko, ovaj program šalje poruku "Start" jednom prilikom pokretanja mikrokontrolera, a zatim kontinuirano šalje poruku "Hello World" putem serijskog porta svakih 1 sekundu.

23. USB UART – PRIJEM I SLANJE PORUKA

Cilj ove vežbe je uspostavljanje UART komunikacije sa računarom i prijem i slanje poruka odnosno poruku primljenu preko UART-a treba da pošaljemo nazad. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Takođe treba omogućiti UART komunikaciju na ploči tako što se uključe prekidači na SW1.1. i SW2.1. označeni kao RC7 i RC6, tj. TX i RX linije UART-a kao i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti UART biblioteku.



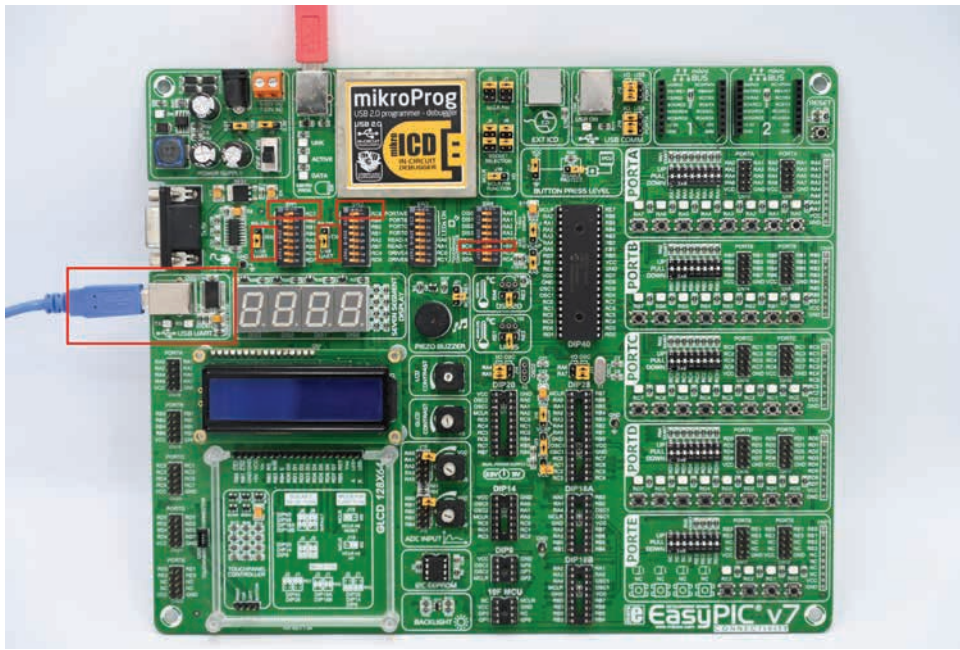

```
1.  char uart_rd;
2.
3.  void main() {
4.    ANSELC = 0;
5.
6.    UART1_Init(9600);
7.    Delay_ms(100);
8.
9.    UART1_Write_Text("Start");
10.   UART1_Write(13);
```

```
11.   UART1_Write(10);
12.
13.   while (1) {
14.     if (UART1_Data_Ready()) {
15.       uart_rd = UART1_Read();
16.       UART1_Write(uart_rd);
17.     }
18.   }
19. }
```

Deklaracija promenljive: Red 1: uart_rd je promenljiva tipa char koja se koristi za skladištenje podataka primljenih putem UART-a. Glavna funkcija main(): Red 3: Početak glavne funkcije gde se vrši konfiguracija i inicijalizacija. Konfiguracija portova: Red 4: ANSELC = 0; isključuje analogne funkcije na portu C, omogućavajući digitalne funkcije. Ovo je važno jer UART koristi digitalne signale. Inicijalizacija UART-a: Red 6: UART1_Init(9600); inicijalizuje UART1 modul sa brzinom od 9600 bita u sekundi. Vreme stabilizacije: Red 7: Delay_ms(100); ubacuje pauzu od 100 milisekundi kako bi se omogućilo stabilizovanje UART modula pre nego što se nastavi sa daljom komunikacijom. Slanje početne poruke: Red 9: UART1_Write_Text("Start"); šalje tekst "Start" putem UART-a, što signalizira početak komunikacije. Redovi 10-11: UART1_Write(13); i UART1_Write(10); šalju specijalne karaktere: 13 je ASCII kod za "carriage return" (povratak na početak reda), a 10 za "line feed" (novi red). Ovi karakteri se često koriste za prelazak u novi red na terminalu. Beskonačna petlja za prijem i slanje podataka: Red 13: while (1) je beskonačna petlja koja omogućava kontinuiranu komunikaciju. Red 14: if (UART1_Data_Ready()) proverava da li su novi podaci stigli putem UART-a. Red 15: uart_rd = UART1_Read(); čita jedan karakter iz UART prijemnika i smešta ga u promenljivu uart_rd. Red 16: UART1_Write(uart_rd); šalje nazad (eho) primljeni karakter. Ovo omogućava korisniku da vidi svaki znak koji je poslao ka mikrokontroleru. Ukratko, osvaj kod implementira jednostavan eho program koji omogućava mikrokontroleru da komunicira sa računarom putem UART protokola. Nakon inicijalnog slanja poruke "Start", program stalno osluškuje dolazne podatke, vraća ih nazad putem UART-a, omogućavajući korisniku da vidi šta je poslao.

24. USB UART – ISPISIVANJE PORUKE NA LCD DISPLEJ

Cilj ove vežbe je uspostavljanje UART komunikacije sa računarom i prijem i slanje poruka odnosno poruku primljenu preko UART-a treba da pošaljemo nazad i takođe da je ispišemo na LCD displej. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Takođe treba omogućiti UART komunikaciju na ploči tako što se uključe prekidači na SW1.1. i SW2.1. označeni kao RC7 i RC6, tj. TX i RX linije UART-a, SW4.6. za LCD kao i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti UART i Lcd biblioteke.



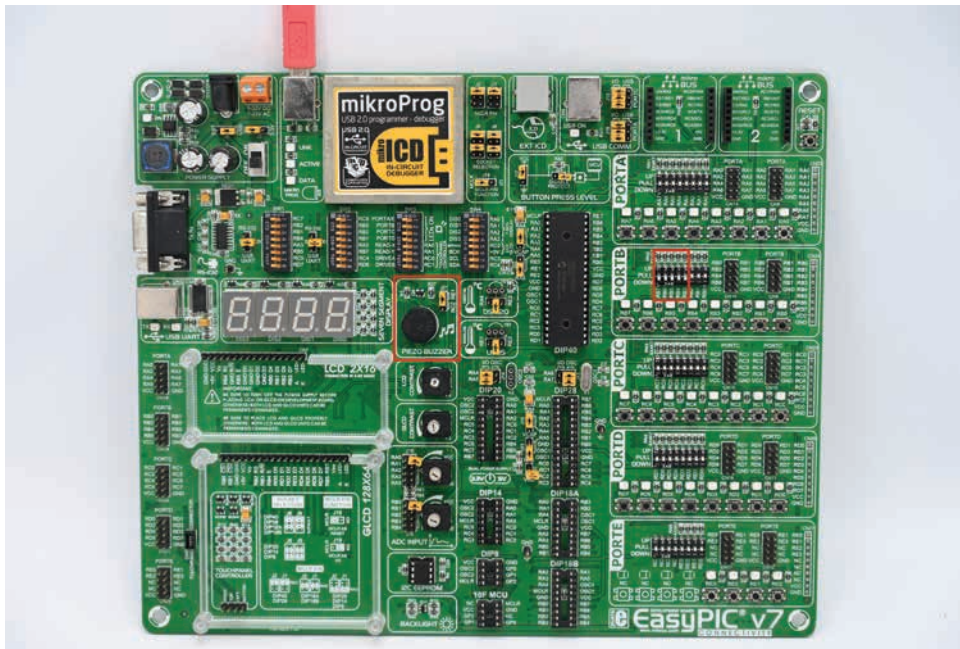

```
1.  sbit LCD_RS at LATB4_bit;
2.  sbit LCD_EN at LATB5_bit;
3.  sbit LCD_D4 at LATB0_bit;
4.  sbit LCD_D5 at LATB1_bit;
5.  sbit LCD_D6 at LATB2_bit;
6.  sbit LCD_D7 at LATB3_bit;
7.
8.  sbit LCD_RS_Direction at TRISB4_bit;
9.  sbit LCD_EN_Direction at TRISB5_bit;
10. sbit LCD_D4_Direction at TRISB0_bit;
```

```
11. sbit LCD_D5_Direction at TRISB1_bit;
12. sbit LCD_D6_Direction at TRISB2_bit;
13. sbit LCD_D7_Direction at TRISB3_bit;
14.
15. char uart_rd;
16.
17. void main() {
18.   ANSELC = 0;
19.   ANSELB = 0;
20.
21.   UART1_Init(9600);
22.   Delay_ms(100);
23.
24.   UART1_Write_Text("Start");
25.   LCD_Init();
26.   Lcd_Cmd(_LCD_CLEAR);
27.   Lcd_Cmd(_LCD_CURSOR_OFF);
28.
29.   while (1) {
30.     if (UART1_Data_Ready()) {
31.       uart_rd = UART1_Read();
32.       UART1_Write(uart_rd);
33.       Lcd_Chr_CP(uart_rd);
34.     }
35.   }
36. }
```

Definisanje LCD pinova: Redovi 1-13: Ovi redovi definišu povezivanje LCD displeja sa mikrokontrolerom. Deklaracija promenljivih: Red 15: uart_rd je promenljiva u kojoj se čuva primljeni karakter sa UART-a. Inicijalizacija u main() funkciji: Redovi 18-19: Onemogućava se analogna funkcija na portovima B i C pomoću ANSELC = 0 i ANSELB = 0, čime se portovi konfigurišu kao digitalni. Red 21: UART1_Init(9600) inicijalizuje UART sa baud rate-om od 9600. Ovo podešava brzinu serijske komunikacije između mikrokontrolera i računara. Red 22: Delay_ms(100) uvodi kratku pauzu od 100 ms kako bi se UART inicijalizacija stabilizovala. Red 24: UART1_Write_Text("Start") šalje tekstualnu poruku "Start" preko UART-a ka računaru, signalizirajući početak rada programa. Red 25-27: Inicijalizuje se LCD displej (Lcd_Init()), briše se ekran (Lcd_Cmd(_LCD_CLEAR)) i isključuje se kursor (Lcd_Cmd(_LCD_CURSOR_OFF)). Glavna petlja: Redovi 29-35: Ova petlja se vrti beskonačno i stalno proverava da li je dostupan podatak na UART-u. Red 30: if (UART1_Data_Ready()) proverava da li je primljen neki podatak putem UART-a. Red 31: Ako je podatak dostupan, UART1_Read() čita taj podatak i čuva ga u promenljivu uart_rd. Red 32: UART1_Write(uart_rd) šalje (echo) isti karakter nazad računaru preko UART-a. Red 33: Lcd_Chr_CP(uart_rd) prikazuje primljeni karakter na trenutnoj poziciji kursora na LCD displeju. Ukratko, ovaj program omogućava mikrokontroleru da komunicira sa računarom putem serijskog (UART) interfejsa. Svi karakteri koji se unesu na računaru šalju se mikrokontroleru, gde se prikazuju na LCD displeju i istovremeno vraćaju računaru.

