BORIS JAKOVLJEVIĆ, STEFANA JOCIĆ, MILOŠ MILETIĆ

UPRAVLJAČKI ALGORITMI I SOFTVERSKI DIZAJNIRANA INSTRUMENTACIJA REALIZOVANA POMOĆU LABVIEW-A

FAKULTET TEHNIČKIH NAUKA

PUBLISHED BY FAKULTET TEHNIČKIH NAUKA

Co-funded by the
Erasmus+ Programme
of the European Union

Sadržaj

Spisak slika

Spisak tabela

Našim porodicama...

Predgovor

Ova knjiga predstavlja sveobuhvat višegodišnjeg rada autora na poljima realizacije upravljačkih sistema u LabView-u. Knjiga je struktirana tako da prati tematiku predmeta Upravljački algoritmi u realnom vremenu na kursu u okviru osnovnih studija Računarstva i automatike na Fakultetu tehničkih nauka u Novom Sadu. Autori su želeli da vrlo intuitivno kroz detaljno objašnjene primere ilustruju implementaciju različitih upravljačkih struktura, kao i dizajniranje softverski dizajniranih instrumenata kroz jednostavane i napredne tehnike programiranja u *LabView*-u. Na početku su date osnove samog programskog jezika *LabView*. Opisani su elementi poput petlji, nizova, klastera, te specifičnih LabView komponenti poput *shift* registara, *flat* sekvenci. Naredna celina koja je obrađena je rad sa fajlovima i neke osnovne operacije sa njima koje su integrisane u složeniji projekat. Potom je jedna velika celina posvećena automatima stanja i njihovim realizacijima kroz različite šeme projektovanja koje imaju drugačiju ulogu u zavsnosti od projektnog zadatka. U nastavku je obrađena jedna implementacija najzastuplljenijeg industrijskog upravljačkog algoritma, PID algoritma u digitalnoj formi. Na posletku su detaljno obajašnjeni primeri realizacije projekata sa višeslojnim arhitektutama i različitim platformama.

Uvod u LabVIEW

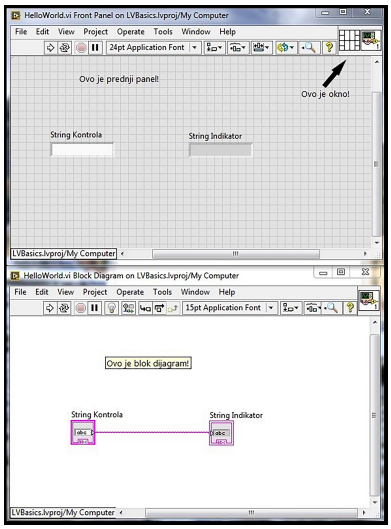
LabVIEW (*Laboratory Virtual Instrumentation Engineering Workbench*) predstavlja razvojno okruženje za grafičko programiranje. To je potpun programski jezik koji koriste inženjeri, naučnici i studenti za razvoj različitih aplikacija za akviziciju, merenje, upravljanje i testiranje [1]. *LabVIEW* ima sve standardne elemente tipčnog programskog jezika: različite tipove podataka, strukture podataka, nizove (*arrays*), petlje (*loops*), uslovne izjave (*conditional statements*), funckije za rad sa stringovima, fajlovima, itd. Tehnički govoreći, *LabVIEW* je ime okruženja, dok je "G" naziv grafičkog programskog jezika koji se koristi u tom okruženju. Međutim, "G" kod se jedino može razvijati u okviru *LabVIEW* okruženja, pa je stoga opšte prihvaćeno da se termin *LabVIEW* odnosi na oba, i razvojno okruženje i programski jezik. *LabVIEW* kod se komplajlira u pozadini (direktno u mašinski kod), odnosno nema potrebe za kompajiranjem kao posebnim korakom. *LabVIEW* se može integrisati sa hiljadama hardverskih uređaja i dolazi sa stotinama ugrađenih biblioteka za analizu i vizuelizaciju podataka. Iako ima mnogo zajedničkih stvari sa tipičnim programskim jezikom, postoje dva aspekta zbog kojih se *LabVIEW* razlikuje od ostalih programskih jezika: grafičko programiranje i tok kretanja podataka (*dataflow*). U ovom poglavlju biće objašnjene osnove programiranja u *LabVIEW* programskom jeziku.

[1] Jeffrey Travis; Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall, New Jersey, USA, third edition, 2006

Anatomija jednog virtuelnog instrumenta

Očigledna razlika u odnosu na tekstualne programske jezike jeste grafička priroda *LabVIEW*-a. *LabVIEW* program naziva se virtuelni instrument (*virtual instrument*), skraćeno VI, jer svojim izgledom treba da podseća na fizičke instrumente za merenje kao što su npr. osciloskop, multimetar, itd. Prilikom kreiranja novog VI-a kreiraju se tri osnovna elementa svakog virtuelnog instrumenta: prednji panel (*front panel*), blok dijagram (*block diagram*) i okno (*icon/connector pane*). Prednji panel i blok dijagram se kreiraju kao dva odvojena prozora. Prednji panel ujedno predstavlja i korisnički interfejs koji vidi korisnik i obično sadrži dugmiće (*buttons*), brojčanike (*dials*), grafike, tekstualna polja, itd.

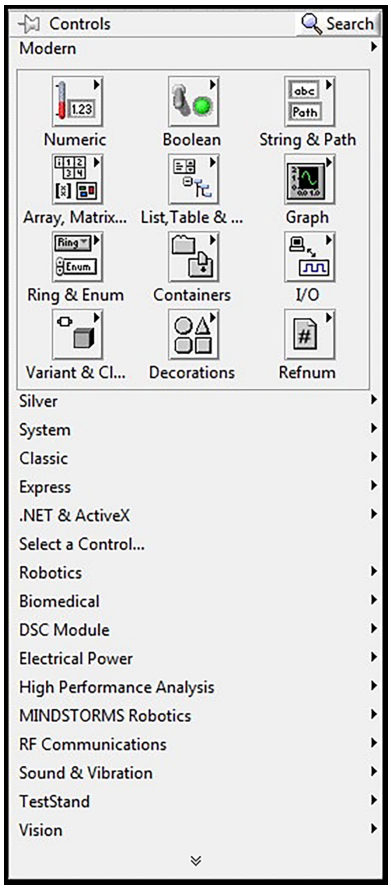
Blok dijagram sadrži programski kod i njega vidi samo programer. U gornjem desnom uglu prednjeg panela nalazi se okno. Okno se sastoji od ikonice, slike koju je kreirao programer i ona predstavlja kako će izgledati VI ukoliko se koristi kao pod VI u nekom većem programu. Okno takođe sadrži i šemu ulaza i izlaza virtuelnog instrumenta. Primer jednog virtuelnog instrumenta se može videti na slici 1 (gore se nalazi prednji panel, dok je dole prikazan blok dijagram).



Slika 1: Ovo je primer jednog virtuelnog instrumenta. U gornjem delu slike nalazi se prednji panel, dok je u donjem delu slike blok dijagram. Na prednjem panelu strelicom je označeno mesto gde se nalazi okno. Okno se sastoji od ikonice (desno) i šeme povezivanja ulaza i izlaza (levo).