25. BUZZER

Cilj ove vežbe je koristiti buzzer i implementirati 3 note na pinovima RB7, RB6, RB5, koje predstavlja note mi, fa, sol respektivno. Finalno treba odsvirati pesmu "Na kraj sela" pritiskom na dugme RB3. Dodatno na dugme RB4 odsvirati neku nasumičnu melodiju po želji. Na ploči je potrebno postaviti jumper J21 na RE1 položaj za buzzer i prekidače 3,4,5,6 i 7 na portu B u pulldown položaj. Takođe je potrebno uključiti Button i Sound biblioteke.




```
1.  void Tone1() {
2.   Sound_Play(659, 250); // Frequency = 659Hz, duration = 250ms
3.  }
4.  void Tone2() {
5.   Sound_Play(698, 250); // Frequency = 698Hz, duration = 250ms
6.  }
7.  void Tone3() {
8.   Sound_Play(784, 250); // Frequency = 784Hz, duration = 250ms
9.  }
10. void Melody() { // Melodija žuta kuća
11.  Tone1(); Tone2(); Tone3(); Tone3();
12.  Tone1(); Tone2(); Tone3(); Tone3();
13.  Tone1(); Tone2(); Tone3();
14.  Tone1(); Tone2(); Tone3(); Tone3();
```

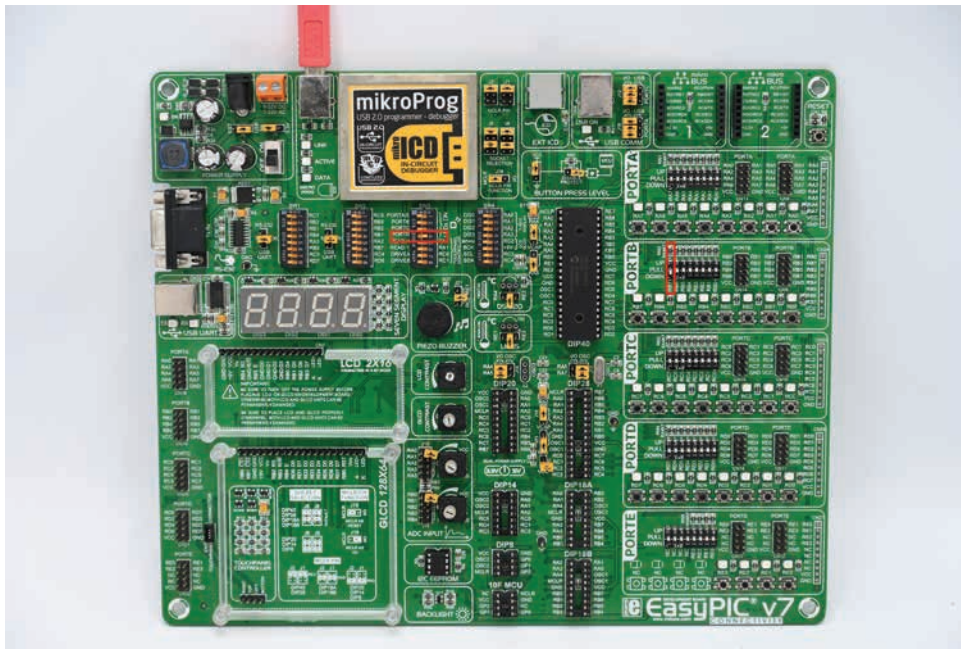
```
15.  Tone1(); Tone2(); Tone3();
16.  Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
17. }
18. void ToneA() {
19.  Sound_Play( 880, 50);
20. }
21. void ToneC() {
22.  Sound_Play(1046, 50);
23. }
24. void ToneE() {
25.  Sound_Play(1318, 50);
26. }
27. void Melody2() {
28.  unsigned short i;
29.  for (i = 9; i > 0; i--) {
30.     ToneA(); ToneC(); ToneE();
31.  }
32. }
33. void main() {
34.  ANSELB = 0;
35.  ANSELE = 0;
36.  TRISB = 0xF8;
37.  Sound_Init(&PORTE, 1);
38.  Sound_Play(880, 1000);
39.  while (1) {
40.     if (Button(&PORTB,7,1,1))
41.        Tone1();
42.     while (RB7_bit) ;
43.     if (Button(&PORTB,6,1,1))
44.         Tone2();
45.     while (RB6_bit) ;
46.     if (Button(&PORTB,5,1,1))
47.         Tone3();
48.     while (RB5_bit) ;
49.     if (Button(&PORTB,4,1,1))
50.         Melody2();
51.     while (RB4_bit) ;
52.     if (Button(&PORTB,3,1,1))
53.         Melody();
54.     while (RB3_bit) ;
55.  }
56. }
```

Definicija funkcija za tonove: Redovi 1-9: Ove funkcije (Tone1, Tone2, Tone3) koriste funkciju Sound_Play za reprodukciju tonova sa različitim frekvencijama: Tone1() svira ton frekvencije 659 Hz u trajanju od 250 ms. Tone2() svira ton frekvencije 698 Hz u trajanju

od 250 ms. Tone3() svira ton frekvencije 784 Hz u trajanju od 250 ms. Definicija funkcija za melodije: Redovi 10-17: Melody() funkcija svira melodiju "Žuta kuća" kombinacijom tonova definisanih u Tone1, Tone2, i Tone3. Svaki ton se pušta redosledom kako je definisano, sa određenim ponavljanjima. Redovi 18-26: ToneA, ToneC, i ToneE funkcije puštaju tonove sa višim frekvencijama: ToneA() svira ton od 880 Hz u trajanju od 50 ms. ToneC() svira ton od 1046 Hz u trajanju od 50 ms. ToneE() svira ton od 1318 Hz u trajanju od 50 ms. Redovi 27-32: Melody2() funkcija ponavlja tonove definisane u ToneA, ToneC, i ToneE devet puta, stvarajući brz i visoko-tonski zvučni efekat. Inicijalizacija i glavni program: Redovi 33-38: U main() funkciji se inicijalizuju portovi i buzzer: ANSELB = 0; i ANSELE = 0; postavljaju portove B i E u digitalni režim rada. TRISB = 0xF8; postavlja niže bitove porta B (RB0-RB4) kao izlazne (0), a više bitove kao ulazne (1). Sound_Init(&PORTE, 1); inicijalizuje buzzer na pin RE1. Sound_Play(880, 1000); svira ton od 880 Hz u trajanju od 1 sekunde kada se program pokrene. Glavna petlja programa: Redovi 39-55: Ova petlja omogućava korisniku da pritiskom na tastere na PORTB svira određene tonove ili melodije: Ako je pritisnut taster na RB7, pušta se ton Tone1(). Ako je pritisnut taster na RB6, pušta se ton Tone2(). Ako je pritisnut taster na RB5, pušta se ton Tone3(). Ako je pritisnut taster na RB4, svira se melodija Melody2(). Ako je pritisnut taster na RB3, svira se melodija Melody(). Svaki put kada se taster pritisne, odgovarajuća funkcija se poziva i ton ili melodija se reprodukuje. Nakon toga, petlja čeka dok se taster ne otpusti pre nego što nastavi sa daljim proverama (while (RBx_bit);). Ukratko, ovaj kod omogućava korisniku da generiše različite tonove i melodije pomoću buzzera. Korišćenjem pet različitih tastera, korisnik može da odabere koji ton ili melodiju želi da pusti.

26. IGRA ODBROJAVANJA

Cilj ove vežbe je napisati program koji redom gasi diode na portu D, i korisnik treba da pritisne dugme RB7 kada su sve diode isključene, u slučaju da je uspeo diode treba da trepću par sekundi i nakon toga sekvenca se ponavlja. Na ploči je potrebno uključiti prekidač SW3.4. i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  unsigned st;
2.  bit oldstate;
3.  bit bl;
4.  void blink(){
5.      LATD = 0xFF;
6.      Delay_ms(500);
7.      LATD = 0x00;
8.      Delay_ms(500);
9.  }
10. void interrupt() {
11.     if (TMR0IF_bit) {
12.         TMR0IF_bit = 0;
13.         if (bl == 0){
14.             st++;
```

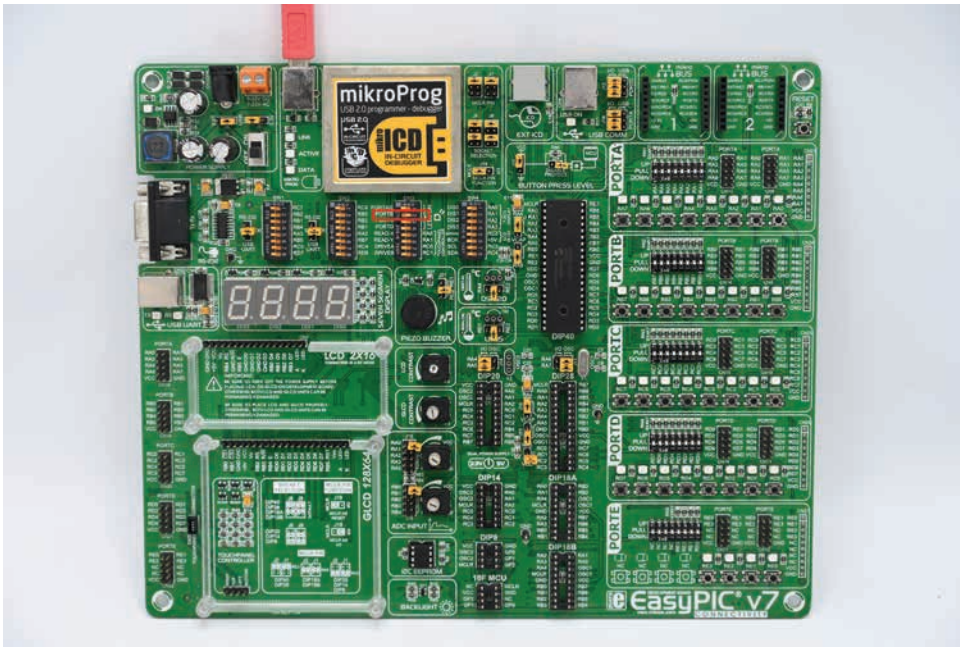
```
15.             LATD = LATD <<1;
16.         }
17.     }
18. }
19. void main() {
20.     TRISB7_bit = 1;
21.     TRISD = 0;
22.     LATD = 0xFF;
23.     oldstate = 0;
24.     TMR0L = 0;
25.     T0CON = 0x84;
26.     GIE_bit = 1;
27.     TMR0IE_bit = 1;
28.     st = 0;
29.     bl = 0;
30.     do {
31.         if(st >= 9){
32.             st = 0;
33.             LATD = 0xFF;
34.         }
35.         if (Button(&PORTB, 7, 1, 1)) {
36.             oldstate = 1;
37.         }
38.         if (oldstate && Button(&PORTB, 7, 1, 0) && (st == 0) ) {
39.             oldstate = 0;
40.             bl = 1;
41.             blink();
42.             blink();
43.             blink();
44.             bl = 0;;
45.         }
46.     } while(1);
47. }
```

Deklaracija promenljivih: Red 1: unsigned st; - Promenljiva st se koristi za praćenje stanja odbrojavanja. Red 2: bit oldstate; - oldstate se koristi za detekciju promena u stanju dugmeta. Red 3: bit bl; - bl kontroliše da li treba izvesti treptanje ili ne. Funkcija blink: Redovi 4-9: Funkcija blink() trepće svim LED diodama tri puta: Uključuje sve LED diode na portu D (LATD = 0xFF), čeka 500 ms, zatim isključuje sve LED diode (LATD = 0x00), i opet čeka 500 ms. Prekidna rutina interrupt: Redovi 10-18: Ova rutina se poziva svaki put kada dođe do prekida tajmera 0 (TMR0): Red 11: TMR0IF_bit je zastavica koja označava prekid i mora se resetovati unutar prekidne rutine. Red 13: Ako je bl jednako 0 (nema treptanja), onda se stanje st povećava i LED diode se pomeraju ulevo. Red 15: LATD = LATD <<1; - Ova linija koda pomera osvetljenje LED dioda ulevo. Glavna funkcija main: Redovi 19-29: Inicijalizacija i konfiguracija sistema: Red 20: Postavlja TRISB7 kao ulazni

pin (gde je dugme priključeno). Red 21: Postavlja port D kao izlazni (TRISD = 0;). Red 22: Svi LED-ovi na portu D se uključuju (LATD = 0xFF). Redovi 24-27: Postavljanje tajmera 0 za generisanje prekida svake sekunde (ili odgovarajućeg intervala) i omogućavanje globalnih i tajmer prekida. Red 28-29: Postavljanje početnog stanja za promenljive st i bl. Glavna petlja do-while: Red 30: Beskonačna petlja koja obavlja glavnu logiku igre: Red 31-34: Kada st dostigne 9 (što znači da je odbrojavanje završeno), ponovo postavlja st na 0 i uključuje sve LED diode. Red 35-37: Detekcija pritiska dugmeta (Button(&PORTB, 7, 1, 1)), gde se oldstate postavlja na 1 kada se dugme pritisne. Redovi 38-44: Ako je dugme otpušteno (Button(&PORTB, 7, 1, 0)) i odbrojavanje je završeno (st == 0): Podesi bl = 1 da bi zaustavio pomeranje LED dioda. Izvrši tri treptanja pozivajući blink() tri puta. Nakon treptanja, bl se ponovo postavlja na 0, što omogućava nastavak odbrojavanja. Ukratko, ovaj program implementira jednostavnu igru odbrojavanja koristeći LED diode priključene na port D mikrokontrolera. LED diode se sukcesivno gase pomeranjem ulevo sa svakim prekidom tajmera. Kada se pritisne dugme, ako je odbrojavanje završeno (sve diode su isključene), sve diode trepnu tri puta pre nego što započne novo odbrojavanje.

27. BITSKA ŠTOPERICA

Cilj ove vežbe je napisati program koji prikazuje broj proteklih sekundi na jednom portu, od 0 do 256 sekundi u binarnom zapisu, s obzirom da postoje 8 dioda na jednom portu koristiti tajmer. Na ploči je potrebno uključiti prekidač SW3.2.



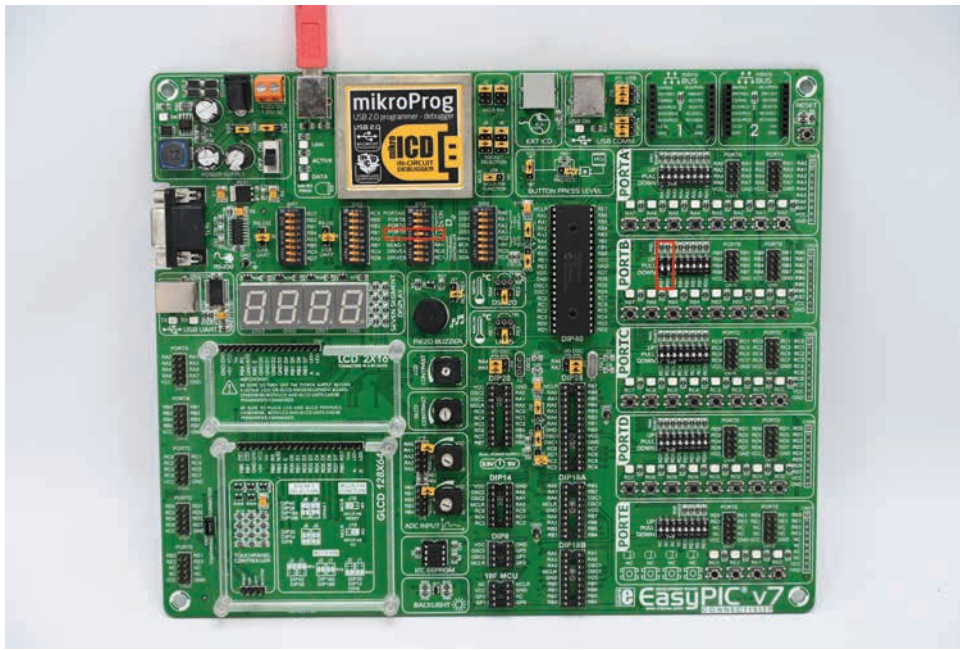