Kontrole, indikatori i žice

Prednji panel i blok dijagram povezani su kontrolama i indikatorima. Kontrole se nalaze na prednjem panelu i njih kontroliše korisnik, odnosno preko njih unosi informacije u VI, dok indikatori služe da prikažu informacije korisniku. Kontrole i indikatori postavljaju se na prednji panel prevlačenjem iz palete (čiji se izgled može videti na slici 2), a kojoj se može pristupiti desnim klikom miša na prazan prostor na panelu ili odabrati iz padajućeg menija *View* → *Controls Palette*. Svaki put kada se sa palete prevuče neki element na prednji panel, odgovarajući element se pojavi i na blok dijagramu. Pomoću kursora, kontrole i indikatori se povezuju predstavljajući tako tok kretanja podataka kroz program. Podaci putuju od kontrola do indikatora. Primeri pojedinih kontrola i indikatora se mogu videti na slici 3. Kontrole i indikatori na blok dijagramu se mogu prikazati ili kao ikonice ili kao terminali. Ikonice su detaljnije i veće, dok su terminali manji, pa samim tim zauzimaju manje mesta na ekranu. Podrazumevani prikaz kontrola i indikatora su ikonice. Uporedni prikaz ikonica i terminala na blok dijagramu se može videti na slici 4. Bez obzira na to da li se koristi izgled ikonica ili indikatora, svaka kontrola i svaki indikator moraju imati jedinstveno ime u programu. Naziv elementa je vidljiv i na blok dijagramu i na prednjem panelu, ali se može i ukloniti. Slično kao i na prednjem panelu, i na blok dijagramu postoji paleta iz koje je moguće prevući razne funkcije za pisanje koda. Izgled palete sa blok dijagrama i njen delimični sadržaj može se videti na slici 5.

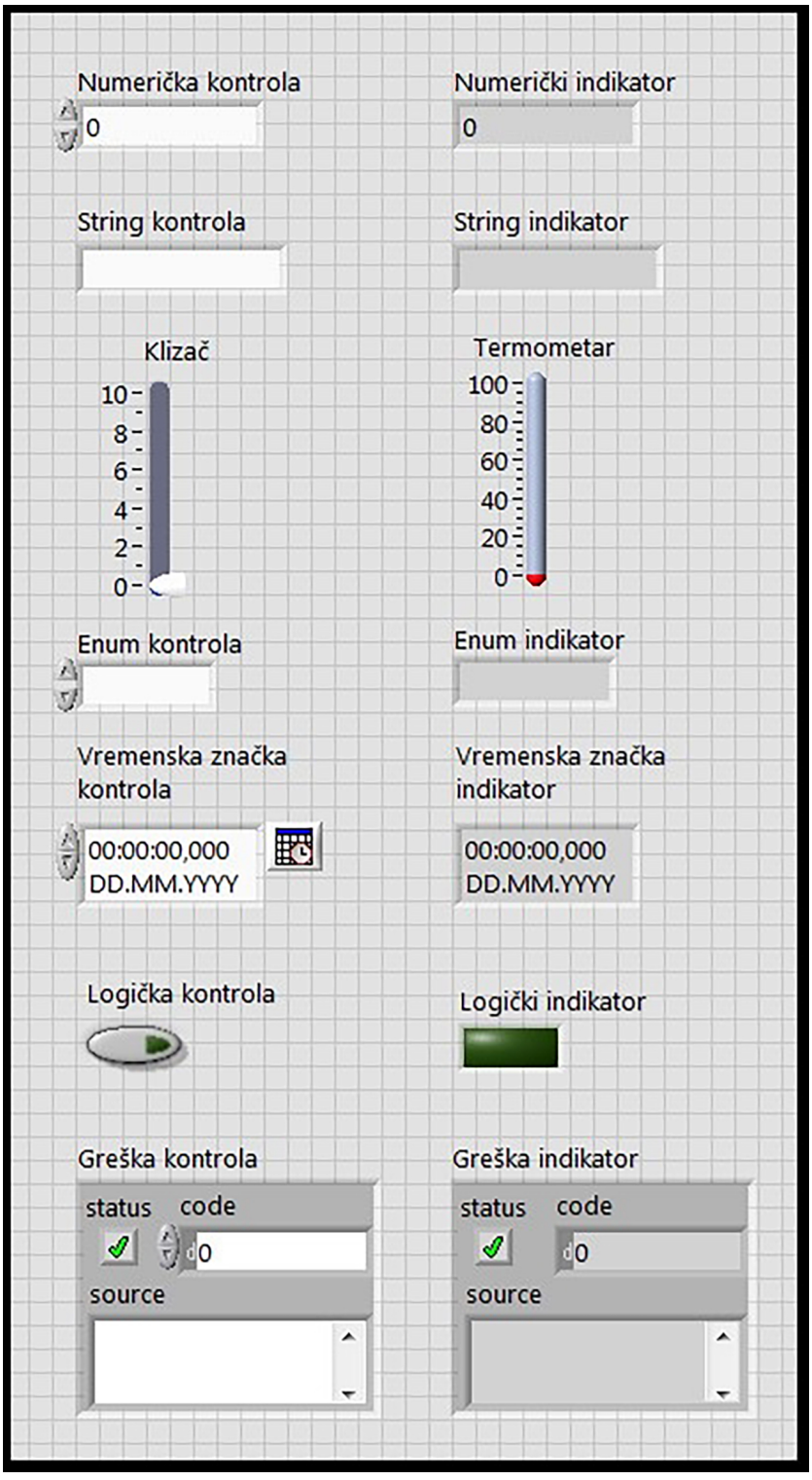


Slika 2: Paleta sa prednjeg panela sadrži kontrole i indikatore, koji su podeljeni u kategorije kao što se može videti na slici.