```
1.  unsigned st;
2.
3.  void Interrupt(){
4.    if (TMR0IF_bit){
5.      TMR0IF_bit = 0;
6.      TMR0H    = 0x0B;
7.      TMR0L    = 0xDC;
8.      st++;
9.      LATB = st;
10.     if(st >= 255){
11.       st = 0;
12.     }
13.   }
14. }
15.
16. void main() {
```

```
17.   TRISB = 0;
18.   LATB = 0xFF;
19.   T0CON  = 0x85;
20.   TMR0H  = 0x0B;
21.   TMR0L  = 0xDC;
22.   GIE_bit    = 1;
23.   TMR0IE_bit     = 1;
24.   st = 0;
25.   do {
26.   } while(1);
27. }
```

Deklaracija promenljive: Red 1: unsigned st; - Deklaracija promenljive st, koja se koristi za čuvanje trenutne vrednosti štoperice. Rukovanje prekidom: Redovi 3-13: Ova funkcija se poziva kada tajmer TMR0 izazove prekid. Red 4: if (TMR0IF_bit){ - Proverava se da li je došlo do prekida od TMR0 (flag bit TMR0IF je postavljen). Red 5: TMR0IF_bit = 0; - Resetuje se flag prekida, omogućavajući sledeći prekid. Redovi 6-7: TMR0H = 0x0B; i TMR0L = 0xDC; - Ponovno postavljanje vrednosti tajmera tako da on izazove prekid nakon određenog vremena. Red 8: st++; - Inkrementira se vrednost štoperice. Red 9: LATB = st; - Vrednost štoperice se prikazuje na portu B. Red 10-12: if(st >= 255) - Ako je štoperica dostigla maksimalnu vrednost (255), resetuje se na 0. Inicijalizacija i glavni program: Red 16: void main() - Početak glavne funkcije programa. Red 17: TRISB = 0; - Port B se postavlja kao izlazni port. Red 18: LATB = 0xFF; - Inicijalna vrednost na portu B je postavljena na 255 (sve diode uključene). Red 19: T0CON = 0x85; - Konfiguracija tajmera TMR0: Tajmer TMR0 se postavlja na 16-bitni mod, koristi se interna frekvencija, predskaler se postavlja na 1:64. Redovi 20-21: TMR0H = 0x0B; i TMR0L = 0xDC; - Inicijalne vrednosti tajmera za generisanje prekida na svakih 50 ms (ova vrednost se može prilagoditi zavisno od frekvencije takta i željenog intervala). Red 22: GIE_bit = 1; - Globalni prekidi se omogućavaju. Red 23: TMR0IE_bit = 1; - Omogućava se prekid od tajmera TMR0. Red 24: st = 0; - Početna vrednost štoperice je postavljena na 0. Red 25-26: Glavna petlja programa je beskonačna (while(1);), što znači da će program nastaviti da radi dok se ne isključi. Ukratko, ovaj program implementira jednostavnu bitsku štopericu koristeći tajmer TMR0 mikrokontrolera. Tajmer generiše prekid na svakih 50 ms (ili drugu vrednost zavisno od konfiguracije), inkrementira vrednost st i prikazuje je na portu B. Kada štoperica dostigne vrednost 255, vraća se na 0 i nastavlja brojanje ispočetka.

28. PROMENA FREKVENCIJE TREPTANJA

Cilj ove vežbe je napisati program kojim se ostvaruje treptanje (blinkanje) svih dioda na portu D sa frekvencijom koja može da se podešava uz pomoć tastera. Pri pokretanju programa frekvencija treptanja treba da bude 5 sekundi. Svakim pritiskom na taster RB7 ova frekvencija može da se smanji za 1 sekund, a pritiskom na RB6 može da se poveća za 1 sekund. Frekvencija treptanja uvek ostaje u opsegu od 0 do 10s. Na ploči je potrebno uključiti prekidač SW3.4. i postaviti prekidače 6 i 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  bit oldstate7, oldstate6;
2.  int freq,cnt;
3.  void interrupt() {
4.      if (TMR0IF_bit) {
5.          TMR0IF_bit = 0;
6.          TMR0H    = 0x3C;
7.          TMR0L    = 0xB0;
8.          cnt++;
9.      }
10. }
11. void main() {
```

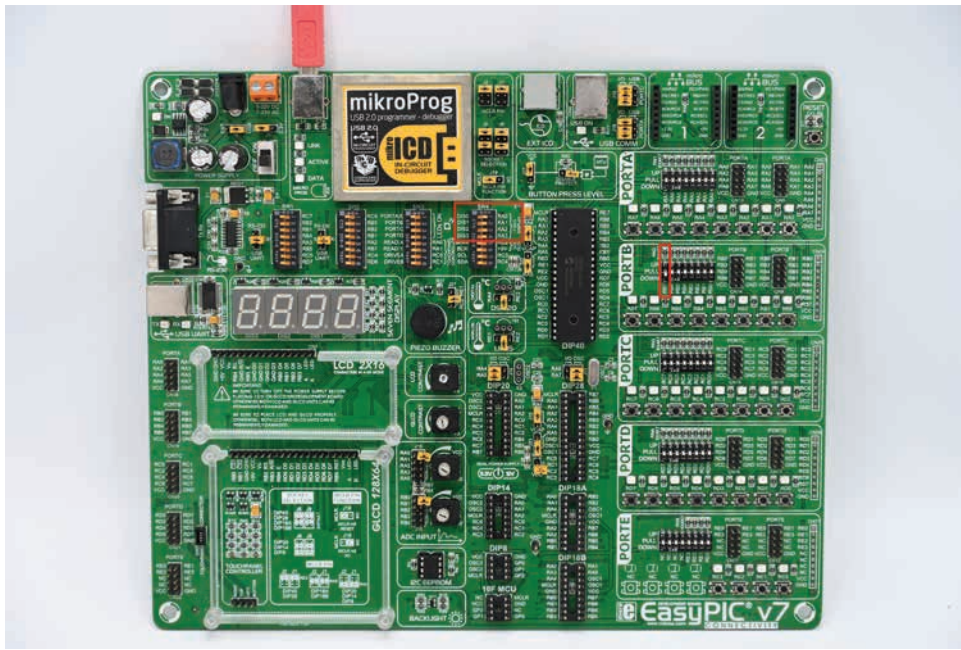
```
12.     TRISD = 0;
13.     LATD = 0xFF;
14.     TRISB7_bit = 1;
15.     TRISB6_bit = 1;
16.     T0CON    = 0x81;
17.     TMR0H    = 0x3C;
18.     TMR0L    = 0xB0;
19.     GIE_bit = 1;
20.     TMR0IE_bit = 1;
21.     oldstate6 = 0;
22.     oldstate7 = 0;
23.     cnt = 0;
24.     freq = 5;
25.     do {
26.         if (Button(&PORTB, 7, 1, 1)) {
27.             oldstate7 = 1;
28.         }
29.         if (oldstate7 && Button(&PORTB, 7, 1, 0) && (freq>=0)) {
30.             freq--;
31.             oldstate7 = 0;
32.         }
33.         if (Button(&PORTB, 6, 1, 1)) {
34.             oldstate6 = 1;
35.         }
36.          if (oldstate6 && Button(&PORTB, 6, 1, 0) && (freq
                <= 10)) {
37.             freq++;
38.             oldstate6 = 0;
39.         }
40.         if (cnt >= freq){
41.             LATD=~LATD;
42.             cnt=0;
43.         }
44.     } while(1);
45. }
```

Definisanje promenljivih: Red 1: oldstate7 i oldstate6 su promenljive koje prate stanje dugmadi na portovima RB7 i RB6, kako bi se detektovalo da li je dugme pritisnuto i pušteno. Red 2: freq predstavlja trenutnu frekvenciju treptanja LED dioda, dok cnt predstavlja brojač koji se koristi za praćenje broja prebrojavanja tajmera. Interrupt rutina: Redovi 3-10: Ova rutina se pokreće svaki put kada tajmer TMR0 pređe svoju vrednost i generiše prekid. Red 4: Proverava se zastavica prekida TMR0IF_bit kako bi se potvrdilo da je prekid izazvao tajmer TMR0. Red 5: Resetuje se zastavica prekida da bi se omogućilo dalje praćenje. Redovi 6-7: Postavljaju početne vrednosti za tajmer TMR0 (0x3CB0), što omogućava periodični prekid sa određenom frekvencijom. Red 8: cnt++ se inkrementira

svaki put kada se dogodi prekid, što prati koliko puta se tajmer prekidao. Inicijalizacija u main funkciji: Redovi 12-15: Podešavanje portova: Port D se konfiguriše kao izlaz (TRISD = 0), a port B pinovi 7 i 6 kao ulazi (TRISB7_bit i TRISB6_bit postavljeni na 1). Redovi 16-20: Podešavanje tajmera: T0CON = 0x81; postavlja TMR0 tajmer na rad sa prescaler-om 1:4 i omogućava ga. TMR0H i TMR0L postavljaju početnu vrednost tajmera za period prekida. GIE_bit = 1; omogućava globalne prekide. TMR0IE_bit = 1; omogućava prekide za TMR0 tajmer. Redovi 21-24: Inicijalizuju se promenljive za praćenje stanja dugmadi i frekvencije treptanja. Glavna petlja (do-while): Redovi 26-32: Kontrola frekvencije smanjenjem: Proverava se da li je dugme na RB7 pritisnuto (Button(&PORTB, 7, 1, 1)), čime se postavlja oldstate7 na 1. Kada se dugme otpusti (Button(&PORTB, 7, 1, 0)), a frekvencija je veća od 0, freq se smanjuje, i oldstate7 se resetuje. Redovi 33-39: Kontrola frekvencije povećanjem: Slična logika se koristi za dugme na RB6. Ako je dugme pritisnuto, a zatim pušteno i frekvencija je manja ili jednaka 10, freq se povećava. Redovi 40-43: Treptanje LED dioda: Kada brojač cnt dosegne vrednost freq, port D se invertuje (LATD = ~LATD), čime se menja stanje LED dioda. Brojač cnt se resetuje na 0. Beskonačna petlja: Red 44: Glavna petlja (while(1)) obezbeđuje da program stalno prati dugmad i kontroliše treptanje LED dioda u skladu sa trenutnom frekvencijom. Ukratko, ovaj program omogućava korisniku da kontroliše frekvenciju treptanja LED dioda na portu D pomoću dva dugmeta povezana na portove RB6 i RB7. Tajmer TMR0 generiše prekide sa određenom frekvencijom, a na osnovu broja tih prekida LED diode na portu D menjaju svoje stanje (trepću). Korisnik može da poveća ili smanji brzinu treptanja pritiskom na odgovarajuće dugme.