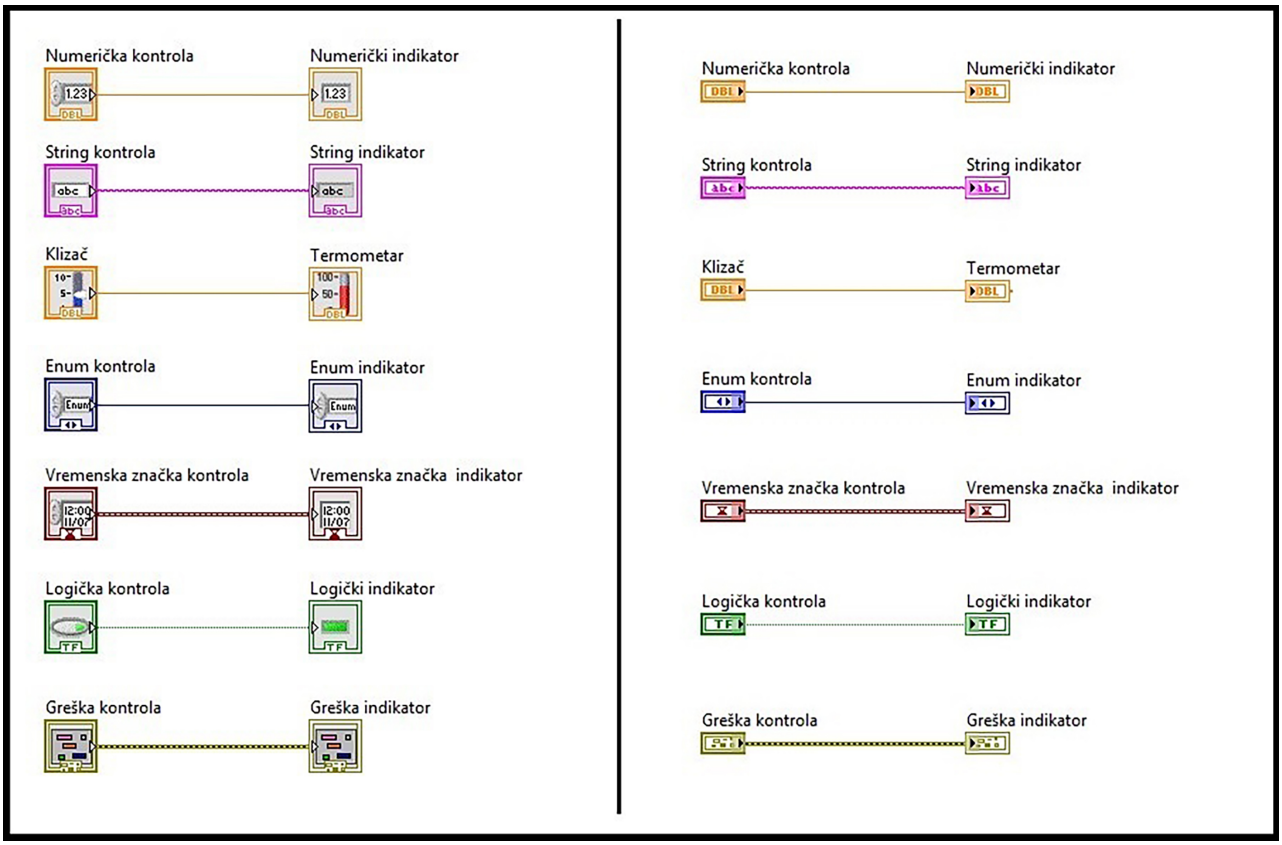
Tok kretanja podataka kroz program

Tok kretanja podataka određuje redosled izvršavanja programa. Kod većine programskih jezika, kod se izvršava sekvencijalno, odnosno kada se jedna linija izvrši ide sledeća, i tako redom. Međutim, u *LabVIEW*-u podaci se kreću duž žica u paraleli i osnovno pravilo jeste da se bilo koja funkcija u programu može izvršiti onda kada ima sve potrebne ulazne podatke. VI se izvršava sa leve na desnu stranu, ali to je samo konvencija.

Posmatrajamo jednostavan program sa slike 1. Kada se VI pokre-



Slika 3: Podrazumevani prikaz pojedinih kontrola i indikatora na prednjem panelu. Kontrole se nalaze sa leve strane, dok su indikatori smešteni sa desne strane.



Slika 4: Uporedni prikaz ikonica i terminala. Na levoj polovini slike nalaze se kontrole i indikatori prikazani kao ikonice, dok su na desnoj polovini prikazani kao terminali.

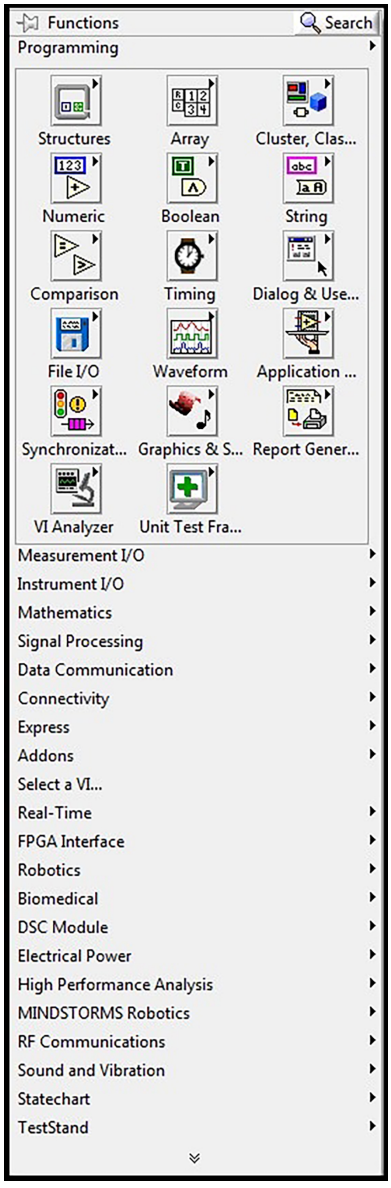
ne kontrola pod nazivom *String kontrola* se može izvršiti pošto nema ulaza, dok indikator nazvan *String indikator* se ne može odmah pri pokretanju izvršiti, jer ulazni podaci potrebni za njegovo izvršavanje nisu dostupni. Podaci iz kontrole putuju duž žice do indikatora. Kada indikator primi podatke, on se istog trenutka izvršava i prikazuje tekst koji je primio od kontrole.

Tipovi podataka u LabVIEW-u

Različiti tipovi podataka u *LabVIEW*-u predstavljeni su različitim bojama žica, tako narandžasta boja predstavlja tip sa pokretnim zarezom (*floating-point*), plava celobrojni tip podatka (*integer*), logički tip (*boolean*) prikazuje se zelenom bojom, dok je tekstualni tip (*string*) roze boje. Kontrole i indikatori se mogu povezati samo ako su istog tipa. Ukoliko se ne poklapaju žica će se prekinuti i VI nije moguće pokrenuti, što je predstavljeno prekinutom strelicom za pokretanje programa. Neki od tipova podataka, kao što je numerički tip, moguće je prikazati na više načina. Numerički tip podatka sa pokretnim zarezom može se

prikazati u proširenoj preciznosti, sa dve decimalne vrednosti i jednom decimalnom vrednosti. Celobrojni tip podatka može biti označen (*signed*) ili neoznačen (*unsigned*) i predstvaljen kao 8, 16, 32 i 64 bita. Žice i terminale različitih reprezentacija istog tipa podataka moguće je povezati bez pucanja žice, tako što će sam *LabVIEW* u pozadini izvršiti automatsku konverziju podataka. Ova automatska konverzija prikazana je crvenom tačkicom na blok dijagramu, što je ilustrovano na slici 6. Ipak, savetuje se da bitne odluke oko konverzije podataka bolje je prepustiti programeru, nego *LabVIEW*-u, jer u suprotnom može doći do neočekivanih rezltata.

Kao što numerički tip podatka može imati različite reprezentacije, tako i logički tip ima posebnu karakteristiku pod nazivom mehaničke akcije (*mechanical actions*). U fizičkom svetu, logičke kontrole odgovaraju prekidačima i tasterima, gde ova dva imaju različite mehaničke akcije. Kada uključimo ili isključimo prekidač za svetlo, svetlo se istog trenutka uključuje ili isključuje i prekidač ostaje u zadatom položaju. Ova radnja odgovara mehaničkoj akciji u *LabVIEW*-u pod nazivom "prebaciti kada se pritisne" (*"switch when pressed"*). Zvono na vratima će zvoniti onoliko dugo koliko je prekidač pritisnut, što odgovara mehaničkoj akciji nazvanoj "prebaciti dok se ne otpusti" (*"switch until released"*). U operativnom sistemu *Windows* dok držimo levi klik miša iznad znaka "X" na prozoru ne dešava se ništa, sve dok ne otpustimo klik na mišu i prozor će se zatvoriti. Ovakava radnja odgovara mehaničkoj akciji "prebaciti kada se otpusti" (*"switch when released"*). Sve ove akcije koje omogućavaju istovremeno prebacivanje (*switching actions*) moguće je simulirati u *LabVIEW*-u, ali takođe je moguće i simulrati iste ove akcije koje imaju kašnjenje (*latching actions*).



Slika 5: Izgled palete sa blok dijagrama. Sadrži različite funkcije za rad sa numeričkim i logičkim tipom podataka, klasterima, nizovima, itd.

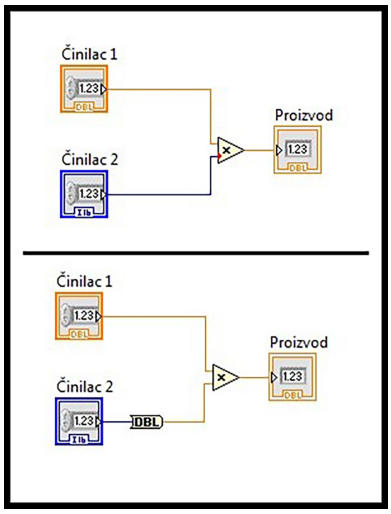
Nizovi i klasteri

Kao što smo spomenuli na početku poglavlja *LabVIEW* ima sve standardne elemente tipčnog programskog jezika, što uključuje i nizove. Niz predstavlja uređenu grupu elemenata koji su istog tipa. Elementi niza moraju biti sve samo kontrole, samo konstante, samo indikatori, pa čak i složenije strukture kao što su klasteri, o kojima će biti opisani u nastavku. Svakom elementu je moguće pristupiti pojedinačno pomoću njegovog indeksa, što predstavlja celi broj koji definiše mesto tog elementa u nizu. Indeksi u LabVIEW-u kreću od 0, pa tako na primer ukoliko niz ima 10 elemenata njihovi indeksi idu od 0 do 9. Izgled niza u obliku ikonica i terminala prikazani su na slici 7.

Klasteri predstavljaju grupu podataka koji nisu svi istog tipa, ali takođe moraju sadržati samo kontrole ili samo indikatore. Za razliku od nizova, gde broj elemenata niza nije fiksan, kod klastera se broj elemenata definiše onda kada je klaster kreiran. Jedan od često kori-

šćenih klastera jeste klaster greške (*Error Cluster*) prikazan na slici 8. Ovaj klaster sastoji se od tri elementa: logičke kontrole koja govori da li je greška prisutna, 32-bitnog celobrojnog podatka koji predstavlja kod greške i tekstualni tip koji sadrži izvor greške.

Da bi se pročitao element iz klastera, on prvo mora biti izdvojen iz njega. Postoje dve funkcije u *LabVIEW*-u koje omogućavaju izdvajanje elemenata iz klastera koje su prikazane na slici 9. Prva funkcija za izdvajanje elemenata (*Unbundle*) omogućava izdvajanje elemenata po tipu podatka, bez naziva promenljive, što može biti nezgodno ukoliko u klasteru postoji više istih tipova podataka. Iz tog razloga, bolje je koristiti funkciju koja razdvaja elemente klastera po imenu (*Unbundle By Name*). Korišćenje ove funkcije olakšava čitanje koda i takođe, ona omogućava da prikažemo samo one elemente iz klastera koji su potrebni (treći slučaj na slici 9). Osim čitanja elemenata iz klastera, moguće je i upisati podatke u klaster. Za ovu radnju takođe postoje dve funkcije, jedna koja grupiše sve podatke u klaster odjednom (*Bundle*) i druga koja ih grupiše po imenu (*Bundle By Name*).



Slika 6: Na gornjoj polovini slike nalazi se crvena tačkica na ulazu funkcije za množenje (*Multiply function*) koja predstavlja automatsku konverziju celobrojnog tipa u tip sa pokretnim zarezom sa dve decimalne vrednosti. Na donjoj polovini slike vidi se da je automatska konverzija izbegnuta postavljanjem funkcije za konverziju broja u tip sa pokretnim zarezom sa dve decimalne vrednosti (*To Double Precision Float*).

Enumeracija

Kao i kod drugih programskih jezika i u *LabVIEW*-u postoji enumeracija. O enumerisanom tipu podatka može se misliti kao o setu od n stavki od kojih je svaka povezana za jednu vrednost celog broja od 0 do $n-1$. Enumeracija je korisna kada se radi poboljšanju čitljivosti koda, jer korisnik na ekranu vidi tekst, a u pozadini su te tekstualne vrednosti vezane za brojeve. Na primer, prodajemo odeću i imamo 3 moguće veličine - mala (S), srednja (M) i velika (L), gde je S označeno nulom, M jedinicom i L dvojkom. Tako će korisnik na ekranu vidi tekstualni zapis svake veličine, dok će se u kodu koristiti celobrojna vrednost. Ovaj tip podataka koristiće se često u narednim poglavljima.

Case struktura

U tekstualnim programskim jezicima kada želimo da proverimo da li neki od uslova (*condition*) ispunjen, koristimo *IF* naredbu. S obzirom da u *LabVIEW*-u ne postoji *IF* naredba, ovde ćemo za isti problem koristiti strukturu sa slučajevima (*case structure*).

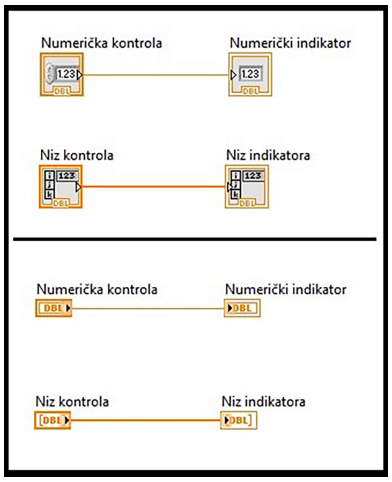
Struktura sa slučajevima predstavljena je u pravougaonastom okviru, na kojem se sa leve strane nalazi terminal izbora (*selector terminal*) koji liči na mali pravougaonik koji sadrži znak upita. Ova struktura se može zamisliti kao niz povezanih ramova i svaki od ramova odgovara jednoj od vrednosti koja je povezana na terminal izbora. U zavisnosti od toga koji je tip podatka povezan na terminal, postoji različit broj slučajeva. Tako na primer, ukoliko je na terminal povezan logički tip

podatka, moguća su dva slučaja - *Tačno* (*true*) i *Netačno* (*false*). Kod koji se nalazi u okviru koji je označan sa *Tačno* se izvršava onda kada je vrednost koja se nalazi na terminalu izbora tačna. Slično važi i za slučaj *Netačno*.

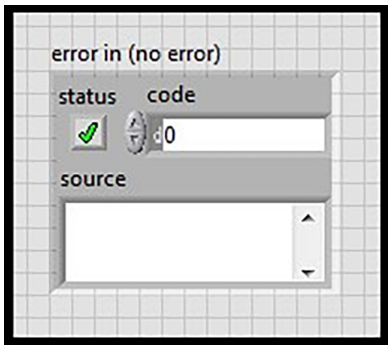
Posmatrajmo jednostavan primer sa slike 10 koji simulira uključivanje i isključivanje svetla pritiskom prekidača. Prekidač, u ovom slučaju logička kontrola, je povezan na terminal izbora strukture sa slučajevima. Kada prekidač nije pritisnut, izvršava se slučaj pod nazivom *False*, što znači da svetlo nije uključeno (gornja polovina slike). Kod je u ovom slučaju veoma jednostavan i predstavlja upisivanje logičke konstante koja ima vrednost netačno (*false*) u indikator nazvan *Svetlo*. Ukoliko korisnik pritisne logičku kontorlu nazvanu *Prekidač*, izvršiće se kod u slučaju nazvanom *True* i svetlo će se uključiti (donja polovina slike). Slično kao u prethodnom slučaju, i ovde kod predstavlja samo upisivanje logičke konstante koja ima vrednost tačno (*true*) u indikator nazvan *Svetlo*.