29. ŠTOPERICA NA 7-SEGMENTNOM DISPLEJU

Cilj ove vežbe je implementirati štopericu na sedmosegmentnom displeju. Kada štoperica nije aktivna, svi segmenti treba da trepću. Štoperica se aktivira pritiskom na taster RB7, a ponovnim pritiskom se zaustavlja. Izmereno vreme je po zaustavljanju vidljivo i potrebno je da segmenti blinkaju. Štoperica može ponovo da se aktivira pritiskom na taster. Na ploči je potrebno uključiti prekidače SW4.1., SW4.2., SW4.3. i SW4.4. za sedmosegmentni displej i postaviti prekidač 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  #include “Display_Utils.h”
2.  unsigned short mask(unsigned short num) {
3.    switch (num) {
4.      case 0 : return 0x3F;
5.      case 1 : return 0x06;
6.      case 2 : return 0x5B;
7.      case 3 : return 0x4F;
8.      case 4 : return 0x66;
9.      case 5 : return 0x6D;
10.     case 6 : return 0x7D;
```

```
11.     case 7 : return 0x07;
12.     case 8 : return 0x7F;
13.     case 9 : return 0x6F;
14.   }
15. }
16. unsigned short shifter, portd_index;
17. unsigned int digit, sec, min;
18. unsigned short portd_array[4];
19. bit oldstate,active;
20. int cnt;
21. void interrupt() {
22.   if(TMR0IF_bit){
23.     cnt++;
24.     LATA = 0;
25.     LATD = portd_array[portd_index];
26.     LATA = shifter;
27.
28.     shifter <<= 1;
29.     if (shifter > 8u)
30.       shifter = 1;
31.
32.       portd_index++ ;
33.     if (portd_index > 3u)
34.       portd_index = 0;
35.       TMR0L = 0;
36.       TMR0IF_bit = 0;
37.   }
38. }
39. void main() {
40.   TRISB7_bit = 1;
41.   ANSELA = 0;
42.   ANSELD = 0;
43.   TRISA = 0;
44.   LATA = 0;
45.   TRISD = 0;
46.   LATD = 0;
47.   T0CON = 0xC4;
48.   TMR0L = 0;
49.   digit = 0;
50.   portd_index = 0;
51.   shifter = 1;
52.   min = 00;
53.   sec = 00;
54.   GIE_bit = 1;
55.   TMR0IE_bit = 1;
56.   oldstate = 0;
```

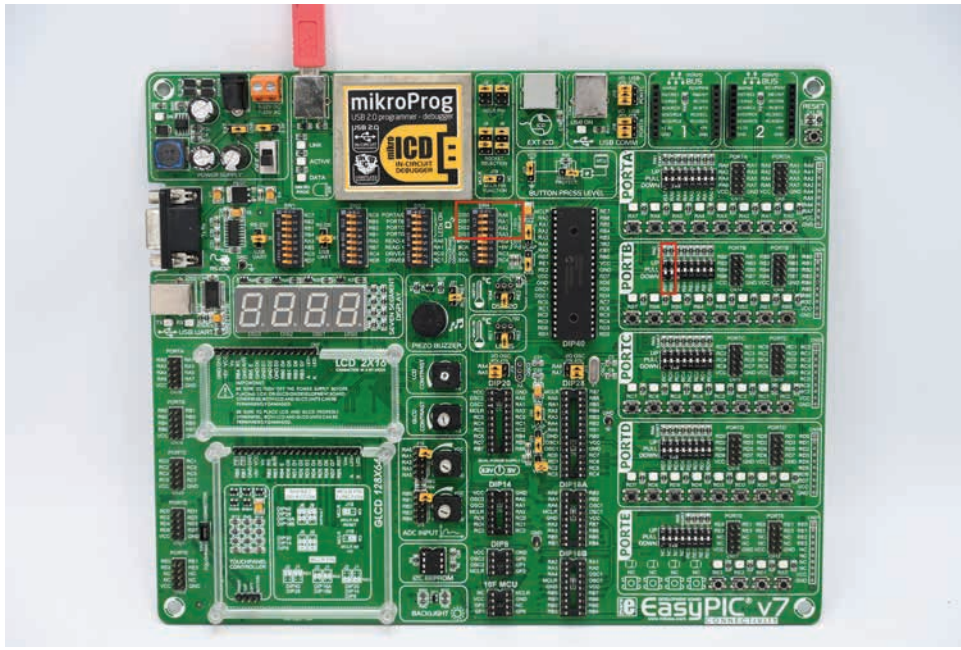
```
57.   active = 0;
58.   do {
59.     if (Button(&PORTB, 7, 1, 1)) {
60.       oldstate = 1;
61.     }
62.     if (oldstate && Button(&PORTB, 7, 1, 0)) {
63.       active = ~active;
64.       if(active){
65.         sec = 0;
66.         min = 0;
67.       }
68.       oldstate = 0;
69.     }
70.     digit = (min / 10u) % 10u;
71.     portd_array[3] = mask(digit);
72.     digit = min % 10u;
73.     portd_array[2] = mask(digit);
74.     digit = (sec / 10u) % 10u;
75.     portd_array[1] = mask(digit);
76.     digit = sec % 10u;
77.     portd_array[0] = mask(digit);
78.     if (cnt >= 1000){
79.       cnt=0;
80.       if (active){
81.         sec++ ;
82.         if (sec == 60u){
83.           sec = 0;
84.           min++;
85.         }
86.       } else{
87.         portd_array[3] = 0x80;
88.         portd_array[2] = 0x80;
89.         portd_array[1] = 0x80;
90.         portd_array[0] = 0x80;
91.         Delay_ms(100);
92.       }
93.     }
94.   } while(1);
95. }
```

Funkcija mask: Redovi 2-15: Funkcija mask vraća odgovarajuću binarnu masku za prikaz broja na 7-segmentnom displeju. Ova maska definiše koji segmenti displeja treba da svetle za dati broj (od 0 do 9). Inicijalizacija promenljivih: Redovi 16-20: shifter: koristi se za sekvencijalno biranje 7-segmentnog displeja (u slučaju multipleksiranja). portd_index: indeks u nizu portd_array, koristi se za odabir cifre koja će biti prikazana. digit: koristi se za skladištenje trenutne cifre koja se prikazuje. sec i min: broje sekunde i minute. portd_

array: niz koji čuva maske za svaku cifru koja će biti prikazana na 7-segmentnom displeju. oldstate i active: promene stanja dugmeta i status štoperice (aktivna/neaktivna). cnt: brojač prekida, koristi se za generisanje vremenskih intervala. Prekidna rutina interrupt: Redovi 21-37: Ova rutina se poziva svaki put kada dođe do prekida od tajmera TMR0. Redovi 24-26: Osvetljava odgovarajuću cifru na 7-segmentnom displeju, koristeći vrednosti iz niza portd_array. Redovi 28-30: Koristi promenljivu shifter za pomeranje aktivnog bita na sledeći displej. Redovi 32-34: Postavlja portd_index na sledeću cifru koju treba prikazati. Redovi 35-36: Resetuje TMR0 brojač i zastavicu prekida. Glavna funkcija main: Redovi 39-46: Inicijalizacija ulazno-izlaznih portova i postavljanje početnih vrednosti za LCD. Redovi 47-57: Inicijalizacija tajmera TMR0, podešavanje promenljivih i uključivanje globalnog prekida (GIE_bit = 1) i prekida od TMR0 (TMR0IE_bit = 1). Glavna petlja: Redovi 58-93: Petlja koja kontinuirano proverava stanje dugmeta, ažurira brojač vremena i prikazuje rezultate na displeju. Redovi 59-69: Proverava pritisak na dugme. Ako je dugme pritisnuto (Button(&PORTB, 7, 1, 1)), promeni stanje štoperice (active). Redovi 70-77: Konvertuje minute i sekunde u maske koje se zatim prikazuju na 7-segmentnom displeju. Redovi 78-92: Ako je proteklo dovoljno vremena (1000 prekida), ažurira brojač sekundi i minuta. Ako je štoperica aktivna (active), broji sekunde, inače trepće ciframa na displeju. Funkcionisanje štoperice: Kada korisnik pritisne dugme, štoperica se resetuje i počinje da broji vreme. Drugi pritisak na dugme zaustavlja štopericu. Kada je štoperica zaustavljena, cifre na displeju trepere. Ukratko, ovaj program implementira štopericu na 7-segmentnom displeju koristeći mikrokontroler. Tajmer TMR0 se koristi za generisanje prekida svakih 1 ms, dok se dugme koristi za pokretanje i zaustavljanje štoperice. Vreme u minutima i sekundama se prikazuje na 7-segmentnom displeju u realnom vremenu.

30. PIN KOD NA 7-SEGMENTNOM DISPLEJU

Cilj ove vežbe je napisati program za ispisivanje željenog broja na sedmosegmentnom displeju. Tasterom RB7 se bira aktivan displej dok se tasterom RB6 povećava vrednost sa vraćanjem na nulu. Označavanje aktivnog segmenta vrši se blinkanjem. Na ovaj način se ispisuje željeni broj na sedmosegmentni displej. Na ploči je potrebno uključiti prekidače SW4.1., SW4.2., SW4.3. i SW4.4. za sedmosegmentni displej i postaviti prekidače 6 i 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  unsigned short shifter, portd_index;
2.  unsigned short portd_array[4], val_array[4];
3.  bit oldstate6, oldstate7,active;
4.  int cnt,selected, value,i, digit;
5.  unsigned short mask(unsigned short num) {
6.    switch (num) {
7.      case 0 : return 0x3F;
8.      case 1 : return 0x06;
9.      case 2 : return 0x5B;
10.     case 3 : return 0x4F;
11.     case 4 : return 0x66;
```

```
12.     case 5 : return 0x6D;
13.     case 6 : return 0x7D;
14.     case 7 : return 0x07;
15.     case 8 : return 0x7F;
16.     case 9 : return 0x6F;
17.   }
18. }
19.
20. void interrupt() {
21.   if(TMR0IF_bit){
22.     cnt++;
23.     LATA = 0;
24.     LATD = portd_array[portd_index];
25.     LATA = shifter;
26.     shifter <<= 1;
27.     if (shifter > 8u)
28.       shifter = 1;
29.       portd_index++ ;
30.     if (portd_index > 3u)
31.       portd_index = 0;
32.       TMR0L = 0;
33.       TMR0IF_bit = 0;
34.   }
35. }
36. void main() {
37.   TRISB7_bit = 1;
38.   TRISB6_bit = 1;
39.   ANSELA = 0;
40.   ANSELD = 0;
41.   TRISA = 0;
42.   LATA = 0;
43.   TRISD = 0;
44.   LATD = 0;
45.   T0CON = 0xC4;
46.   TMR0L = 0;
47.   digit = 0;
48.   portd_index = 0;
49.   shifter = 1;
50.   GIE_bit = 1;
51.   TMR0IE_bit = 1;
52.   for ( i = 0; i < 4; i++ ){
53.     val_array[i] = 0;
54.     portd_array[i] = 0x00;
55.   }
56.   oldstate7 = 0;
57.   oldstate6 = 0;
```