Osim logičkog tipa podatka, na terminal izbora može biti povezan i tekstualni tip, celobrojni tip, enumerisani tip (*enumerated control*) i klaster greške (*error cluster*). Kao što smo videli, kada je logički tip podatka bio povezan na terminal izbora postojala su dva moguća slučaja. Slično važi i ako su na terminal povezani enumerisani tip i klaster greške, jer u oba slučaju postoji tačno definisan broj vrednosti. Međutim, u slučaju kada je na terminal povezan tekstualni ili celobrojni tip podatka, tada postoji veliki broj vrednosti ograničen veličinom tipa podatka. Ukoliko se koristi neki od ova dva tipa, potrebno je definisati podrazumevani slučaj u strukturi, koji će se izvršavati onda kada vrednost na terminalu ne odgovara nijednoj vrednosti slučajeva.

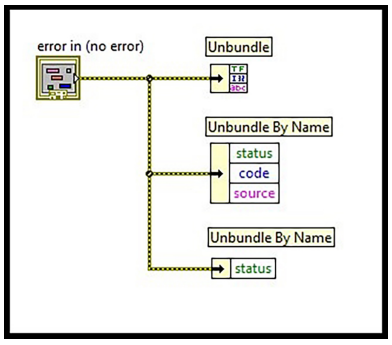
Slika 11 prikazuje izglede tunela ulaznih i izlaznih signala u različitim situacijama. Naime, ulazi u strukturu slučajeva ne moraju biti korišćeni u svakom slučaju, njihovi tuneli će uvek izgledati popunjeno (tunel na levoj strani strukture slučaja) bez obzira da li su iskorišćeni u svakom slučaju. Međutim, to ne važi kod izlaza iz strukture slučajeva. Kako bi se VI mogao pokrenuti potrebno je da vrednost bude dodeljena svakom tunelu u svakom slučaju (prvi tunel sa desne strane strukture slučaja). Ukoliko u bar jednom od slučajeva nije dodeljena vrednost tunelu (drugi tunel sa desne strane strukture slučaja), VI nije moguće pokrenuti. Taj problem je moguće rešiti desnim klikom na tunel i izborom opcije *Use Default If Unwired* i u tom slučaju će podrazumevana vrednost za određeni tip podataka biti prosleđena na izlaz, a tunel dobija izgled kao treći tunel sa desne strane strukture slučaja na slici 11.



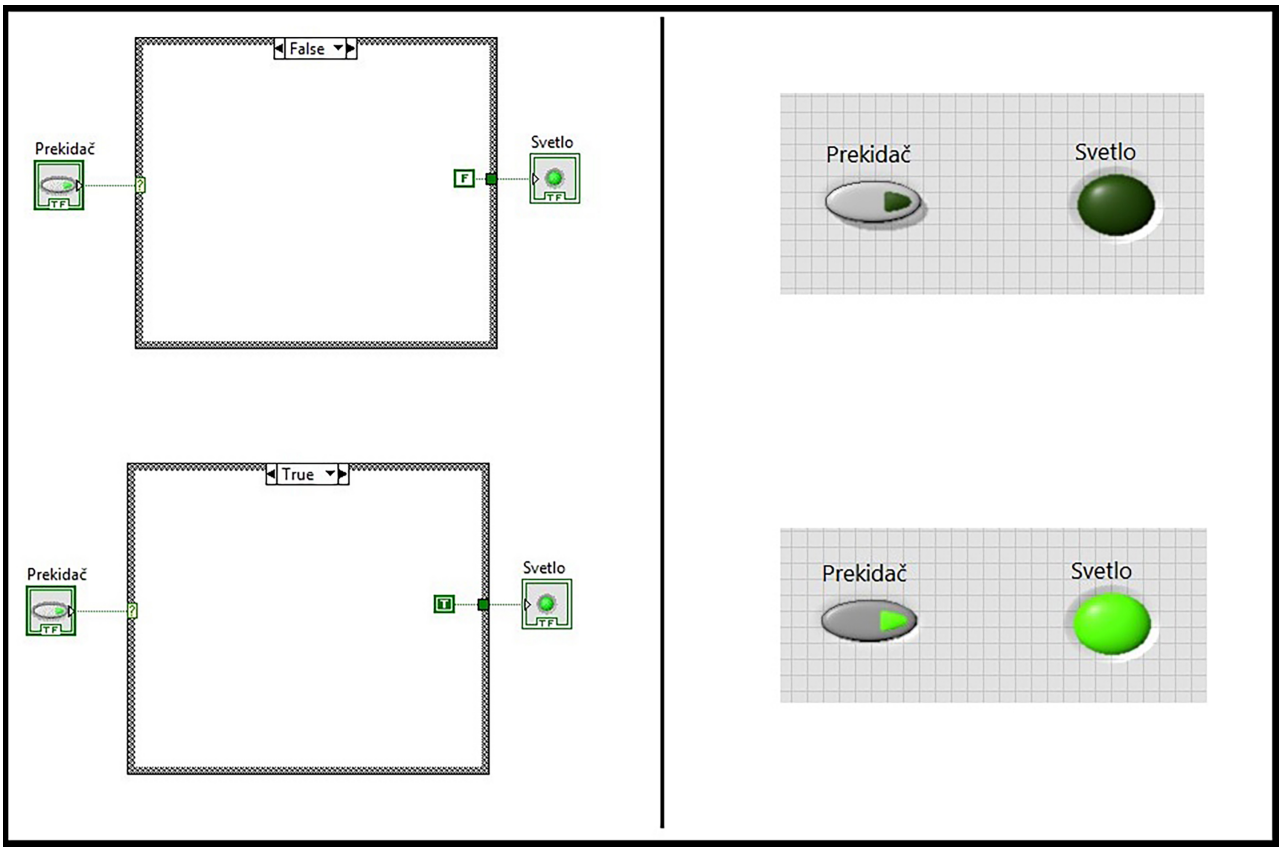
Slika 7: Na slici se jasno uočava razlika između niza i skalara, žica koja spaja Niz kontrola i Niz indikatora je deblja u odnosu na žicu koja spaja *Numeričku kontrolu* i *Numerički indikator*. Takođe, ikonica niza se razlikuje od ikonice skalara iako su podaci istog tipa.



Slika 8: Izgled klastera greške na prednjem panelu.



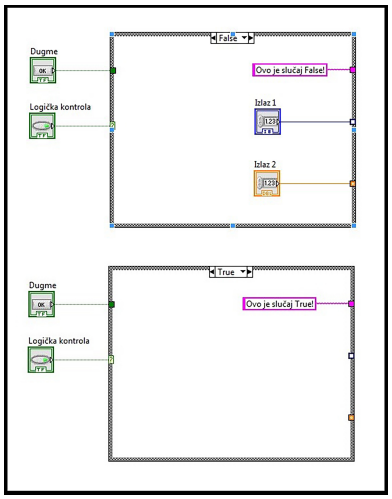
Slika 9: Funkcije za izdvajanje elemenata iz klastera.



Slika 10: Izgled blok dijagrama i prednjeg panela kada je prekidač isključen se nalazi na gornjoj polovini slike, dok donja polovina slike prikazuje izgled blok dijagrama i prednjeg panela u slučaju kada je prekidač uključen.

Petlje

Slično kao u drugim programskim jezicima, *LabVIEW* obezbeđuje dve strukture za višestruko izvršavanje delova koda: *While* petlje i *For* petlje. Njihove strukture su slične, ali glavna razlika je u tome kako petlje prestaju sa izvršavanjem koda. Tipično, *While* petlje se koriste kada programer ne zna unapred koliko puta će se petlja izvoditi, kao npr. kada pretlja treba da se izvršava sve dok korisnik ne odluči da je prekine ili kada se javi greška u programu. Za razliku od *While* petlje, *For* petlja se koristi kada je broj ponavljanja poznat unapred.



Slika 11: Različiti izgledi tunela izlaznih signala.

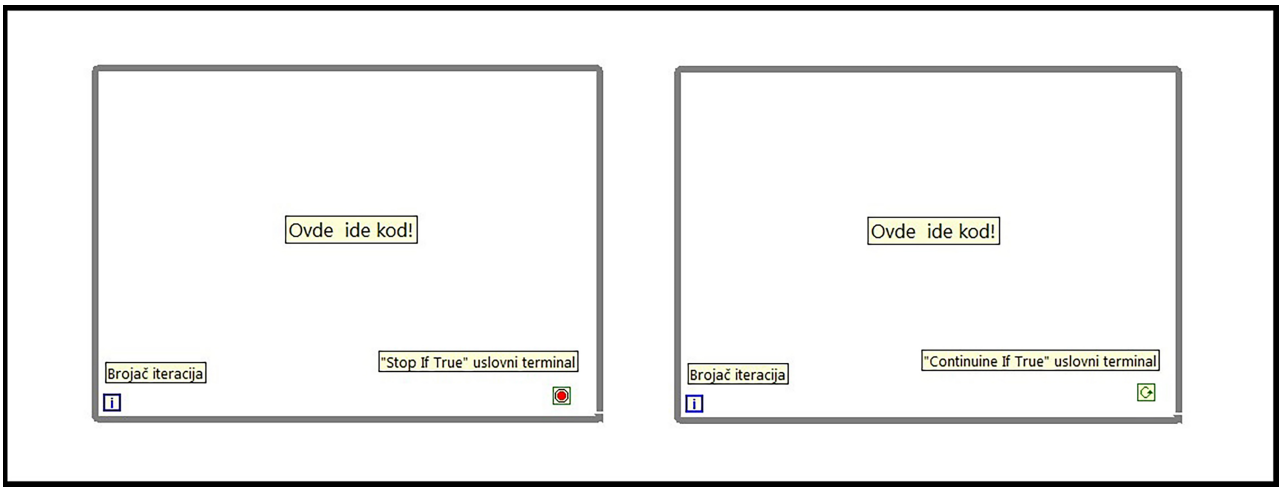
While petlja

Primer *While* petlje je prikazan na slici 12. Petlja je predstavljena sivom strelicom u obliku pravougaonika i kod koji se nalazi unutar tog pravougaonika se ponavlja sve dok je dati uslov zadovoljen. Svaka *While* petlja ima dva terminala: brojač iteracija (*iteration counter*) i uslovni terminal (*conditional terminal*). Brojač iteracija deluje kao kontrola i daje informaciju o trenutnoj iteraciji petlje kao označeni 32-bitni celobrojni podatak. Brojanje iteracija počinje od 0, tako da u prvoj iteraciji brojač iteracija ima vrednost 0. Brojač iteracija nalazi se u levom donjem ćošku petlje, što se može videti na slici 12.

Uslovni terminal ima ulogu u određivanju kada se petlja završava. Ovaj terminal nalazi se u desnom donjem ćošku petlje (takođe slika 12). Podrazumevani oblik uslovnog terminala je "zaustavi ako je tačno" (*Stop if True*) (leva petlja na slici 12), što znači da terminal deluje kao indikator i prihvata logički ulaz. Vrednost logičkog ulaza se proverava na kraju svake iteracije petlje. Ukoliko je vrednost tačna, petlja prestaje sa izvršavanjem, u suprotnom se i dalje izvršava. Uslovni terminal može biti promenjen u oblik "nastaviti ukoliko je tačno" (*Continuine if True*) klikom na terminal. U ovom slučaju, petlja će se nastaviti ako je logički ulaz tačan i završiće se ako je logički ulaz netačan.

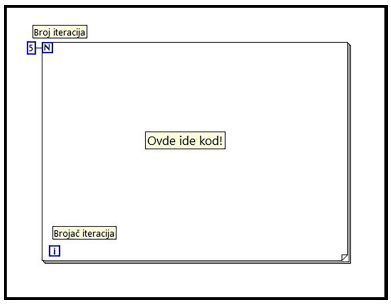
For petlja

Primer *For* petlje je prikazan na slici 13. *For* petlja izgleda kao gomila papira čija je prva strana savijena u donjem desnom ćošku. Svaka petlja ima dva terminala: brojač iteracije i terminal za broj iteracija. Brojač iteracija je identičan onom koji se nalazi u *While* petlji i nalazi se u donjem levom ćošku petlje. Brojanje iteracija takođe počinje od 0. Terminal za ukupan broj iteracija ponaša se kao indikator i nalazi se u gornjem levom ćošku petlje. Broj, 32-bitni celobrojni tip podatka, povezan na terminal brojača određuje koliko će puta petlja biti izvršena.

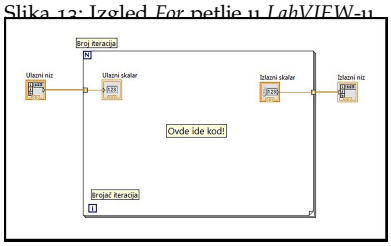


Slika 12: Izgled *While* petlje u *LabVIEW*-u.

For petlje se najčešće koriste za rad sa nizovima. Ukoliko nema drugih ulaza povezanih na *For* petlju, sem niza elemenata, ta petlja će se izvršiti onoliko puta koliko ima elemenata u nizu, čak iako ne postoji konstanta povezana na terminal za ukupan broj iteracija. Odnosno za svaki element niza biće izvršena po jedna iteracija petlje. Zahvaljujući autoindeksiranju (*auto-indexing*) sa svakom iteracijom odgovarajući element je dostupan u petlji. To znači da u prvoj iteraciji je dostupan prvi element niza, u drugoj iteraciji drugi element, itd. Na slici 14 može se videti primena autoindeksiranja u radu sa nizovima. Može se uočiti da je žica iz ulaznog niza deblja u odnosu na žicu koja prelazi u *For* petlju iz istog niza. To ukazuje da je u petlji dostupan samo jedan element niza. Autoindeksiranje se može koristiti i na izlazu iz *For* petlje ukoliko želimo da kreiramo niz, što se takođe može videti na slici 14.



Slika 13: Izgled *For* petlje u *LabVIEW*-u



Slika 14: Korišćenje *For* petlje i autoindeksiranja za rad sa nizovima.

Autoindeksiranje se prepoznaje po narandžastim uglastim zagradama "[]" u ulaznim i izlaznim tunelima signala i moguće ga je uključiti ili isključiti, biranjem iz menija dobijenim desnim klikom na željeni tunel. Ukoliko je indeksiranje isključeno, tunel ulaznog niza izgleda popunjeno i ceo niz biće dostupan u iteraciji.