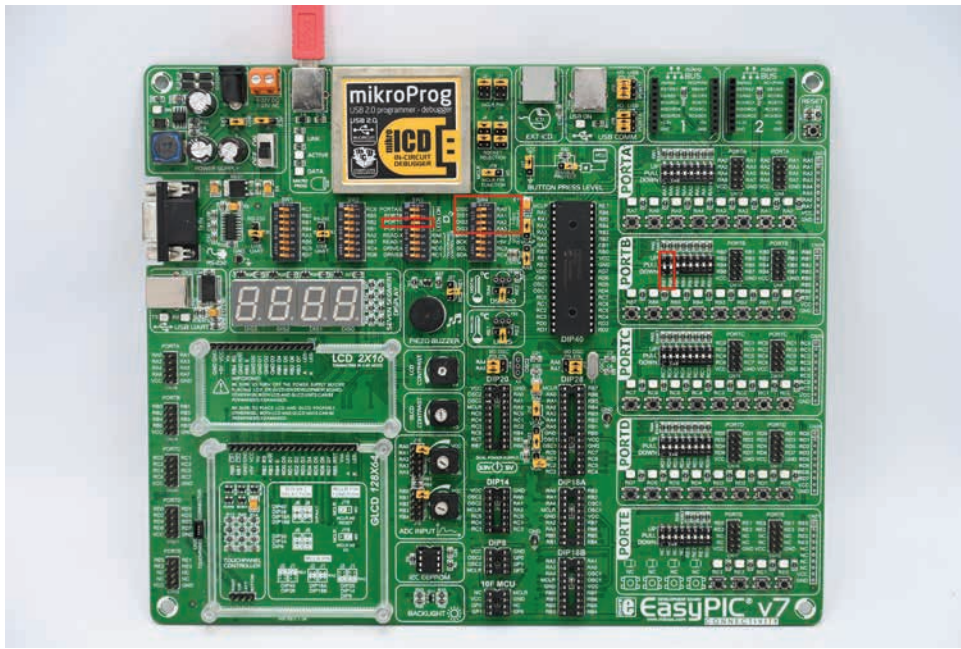
```
58.   active = 0;
59.   selected = 0;
60.   value = 0;
61.   do {
62.     if (Button(&PORTB, 7, 1, 1)) {
63.       oldstate7 = 1;
64.     }
65.     if (oldstate7 && Button(&PORTB, 7, 1, 0)) {
66.       oldstate7 = 0;
67.       selected++;
68.       if (selected ==4) selected =0;
69.       value = val_array[selected];
70.     }
71.     if (Button(&PORTB, 6, 1, 1)) {
72.       oldstate6 = 1;
73.     }
74.     if (oldstate6 && Button(&PORTB, 6, 1, 0)) {
75.       oldstate6 = 0;
76.       value++;
77.       if (value > 9) value =0;
78.       portd_array[selected] = mask(value);
79.       val_array[selected] = value;
80.     }
81.     if (cnt >= 1000){
82.       cnt=0;
83.       if (active){
84.         for ( i = 0; i < 4; i++ )
85.         portd_array[i] = mask(val_array[i]);
86.       } else{
87.         portd_array[selected] = 0x00;
88.         Delay_ms(100);
89.       }
90.       active=~active;
91.     }
92.   } while(1);
93. }
```

Deklaracija promenljivih: Redovi 1-4: Ove promenljive su korišćene za kontrolu i prikaz vrednosti na 7-segmentnim displejima: shifter: Koristi se za pomeranje između 7-segmentnih displeja. portd_index: Indeks koji prati koji displej treba da bude osvežen. portd_array[4]: Niz koji sadrži vrednosti za svaki 7-segmentni displej. val_array[4]: Niz koji sadrži trenutne vrednosti (cifre) za svaki segment displeja. oldstate6, oldstate7, active: Buleanski signali za kontrolu stanja tastera i trepćućeg efekta. cnt, selected, value, i, digit: Promenljive za brojanje, odabir cifre, vrednost cifre i petlje. Funkcija mask: Redovi 5-18: Ova funkcija mapira brojeve 0-9 na odgovarajuće binarne kodove za prikazivanje na 7-segmentnom displeju. Interrupt rutina: Redovi 20-34: Interrupt rutina se poziva kada

dođe do prelivanja tajmera (Timer0): cnt se povećava svakim prekidom. LATA se postavlja na 0 kako bi se isključili svi displeji. LATD se postavlja na vrednost iz portd_array za trenutni segment displeja. LATA se postavlja na odgovarajući bit shifter kako bi se aktivirao tačno jedan displej. shifter se pomera ulevo za sledeći segment. Kada shifter dođe do kraja, resetuje se na početak. TMR0IF_bit se resetuje, signalizirajući da je prekid obrađen. Inicijalizacija u main funkciji: Redovi 36-44: Podešavanje portova i inicijalizacija početnih vrednosti: TRISB7_bit i TRISB6_bit se postavljaju kao ulazni za tastere. ANSELA i ANSELD se postavljaju kao digitalni portovi. TRISA i TRISD se postavljaju kao izlazni portovi. T0CON = 0xC4; - Podešavanje Timer0 (interni tajmer). TMR0L = 0; - Resetovanje početne vrednosti tajmera. Redovi 47-51: Inicijalizacija promenljivih za displeje i aktivacija prekida: shifter počinje sa 1, što znači da prvi displej postaje aktivan. Globalni prekidi i prekidi od tajmera su uključeni. Redovi 52-60: Postavljanje početnih vrednosti za val_array i portd_array (sve cifre su 0). Glavna petlja (do while): Redovi 62-70: Detekcija pritiska tastera za selekciju cifre: Ako je taster PORTB7 pritisnut, selektovana cifra se pomera za jednu udesno. Ako selected dostigne vrednost 4, resetuje se na 0. Redovi 71-79: Detekcija pritiska tastera za promenu vrednosti cifre: Ako je taster PORTB6 pritisnut, vrednost trenutne selektovane cifre se povećava za 1. Kada value dostigne 10, resetuje se na 0. portd_array[selected] se ažurira sa binarnim kodom cifre za prikazivanje na displeju. Redovi 81-91: Trepćući efekat selektovane cifre: Ako je cnt >= 1000, nakon približno 1 sekunde se proverava da li je aktivan efekat treperenja. Ako je active, sve cifre se prikazuju normalno. Ako nije active, selektovana cifra treperi (isključuje se na kratko). active menja stanje između 1 i 0 kako bi omogućio treperenje. Ukratko, ovaj program omogućava unos četvorocifrenog PIN koda pomoću dva tastera i njegov prikaz na 7-segmentnim displejima. Program koristi taster za selekciju cifre i taster za promenu vrednosti te cifre, uz dodatni trepćući efekat koji signalizira selektovanu cifru.

31. VARIJABILNI PREKID

Cilj ove vežbe je napisati program koji pali redom diode na portu C sa određenom frekvencijom koja može biti od 1 do 9 i povećava se i smanjuje putem dugmeta 6 i 7 na portu B. Trenutna frekvencija treba da bude ispisana na sedmosegmentnom displeju. Na ploči je potrebno uključiti prekidače SW4.1., SW4.2., SW4.3. i SW4.4. za sedmosegmentni displej, SW3.3. za diode na portu C i postaviti prekidače 6 i 7 na portu B u pull down položaj. Takođe je potrebno uključiti Button biblioteku.




```
1.  unsigned short mask(unsigned short num) {
2.    switch (num) {
3.      case 0 : return 0x3F;
4.      case 1 : return 0x06;
5.      case 2 : return 0x5B;
6.      case 3 : return 0x4F;
7.      case 4 : return 0x66;
8.      case 5 : return 0x6D;
9.      case 6 : return 0x7D;
10.     case 7 : return 0x07;
11.     case 8 : return 0x7F;
12.     case 9 : return 0x6F;
13.   }
```

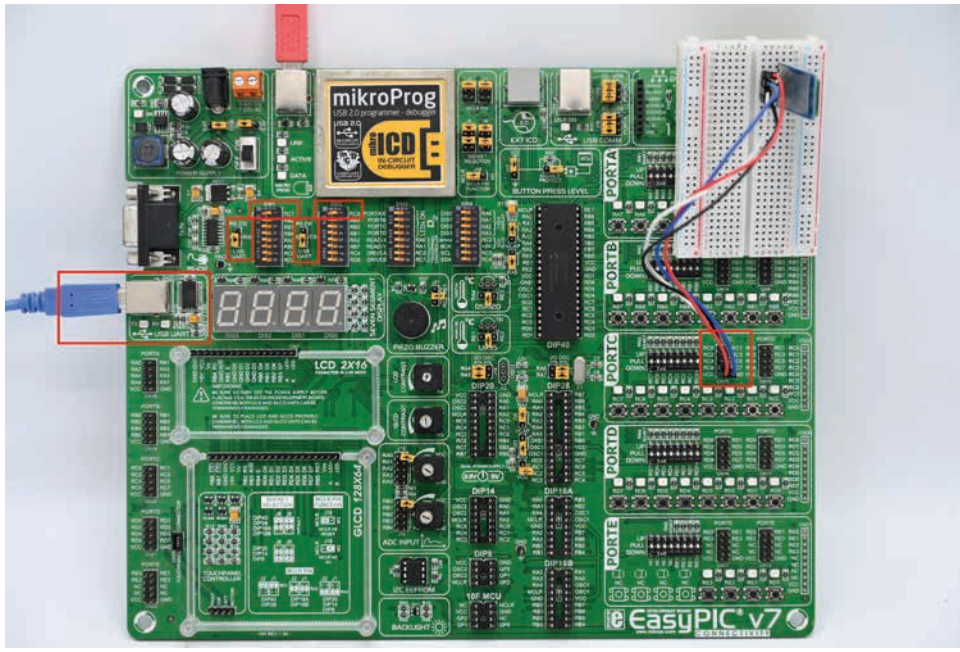
```
14. }
15.
16. bit oldstate7, oldstate6;
17. int freq,cnt,counter;
18. unsigned short vrednost;
19. void interrupt() {
20.   if (TMR0IF_bit) {
21.     TMR0IF_bit = 0;
22.     cnt++;
23.   }
24. }
25. void main() {
26.   TRISC = 0;
27.   LATC = 0xFF;
28.   TRISB7_bit = 1;
29.   TRISB6_bit = 1;
30.   ANSELA = 0;
31.   ANSELD = 0;
32.   TRISA = 0;
33.   LATA = 0;
34.   TRISD = 0;
35.   LATD = 0;
36.   TMR0L = 0;
37.   T0CON = 0x86;
38.   GIE_bit = 1;
39.   TMR0IE_bit = 1;
40.   oldstate6 = 0;
41.   oldstate7 = 0;
42.   cnt = 0;
43.   counter = 0;
44.   freq = 2;
45.   LATA = 1;
46.   do {
47.     if (Button(&PORTB, 7, 1, 1)) {
48.       oldstate7 = 1;
49.     }
50.     if (oldstate7 && Button(&PORTB, 7, 1, 0) && (freq>0)) {
51.       freq--;
52.       oldstate7 = 0;
53.     }
54.     if (Button(&PORTB, 6, 1, 1)) {
55.       oldstate6 = 1;
56.     }
57.     if (oldstate6 && Button(&PORTB, 6, 1, 0) && (freq < 9)) {
58.       freq++;
59.       oldstate6 = 0;
```

```
60.     }
61.     vrednost = mask(freq);
62.     LATD = vrednost;
63.     if (cnt >= freq){
64.       LATC = (0xFF >> counter);
65.       counter--;
66.       cnt=0;
67.     }
68.     if (counter <0){
69.       counter = 8;
70.     }
71.   } while(1);
72. }
```