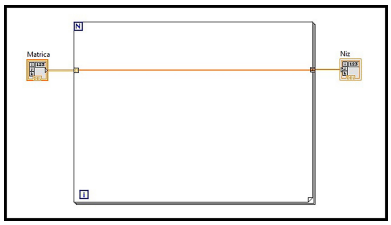
Uslovni terminal koji se javlja kod *While* petlje, može biti omogućen i u *For* petlji. U slučaju da je omogućen, petlja će se izvršavati normalno, ali ukoliko je uslov povezan na uslovni terminal ispunjen, petlja će biti zaustavljena. Tako se npr. uslovni terminal dodaje u *For* petlju kako bi se zaustavila ukoliko se javi greška u programu.

Izlazni režimi iz petlji

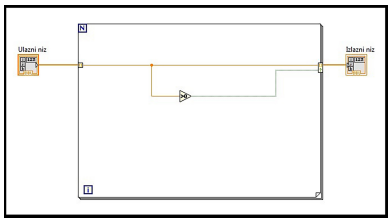
Izlazni tuneli iz petlji mogu imati četiri različita režima: "autoindeksiranje" (*"Indexing"*), "poslednja vrednost" (*"Last Value"*), "spajanje" (*"Concatenating"*) i "uslovno" (*"Conditional"*). *While* petlja kao podrazumevanu vrednost na izlaznom tunelu ima poslednju vrednost, što znači da će vrednost iz poslednje iteracije biti dostupna na izlazu. Kod *For* petlje podrazumevani režim je autoindeksiranje, koje je iznad opisano.

Izlazni režim spajanje može se koristiti u sitacijama kada je od dvodimenzionalnog niza potrebno napraviti niz jedne dimenzije. Blok dijagram programa koji od matrice kreira niz prikazan je na slici 15. U svakoj iteraciji petlje jedan red je dostupan u petlji i on se prosleđuje na izlaz petlje kako bi se upisao u niz.

Kod "uslovnog" izlaznog režima tunel ima dva ulaza - *indeksiranje* (*indexing*) i *logička selekcija* (*boolean selection*). Izlazni tunel ponaša se kao tunel u režimu indeksiranja, ali prikuplja samo one elemente niza koji ispunjavaju logički uslov povezan na izlazni tunel. Primer koda može se videti na slici 16, gde se iz ulaznog niza u izlazni upisuju samo oni brojevi koji su veći od nule.



Slika 15: Blok dijagram u slučaju korišćenja izlaznog režima spajanje kako bi se od matrice dobio niz.



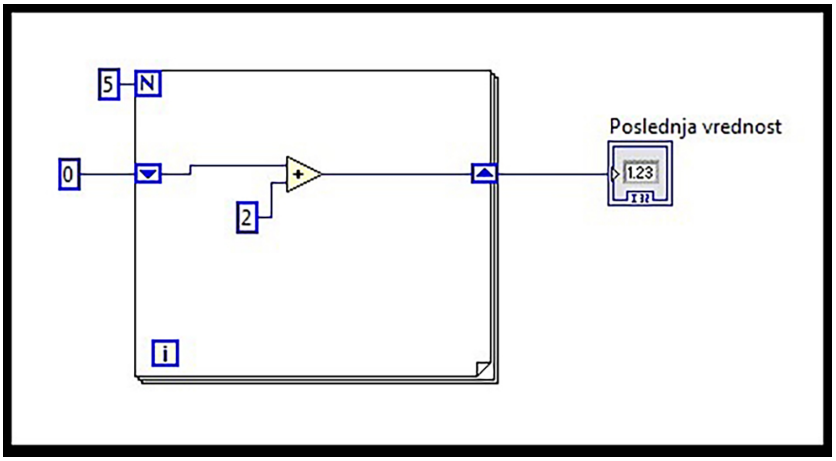
Slika 16: Blok dijagram u slučaju korišćenja uslovnog izlaznog režima.

Shift registri i Feedback čvorovi

Čuvanje podataka u tekstualnim programskim jezicima je normalna pojava, jer se vrednosti dodeljuju promenljivama i mogu se pozvati u bilo kom trenutku. Međutim, u *LabVIEW*-u podaci putuju duž žica, neimenovani, tj. nisu dodeljivani promenljivama i ne čuvaju se nigde. Iz tog razloga ukoliko je potrebno sačuvati neke podatke zarad korišćenja u budućnosti potrebno je implementirati mehanizme za skladištenje podataka.

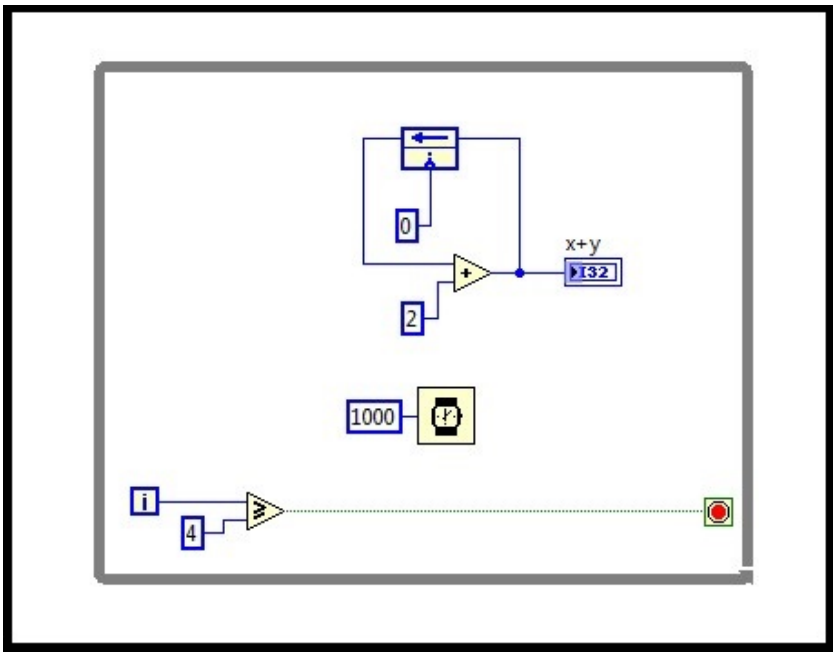
Petlje u *LabVIEW*-u, *While* i *For*, koriste *shift* registre za skladištenje podataka, koji se mogu videti na slici 17. *Shift* registar se sastoji od para terminala koji se nalaze sa leve i desne strane petlje. Terminal sa leve strane je u obliku strelice koja pokazuje na dole, dok je terminal sa desne strane strelica koja pokazuje na gore. Ovo vizuelno nagoveštava da podaci putuju "oko" petlje, od desnog *shift* registra prema levom. Podatak koji je upisan u terminal sa desne strane u jednoj iteraciji, putuje oko petlje i dostupan je na levom terminalu u sledećoj iteraciji, što predstavlja kratkotrajno čuvanje podataka. Terminal sa leve strane se može razvući i tom slučaju može čuvati više od jedne prethodnih vrednosti. Takođe, *shift* registar može biti inicijalizovan sa leve strane van petlje i vrednost koja mu je dodeljena biće vrednost koja je dostupna u petlji u prvoj iteraciji. U slučaju da *shift* registar nije inicijalizovan, u prvoj iteraciji biće dostupna podrazumevana vrednost za taj tip po-

datka.



Slika 17: Primer korišćenja *shift* registara. Na početku vrednost *shift* registar postavljena je na 0. U prvoj iteraciji na vrednost 0 se dodaje 2 i ukupan zbir je 2, koji se upisuje u desni *shift* registar. Zatim, u drugoj iteaciji na levom *shift* registru dostupna je vrednost zbira iz prethodne iteracije (što je 2), pa se na tu vrednost ponovo dodaje 2 i tako sve do kraja izvršavanja *For* petlje. Na kraju posle pet iteracija u indikator *Poslednja vrednsnost* upisuje se broj 10.