Funkcija mask: Redovi 1-14: Ova funkcija konvertuje broj u kod za sedmosegmentni displej: Funkcija prima broj (num) od 0 do 9 i vraća odgovarajući heksadecimalni broj koji predstavlja to cifru na sedmosegmentnom displeju. Deklaracija promenljivih: Redovi 16-18: oldstate7 i oldstate6 prate prethodno stanje tastera kako bi detektovali promenu stanja (debouncing). freq predstavlja trenutnu frekvenciju blinkanja. cnt broji prekide tajmera. counter se koristi za kontrolu LED dioda na portu C. vrednost je promenljiva koja drži vrednost koja se prikazuje na sedmosegmentnom displeju. Funkcija interrupt: Redovi 19-24: Ova funkcija se izvršava svaki put kada tajmer 0 izazove prekid (kada brojač dostigne svoju granicu). Kada se prekid dogodi, zastavica prekida TMR0IF_bit se poništava i povećava se cnt. Funkcija main: Redovi 25-45: Inicijalizacija: Postavljaju se odgovarajuće vrednosti za TRIS registre (smer pinova), LAT registre (početne vrednosti izlaza) i ANSEL registre (konfiguracija pinova kao digitalnih). Tajmer 0 (TMR0L) se postavlja na početnu vrednost, a T0CON se konfiguriše za rad tajmera. GIE_bit se postavlja kako bi omogućio globalne prekide, a TMR0IE_bit kako bi omogućio prekid tajmera. Početna frekvencija (freq) se postavlja na 2. Petlja za kontrolu frekvencije: Redovi 47-60: Ova petlja proverava da li su tasteri pritisnuti i, na osnovu toga, povećava ili smanjuje frekvenciju (freq). Taster na RB7 smanjuje frekvenciju, dok taster na RB6 povećava frekvenciju. Kontrola LED dioda i prikaz na sedmosegmentnom displeju: Redovi 61-70: vrednost = mask(freq); - Na osnovu trenutne frekvencije, poziva se funkcija mask koja konvertuje freq u kod za sedmosegmentni displej. LATD = vrednost; - Prikazuje se vrednost na portu D (sedmosegmentni displej). LED diode na portu C se pomeraju udesno za jedan bit na osnovu counter promenljive. Kada counter dođe do 0, resetuje se na početnu vrednost (8), a cnt se resetuje na 0. Beskonačna petlja: Red 71: Petlja while(1) se stalno izvršava, omogućavajući kontinuirano praćenje tastera, kontrolu frekvencije i ažuriranje prikaza na sedmosegmentnom displeju. Ukratko, ovaj program koristi tajmer za generisanje prekida i varijabilno kontrolisanje frekvencije blinkanja LED dioda na portu C. Korisnik može promeniti frekvenciju pomoću tastera povezanih na RB6 i RB7, a trenutna frekvencija se prikazuje na sedmosegmentnom displeju spojenom na port D.

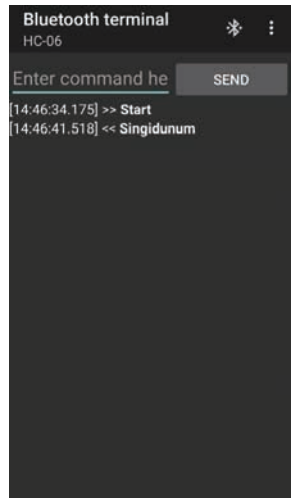
32. BLUETOOTH KOMUNIKACIJA SA MOBILNIM TELEFONOM

Cilj ove vežbe je upoznavanje sa principom Bluetooth modula HC-06. Potrebno je ciklično slati poruku preko Bluetooth-a i pročitati poslatu poruku na drugom uređaju, odnosno mobilnom telefonu. Kako bi verifikovali slanje poruka, treba koristiti bilo koju Bluetooth Terminal aplikaciju na telefonu, koja nam omogućava komunikaciju sa EasyPIC pločom preko Bluetooth-a. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Za realizaciju ove vežbe na ploči je potrebno uključiti prekidače SW1.1. i SW2.1., povezati HC-06 Bluetooth modul, RX pin na RC6 i TX pin na RC7, i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti UART biblioteku.




```
1.  void main() {
2.      ANSELC = 0x00;
3.      UART1_Init(9600);
4.      Delay_ms(100);
5.      UART1_Write_Text("Start\n");
6.      Delay_ms(1000);
7.      while (1) {
```

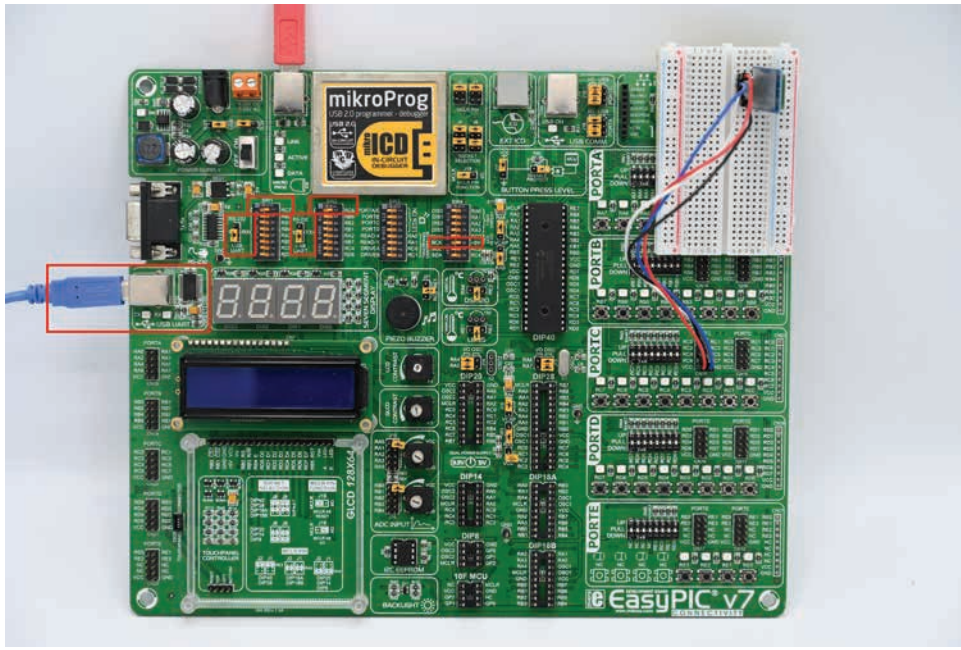
```
8.          UART1_Write_Text(“Singidunum\n”);
9.
10.         Delay_ms(1000);
11.     }
12. }
```



Inicijalizacija: Red 2: ANSELC = 0x00; - Port C se konfiguriše kao digitalni, što znači da će svi pinovi ovog porta biti postavljeni kao digitalni I/O pinovi. Red 3: UART1_Init(9600); - Inicijalizacija UART modula sa baud rate-om od 9600 bita u sekundi. Ovo je standardna brzina koja se koristi za serijsku komunikaciju sa mnogim Bluetooth modulima. Red 4: Delay_ms(100); - Pauza od 100 ms kako bi se omogućilo da se UART modul stabilizuje nakon inicijalizacije. Red 5: UART1_Write_Text(“Start\n”); - Šalje tekstualnu poruku “Start” putem UART-a. Ova poruka se može koristiti za potvrdu da je komunikacija uspostavljena. Red 6: Delay_ms(1000); - Pauza od 1 sekunde (1000 ms) pre nego što se nastavi sa glavnom petljom. Ova pauza omogućava dovoljno vremena za obradu prethodne poruke na strani prijemnika (npr. mobilnog telefona). Glavna petlja (while(1)): Red 8: UART1_Write_Text(“Singidunum\n”); - Kontinuirano šalje tekstualnu poruku “Singidunum” putem UART-a. Svaka poruka je završena sa \n (novi red) što signalizira kraj poruke. Ova poruka će se prikazivati na terminalu mobilnog telefona ili aplikaciji koja je povezana putem Bluetooth-a. Red 9: Delay_ms(1000); - Pauza od 1 sekunde između slanja poruka. Ova pauza omogućava da se poruka pravilno prenese i da se izbegne preopterećenje prijemnika. Ukratko, ovaj program omogućava jednostavnu Bluetooth komunikaciju između mikrokontrolera i mobilnog telefona. Program inicijalizuje UART komunikaciju sa brzinom od 9600 bita u sekundi i kontinuirano šalje tekstualnu poruku “Singidunum” putem Bluetooth modula.

33. LCD ISPIS PREKO BLUETOOTH-A

Cilj ove vežbe je ispis poruke poslate sa mobilnog telefona preko Bluetooth-a na LCD displej, svaka nova poslata poruka treba da bude ispisana od početka. Kako bi slali poruke, treba koristiti bilo koju Bluetooth Terminal aplikaciju na telefonu, koja nam omogućava komunikaciju sa EasyPIC pločom preko Bluetooth-a. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Za realizaciju ove vežbe na ploči je potrebno uključiti prekidače SW1.1., SW2.1. i SW4.6., povezati HC-06 Bluetooth modul, RX pin na RC6 i TX pin na RC7, i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti Lcd i UART biblioteke.



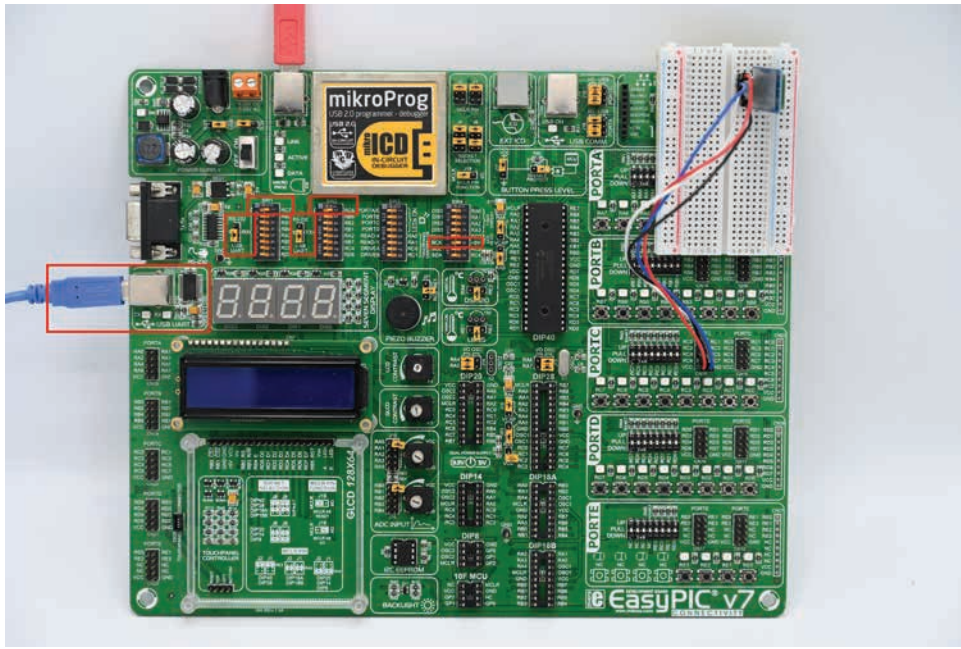

```
1.  sbit LCD_RS at LATB4_bit;
2.  sbit LCD_EN at LATB5_bit;
3.  sbit LCD_D4 at LATB0_bit;
4.  sbit LCD_D5 at LATB1_bit;
5.  sbit LCD_D6 at LATB2_bit;
6.  sbit LCD_D7 at LATB3_bit;
7.  sbit LCD_RS_Direction at TRISB4_bit;
8.  sbit LCD_EN_Direction at TRISB5_bit;
9.  sbit LCD_D4_Direction at TRISB0_bit;
10. sbit LCD_D5_Direction at TRISB1_bit;
11. sbit LCD_D6_Direction at TRISB2_bit;
```

```
12. sbit LCD_D7_Direction at TRISB3_bit;
13.
14. char uart_rd;
15. void main(){
16.     ANSELB = 0;
17.     ANSELC = 0;
18.     UART1_Init(9600);
19.     Delay_ms(100);
20.     UART1_Write_Text("Start\n");
21.     Delay_ms(1000);
22.     Lcd_Init();
23.     Lcd_Cmd(_LCD_CLEAR);
24.     Lcd_Cmd(_LCD_CURSOR_OFF);
25.     while(1) {
26.         if (UART1_Data_Ready()) {
27.             UART1_Read_Text(uart_rd,"\n",16);
28.             Lcd_Cmd(_LCD_CLEAR);
29.             Lcd_Out(1,1,uart_rd);
30.         }
31.     }
32. }
```

Definisanje pinova za LCD: Redovi 1-12: Ovi redovi definišu signale i pravce pinova na mikrokontroleru koji su povezani sa LCD displejem. Inicijalizacija promenljivih: Red 14: uart_rd je promenljiva u kojoj će se čuvati podatak koji se prima putem UART-a. Inicijalizacija u funkciji main: Red 16: ANSELB = 0; - Port B se postavlja kao digitalni. Red 17: ANSELC = 0; - Port C se takođe postavlja kao digitalni. Red 18: UART1_Init(9600); - Inicijalizuje UART modul za komunikaciju na 9600 bita u sekundi (baud rate). Ovo je standardna brzina za mnoge Bluetooth module. Red 19: Delay_ms(100); - Kratko odlaganje kako bi se omogućilo da se UART modul stabilizuje nakon inicijalizacije. Red 20: UART1_Write_Text("Start\n"); - Šalje tekstualni podatak "Start\n" preko UART-a, obaveštavajući Bluetooth uređaj da je sistem spreman. Red 21: Delay_ms(1000); - Dodatno odlaganje da bi se osiguralo da je poruka uspešno poslata i primljena. Redovi 22-24: Inicijalizacija LCD displeja, brisanje ekrana i isključivanje kursora. Beskonačna petlja za prijem podataka i prikaz na LCD-u: Red 25: while(1) - Beskonačna petlja koja se stalno izvršava. Red 26: if (UART1_Data_Ready()) - Proverava da li postoji podatak koji je stigao putem UART-a. Red 27: UART1_Read_Text(uart_rd,"\n",16); - Čita primljeni tekstualni podatak sa UART-a i smešta ga u promenljivu uart_rd. Čitanje se nastavlja dok ne naiđe na karakter za kraj linije (\n) ili dok ne pročita maksimalno 16 karaktera. Red 28: Lcd_Cmd(_LCD_CLEAR); - Briše ekran LCD displeja. Red 29: Lcd_Out(1,1,uart_rd); - Prikazuje primljeni tekst na LCD-u, u prvom redu od prve pozicije. Ukratko, ovaj program demonstrira kako koristiti UART komunikaciju za prijem tekstualnih podataka putem Bluetooth modula i prikazivanje tih podataka na LCD displeju.

34. KRETANJE KARAKTERA NA LCD DISPLEJU PUTEM BLUETOOTH-A

Cilj ove vežbe je realizovati osnovu za LCD igricu kontrolisanjem karaktera na LCD-u preko Bluetooth veze. Potrebno je kontrolisati niz karaktera („[=>“) na LCD displeju, treba koristiti bilo koju Bluetooth Terminal aplikaciju na telefonu, koja nam omogućava komunikaciju sa EasyPIC pločom preko Bluetooth-a. Iz aplikacije je potrebno slati sledeće karaktere preko Bluetooth-a i to 2 za gornju strelicu, 8 za donju strelicu, 4 za levu i 6 za desnu. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Za realizaciju ove vežbe na ploči je potrebno uključiti prekidače SW1.1., SW2.1. i SW4.6., povezati HC-06 Bluetooth modul, RX pin na RC6 i TX pin na RC7, i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti Lcd i UART biblioteke.



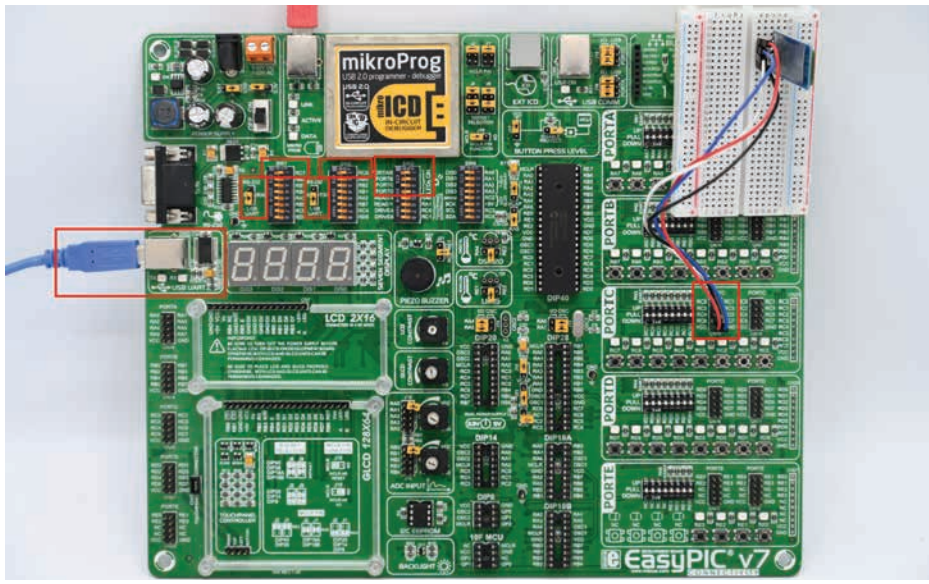

```
1.  sbit LCD_RS at LATB4_bit;
2.  sbit LCD_EN at LATB5_bit;
3.  sbit LCD_D4 at LATB0_bit;
4.  sbit LCD_D5 at LATB1_bit;
5.  sbit LCD_D6 at LATB2_bit;
6.  sbit LCD_D7 at LATB3_bit;
```

```
7.  sbit LCD_RS_Direction at TRISB4_bit;
8.  sbit LCD_EN_Direction at TRISB5_bit;
9.  sbit LCD_D4_Direction at TRISB0_bit;
10. sbit LCD_D5_Direction at TRISB1_bit;
11. sbit LCD_D6_Direction at TRISB2_bit;
12. sbit LCD_D7_Direction at TRISB3_bit;
13. char brod[] = "[=>";
14. char uart_rd;
15. int i=0;
16. void main(){
17.     ANSELB = 0;
18.     ANSELC = 0;
19.     UART1_Init(9600);
20.     Delay_ms(100);
21.     UART1_Write_Text("Start\n");
22.     Lcd_Init();
23.     Lcd_Cmd(_LCD_CLEAR);
24.     Lcd_Cmd(_LCD_CURSOR_OFF);
25.     Lcd_Out(1,1,brod);
26.     while(1) {
27.         if (UART1_Data_Ready()) {
28.             uart_rd = UART1_Read();
29.             if (uart_rd == '2'){
30.                 for ( i = 0; i < 3; i++ ){
31.                     Lcd_Cmd(_LCD_MOVE_CURSOR_LEFT);
32.                 }
33.                 Lcd_Out_Cp("   ");
34.                 Lcd_Cmd(_LCD_FIRST_ROW);
35.                 Lcd_Out_Cp(brod);
36.             } else if (uart_rd == '8'){
37.                 for ( i = 0; i < 3; i++ ){
38.                     Lcd_Cmd(_LCD_MOVE_CURSOR_LEFT);
39.                 }
40.                 Lcd_Out_Cp("   ");
41.                 Lcd_Cmd(_LCD_SECOND_ROW);
42.                 Lcd_Out_Cp(brod);
43.             } else if (uart_rd == '4'){
44.                 Lcd_Cmd(_LCD_SHIFT_LEFT);
45.             }else if (uart_rd == '6'){
46.                 Lcd_Cmd(_LCD_SHIFT_RIGHT);
47.             }
48.         }
49.     }
50. }
```

Definisanje pinova za LCD: Redovi 1-12: Definisanje pinova mikrokontrolera koji su povezani sa LCD displejem. Inicijalizacija promenljivih: Red 13: brod[] je niz karaktera koji predstavlja oblik “broda” koji se prikazuje na LCD-u. Red 14: uart_rd je promenljiva u koju se smešta očitana vrednost sa UART-a (Bluetooth). Red 15: i je brojač koji se koristi u petljama. Inicijalizacija u funkciji main: Red 17: ANSELB = 0; - Konfiguriše Port B kao digitalni. Red 18: ANSELC = 0; - Konfiguriše Port C kao digitalni. Red 19: UART1_Init(9600); - Inicijalizuje UART komunikaciju na 9600 bps za povezivanje sa Bluetooth modulom. Red 20: Delay_ms(100); - Kratko kašnjenje kako bi se omogućilo stabilizovanje UART komunikacije. Red 21: UART1_Write_Text(“Start\n”); - Šalje tekst “Start” preko UART-a na povezani uređaj kako bi se signalizovao početak rada. Redovi 22-24: Inicijalizacija LCD displeja, brisanje ekrana, i isključivanje kursora. Red 25: Lcd_Out(1,1,brod); - Prikazuje “brod” u prvom redu LCD displeja. Glavna petlja programa: Red 26: Program ulazi u beskonačnu petlju (while(1)) kako bi neprestano osluškivao i reagovao na Bluetooth komande. Red 27: if(UART1_Data_Ready()) - Proverava da li je stigao novi podatak preko UART-a. Red 28: uart_rd = UART1_Read(); - Čita primljeni podatak sa UART-a. Pomeranje “broda” po LCD-u u zavisnosti od komande: Redovi 29-35: Ako je primljena komanda ‘8’, “brod” se pomera u prvi red: Najpre se kursor pomera na levo tri puta, briše se prethodna pozicija broda, a zatim se “brod” iscrtava u prvom redu. Redovi 36-42: Ako je primljena komanda ‘2’, “brod” se pomera u drugi red: Slično kao i kod komande ‘8’, ali ovaj put se “brod” iscrtava u drugom redu. Red 43-44: Ako je primljena komanda ‘4’, “brod” se pomera levo na LCD-u koristeći komandu Lcd_Cmd(_LCD_SHIFT_LEFT);. Red 45-46: Ako je primljena komanda ‘6’, “brod” se pomera desno na LCD-u koristeći komandu Lcd_Cmd(_LCD_SHIFT_RIGHT);. Zaključak: Red 48-49: Petlja se ponavlja u beskonačnost, omogućavajući korisniku da neprekidno pomera “brod” putem Bluetooth komandi. Ukratko, ovaj program omogućava korisniku da kontroliše pomeranje karaktera (“broda”) na LCD displeju koristeći Bluetooth komunikaciju. Komande ‘8’, ‘2’, ‘4’, i ‘6’ pomeraju “brod” gore, dole, levo, i desno respektivno, pružajući interaktivan način za upravljanje prikazom na LCD displeju.

35. UPRAVLJANJE LED RASVETOM PUTEM BLUETOOTH-A

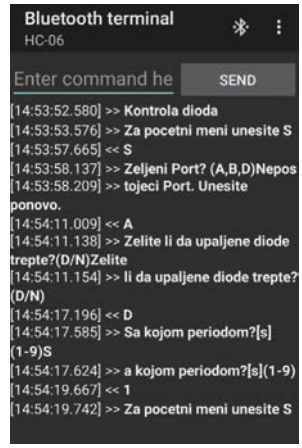
Cilj ove vežbe je napisati program koji omogućava konfiguraciju i upravljanje LED rasvetom pomoću već korišćene Bluetooth aplikacije. Komunikacija između kontrolera i mobilne aplikacije počinje sa ispisivanjem teksta „Kontrola led dioda“ zatim „Za pocetni meni unesite S“. Slanjem S karaktera, kada primi ovu poruku, kontroler postavlja pitanje “ZeljeniPort?”, a korisnik treba da unese port na kome želi da konfiguriše osvetljenje (moguće je konfigurisati portove A,B i D). Ukoliko korisnik unese slovo koje ne odgovara nekom od portova koji mogu da se konfigurišu, kontroler mu šalje poruku “Nepostojeci Port. Unesite ponovo.”. Ovo se ponavlja sve dok ne bude uneto ispravno slovo za željeni port. Po ispravnom unosu porta, kontroler postavlja korisniku pitanje “Zelite li da upaljene diode trepte?”. Sve dok korisnik ne odgovori sa “D“ ili “N“, kontroler ponavlja prethodno pitanje. Ukoliko je odgovor “D“, kontroler pita korisnika “Koja je perioda treptanja [s]?“, nakon čega korisnik unosi ceo broj sekundi koje predstavljaju periodu treptanja upaljenih dioda. Kontroler treba da proveri da li je perioda ispravno uneta. Kontroler zatim primenjuje konfigurisani svetlosni obrazac sve dok korisnik ponovo ne pošalje znak S za pocetni meni. Kako bi vršili komunikaciju preko USB UART-a potrebno je povezati USB UART konektor koji se nalazi na EasyPIC ploči sa USB portom računara. Za realizaciju ove vežbe na ploči je potrebno uključiti prekidače SW1.1., SW2.1., SW3.1., SW3.2. i SW3.4., povezati HC-06 Bluetooth modul, RX pin na RC6 i TX pin na RC7, i postaviti jumpere J3 i J4 na USB UART. Takođe je potrebno uključiti UART biblioteku.



```
1.  char uart_rd;
2.  int mode;
3.  bit flagA,flagB,flagD;
4.  int countA,countB,countD;
5.  int perA,perB,perD;
6.  char active_port;
7.  void interrupt() {
8.      if (TMR0IF_bit) {
9.          TMR0IF_bit = 0;
10.         countA++;
11.         countB++;
12.         countD++;
13.     }
14. }
15. void main() {
16.     ANSELC = 0x00;
17.     ANSELB = 0x00;
18.     ANSELD = 0x00;
19.     ANSELA = 0x00;
20.     TRISA = 0x00;
21.     TRISB = 0x00;
22.     TRISD = 0x00;
23.     UART1_Init(9600);
24.     Delay_ms(100);
25.     LATA=0x00;
26.     LATB=0x00;
27.     LATD=0x00;
28.     T0CON    = 0x86;
29.     TMR0H    = 0x0B;
30.     TMR0L    = 0xDC;
31.     GIE_bit = 1;
32.     TMR0IE_bit = 1;
33.     UART1_Write_Text("Kontrola led dioda\n");
34.     Delay_ms(250);
35.     UART1_Write_Text("Za pocetni meni unesite S\n");
36.     Delay_ms(250);
37.     mode = 0;
38.     flagA=0;
39.     flagB=0;
40.     flagD=0;
41.     countA=0;
42.     countB=0;
43.     countD=0;
44.     perA=0;
45.     perB=0;
46.     perD=0;
```

```
47.     active_port="";
48.     while (1) {
49.         if (UART1_Data_Ready()) {
50.             uart_rd=UART1_Read();
51.             switch (mode){
52.                 case 0:
53.                 //Provera user input-a
54.                     if(uart_rd=='S'){
55.                         mode = 1;
56.                         active_port="";
57.                         UART1_Write_Text("Koji port zelite?
                            (A,B,D)\n");
58.                     }
59.                 break;
60.                 case 1:
61.                     //Izbor odgovarajuceg porta A,B ili D
62.                     if (uart_rd == 'A' || uart_rd == 'B' ||
                            uart_rd == 'D'){
63.                         active_port=uart_rd;
64.                         mode = 2;
65.                         UART1_Write_Text("Diode trepte?(D/N)\n");
66.                     }else{
67.                         UART1_Write_Text("Nepostojeci Port.
                            Unesite ponovo.\n");
68.                     }
69.                 break;
70.                 case 2:
71.                     //Izbor treptanja ili toggle
72.                     if(uart_rd =='N') {
73.                         switch (active_port) {
74.                           case 'A':LATA=~LATA;flagA=0;UART1_
                              Write_Text("Za pocetni meni unesite
                              S\n");break;
75.                           case 'B':LATB=~LATB;flagB=0;UART1_
                              Write_Text("Za pocetni meni unesite
                              S\n");break;
76.                           case 'D':LATD=~LATD;flagD=0;UART1_
                              Write_Text("Za pocetni meni unesite
                              S\n");break;
77.                         }
78.                         mode=0;
79.                         break;
80.                         } else if (uart_rd == 'D'){
81.                             mode = 3;
82.                                 UART1_Write_Text("Sa kojom
                                    periodom?[s](1-9)\n");
```

```
83.                         } else{
84.                             UART1_Write_Text("Diode trepte?
                                (D/N)\n");
85.                         }
86.                 break;
87.                 case 3:
88.                     //Izbor treptanja
89.                     if (uart_rd >= '1' && uart_rd <='9'){
90.                         switch (active_port) {
91.                            case 'A':flagA=1;perA=uart_rd-48;
                               UART1_Write_Text("Za pocetni
                               meni unesite S\n");break;
92.                            case 'B':flagB=1;perB=uart_rd-48;
                               UART1_Write_Text("Za pocetni
                               meni unesite S\n");break;
93.                            case 'D':flagD=1;perD=uart_rd-48;
                               UART1_Write_Text("Za pocetni
                               meni unesite S\n");break;
94.                         }
95.                         mode=0;
96.                     }else {
97.                     UART1_Write_Text("Sa kojom periodom?[s]
                        (1-9)\n");
98.                     }
99.                 break;
100.                }
101.            }
102.            if(flagA && countA>=perA){
103.                LATA=~LATA;countA=0;
104.            }
105.            if(flagB && countB>=perB){
106.                LATB=~LATB;countB=0;
107.            }
108.            if(flagD && countD>=perD){
109.                LATD=~LATD;countD=0;
110.            }
111.        }
112.    }
```



Definicija promenljivih: Red 1-6: Definišu se promenljive koje će se koristiti u programu. uart_rd: Služi za čuvanje karaktera primljenih putem UART-a. mode: Služi za praćenje trenutnog stanja u kojem se nalazi program (konfiguracija, izbor porta, itd.). flagA, flagB, flagD: Zastavice koje označavaju da li diode na portovima A, B ili D treba da trepću. countA, countB, countD: Brojači koji broje koliko je proteklo vremena za treptanje dioda. perA, perB, perD: Periodi treptanja za diode na portovima A, B ili D. active_port: Oznaka aktivnog porta koji je trenutno izabran za upravljanje. ISR (Interrupt Service Routine): Redovi 7-13: Ovo je prekidna rutina koja se aktivira svaki put kada tajmer 0 završi odbrojavanje. Kada se prekid dogodi, brojači countA, countB, i countD se povećavaju, što služi za praćenje vremena između treptanja dioda. Inicijalizacija u funkciji main: Redovi 16-22: Konfigurišu se svi potrebni portovi (A, B, C, D) kao digitalni izlazi. Red 23: UART1_Init(9600); - Inicijalizuje UART sa brzinom od 9600 bita u sekundi za Bluetooth komunikaciju. Redovi 28-30: Inicijalizuje se tajmer 0 koji će biti korišćen za generisanje prekida svakih nekoliko milisekundi. Redovi 31-32: Omogućava se globalni prekid (GIE_ bit = 1;) i prekid za tajmer 0 (TMR0IE_bit = 1;). Osnovna petlja programa: Redovi 48-111: Beskonačna petlja koja proverava dolazne podatke putem UART-a i upravlja LED diodama na osnovu unosa korisnika. Redovi 49-101: Proverava da li je stigao podatak putem UART-a i obrađuje ga. Case 0: Kada je korisnik unese karakter ‘S’, prelazi u režim za izbor porta. Case 1: Korisnik bira port (A, B ili D) koji želi da kontroliše. Case 2: Korisnik bira da li želi da diode na odabranom portu trepću ili da samo uključi/isključi diode. Case 3: Ako korisnik izabere treptanje dioda, ovde unosi period treptanja (u sekundama). Redovi 102-110: Provera da li je protekao period za treptanje dioda i izvršava treptanje. Ukratko, ovaj program omogućava korisniku da putem Bluetooth-a izabere koji port će kontrolisati i na koji način će se diode ponašati. Korisnik može jednostavno uključiti ili isključiti sve diode na odabranom portu ili omogućiti treptanje dioda sa određenim periodom. Program koristi UART za komunikaciju i tajmer za generisanje potrebnih intervala za treptanje dioda.

LITERATURA

[1] Qing Li, Caroline Yao, Real-Time Concepts for Embedded Systems, Elsevier, 2003

[2] Jim Cooling, Real-Time Operating Systems Book 1: The Foundations, Amazon Digital Services LLC, 2018

[3] Mukul Shirvaikar, Fundamentals of Real Time Systems, Cognella Academic Publishing, 2017

[4] Hermann Kopetz, Wilfried Steiner, Real-Time Systems: Design Principles for Distributed Embedded Applications, Springer, 2022

[5] Jane W. S. Liu, Real-Time Systems, Prentice Hall, 2000

[6] Naslov: PIC mikrokontroleri, Izdavač: MikroElektronika, Strana: 396, Godina izdanja: 2007, ISBN: 978-86-84417-14-7

[7] https://www.mikroe.com/easypic-v7

[8] http://tnt.etf.bg.ac.rs/~oe4srv/

BIOGRAFIJE AUTORA



Profesor dr Marko Tanasković

Marko Tanasković rođen je 18.06.1986. godine u Valjevu, gde je završio Valjevsku gimnaziju. Završio je osnovne studije na Elektrotehničkom fakultetu Univerziteta u Beogradu, a master i doktorske studije iz oblasti automatskog upravljanja na ETH u Cirihu. Od 2017. godine zaposlen je na Univerzitetu Singidunum kao redovni profesor gde predaje predmete kao što su Internet stvari i Sistemi za rad u realnom vremenu. Marko Tanasković ima brojne objavljene radove iz oblasti automatskog upravljanja i upravljanja elektromotorima, kao i patente iz ovih oblasti.



Mast. Bojan Gutić

Bojan Gutić, rođen 5.6.1999. u Beogradu, završio je master studije smera Data Science na Univerzitetu Singidunum. Trenutno je doktorand smera Intelligent Software Engineering gde se bavi istraživanjima u oblasti veštačke inteligencije i mašinskog učenja. Angažovan je kao asistent na Univerzitetu Singidunum od 2022. godine gde aktivno drži vežbe iz više predmeta na smeru Softversko i informaciono inženjerstvo na srpskom i engleskom jeziku. Objavio je i nekoliko naučnih radova na domaćim i međunarodnim konferencijama, a učestvovao je i u domaćim i međunarodnim projektima iz više oblasti. Pored akademskog rada, honorarno radi i sarađuje sa nekoliko kompanija i klijenata. U slobodno vreme uživa u istraživanju novih tehnologija, a kao hobi voli da se bavi fotografijom i starim automobilima.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.451.031.43(075.8)

ТАНАСКОВИЋ, Марко, 1986-
Sistemi za rad u realnom vremenu / Marko Tanasković, Bojan Gutić. - 1. izd. - Beograd : Univerzitet Singidunum, 2025 (Beograd : Birograf). - V, 171 str. : ilustr. ; 24 cm

Slike autora. - Tiraž 600. - Biografije autora: str. 170-171. - Bibliografija: str. 168.

ISBN 978-86-7912-817-1

1. Гутић, Бојан, 1999- [аутор]
а) Оперативни системи -- Пројектовање

COBISS.SR-ID 163202057

© 2025.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.

9 788679 128171

www.singidunum.ac.rs

SISTEMI ZA RAD U REALNOM VREMENU

Marko Tanasković
Bojan Gutić

Real-Time Systems

Udžbenik „Sistemi za rad u realnom vremenu“ predstavlja sveobuhvatan pregled oblasti razvoja softvera za sisteme za rad u realnom vremenu. Knjiga detaljno osvetljava osnovne izazove projektovanja ovakvih sistema, baveći se ključnim konceptima kao što su obrada prekida, precizno merenje vremena, raspoređivanje programskih zadataka i problemi njihove komunikacije i sinhronizacije.

Teoretska razmatranja su propraćena nizom praktičnih primera implementiranih na MikroElektronikinoj razvojnoj platformi. Primeri su gradacijski organizovani – od najjednostavnijih koncepata do najkompleksnijih rešenja, pri čemu najzahtevniji primeri mogu da posluže kao mini projekti za proveru znanja ili kao temelj za laboratorijske vežbe. Za svaki primer dato je kompletno detaljno prokomentarisano rešenje, omogućavajući sticanje praktičnih veština kroz analizu i primenu rešenja u realnim uslovima.

Struktura i sadržaj prilagođeni su kako teorijskom usvajanju znanja, tako i praktičnom radu. Kroz kombinaciju teorije i prakse, stiče se temeljno razumevanje principa sistema za rad u realnom vremenu, što omogućava uspešno suočavanje sa izazovima modernog softverskog inženjeringa u oblasti ugrađenih sistema.

Univerzitet Singidunum