Na sličan način funcionišu i *feedback* čvorovi (*nodes*). Primer koda može se videti na slici 18.



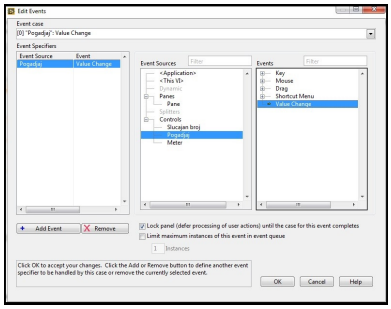
Slika 18: Primer korišćenja *feedback* čvora. Čvor je moguće inicijalizovati, i vrednost koja je povezana na to mesto koristiće se kao početna vrednost. U suprotnom se koriste podrazumevane vrednosti za povezani tip podatka. U indikator $x + y$ se na svaku sekundu upisuje trenutna vrednost, a petlja prestaje sa izvršavanjem posle pet iteracija.

Event i flat sequence struktura

Event struktura čeka da se desi određeni događaj i onda izvršava određeni deo koda koji se bavi tim događajem. Ova struktura se sastoji

od jednog ili više dijagrama, od kojih svaki se izvršava kada se desi određeni događaj. Na narednim slikama prikazano je rešenje zadatka koji omogućava korisniku da pogodi slučajno izabran broj od strane računara. Slučajno izabran broj bira se u opsegu od 0 do 100. U slučaju da korisnik unese manji broj od izabranog treba da se ispiše poruka na ekran da se izabere veći broj i da se omogući nastavak procesa pogađanja broja. U slučaju da korisnik unese veći broj od izabranog treba da se ispiše poruka na ekran da se izabere manji broj i da se omogući nastavak procesa pogađanja broja. Ako je izabran tačan broj, program treba da prestane sa radom.

Za slučajno generisanje broja između 0 i 100 koristićemo funkciju za kreiranje slučajnog decimalnog broja između 0 i 1 (*Random Number (0-1)*). Dobijeni broj ćemo pomnožiti sa 100 i potom zaokružiti na najbliži ceo broj (*Round Toward -Infinity*). Na ovaj način smo dobili slučajan broj. Sada je potrebno da omogućimo korisniku da unese negde broj koji misli da je računar generisao, pa ćemo za to postaviti kontrolu pod imenom *Pogađaj*. Svaki put kada korisnik unese neki broj potrebno je taj broj uporediti sa slučajno generisanim brojem i sprovesti dalju akciju u skladu sa tekstom zadatka. Iz tog razloga koristićemo *event* strukturu, koja će proveravati svaki put kada korisnik unese broj, da li je taj broj veći od pogođenog i u skladu sa tim ispisati određenu poruku na ekran. Dodavanje određenog događaja u *event* strukturu, u ovom slučaju reagovanje na promenu vrednosti kontrole *Pogađaj*, vrši se desnim klikom na *event* strukturu i izborom opcije *Edit Events Handled by This Case*. Ovim se dobija prozor prikazan na slici 19, gde je moguće izabrati tačan događaj na koji će se *event* struktura aktivirati.

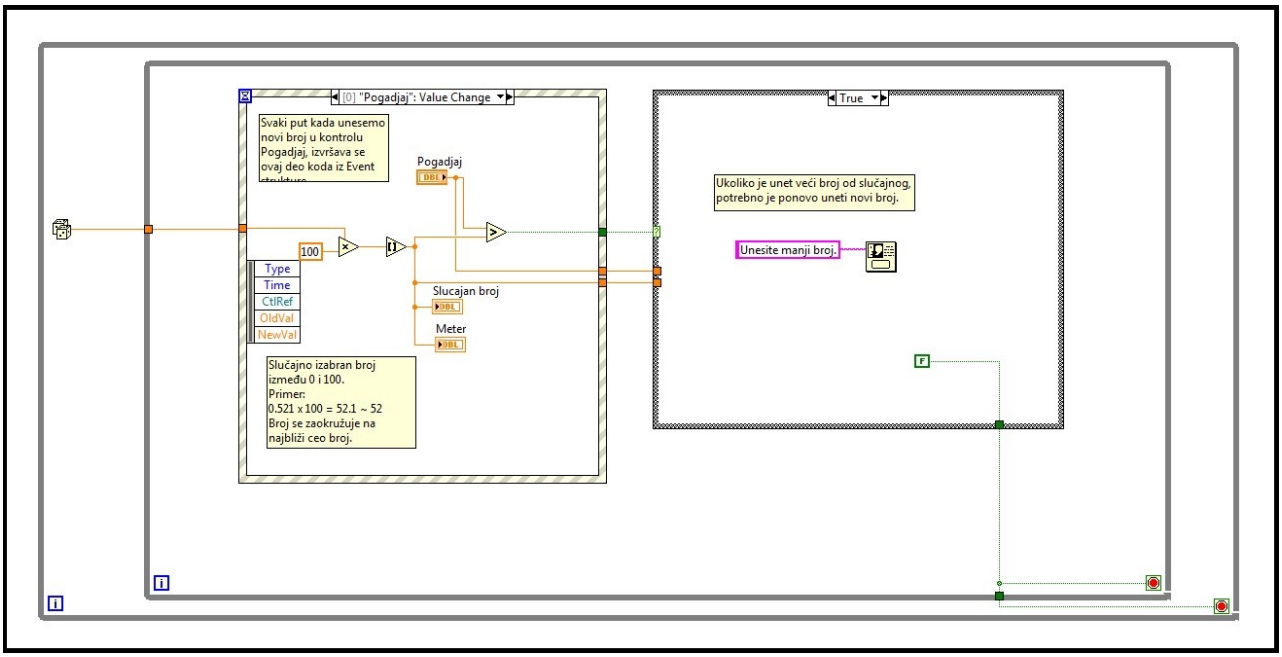


Slika 19: Kreiranje događaja u *event* strukturi.

Pošto je tekstom zadatka navedeno da se omogući izvršavanje programa sve dok korisnik ne pogodi slučajno generisan broj, prvo je potrebno omogućiti da se jednom generisan broj ne menja dok korisnik ne pogodi. To je rešeno tako što je postavljena jedna *While* petlja i u njoj samo funkcija za generisanje slučajno decimalnog broja, a potom taj broj predstavlja ulaz u drugu *While* petlju u kojoj se odvija ostatak zadatka. U skladu sa tekstom zadatka potrebno je prave vrednosti povezati na uslovne terminale petlji.

Nakon što je događaj kreiran, potrebno je omogućiti dalje izvršavanje zadatka. Izlazi iz *event* strukture biće rezultat poređenja, unešeni broj od strane korisnika i slučajno generisani broj. Rezultat poređenja se dovodi na terminal *case* strukture, gde imam dva moguća slučaja - *True* ili *False*. U slučaju *True* (unešeni broj je veći od generisanog, slika 20) potrebno je ispisati poruku korisniku da unese manji broj i da se omogući nastavak izvršavanja programa. Ispisivanje poruke na ekran vrši se pomoću funkcije nazavane *One Button Dialog*, a u oba terminala *While* petlji se upisuje logička kontanta vrednosti *False*. Ukoliko je pak vrednost unešenog broja manja od generisanje (slučaj *False*), potreb-

no je proveriti da li je uneti broj zapravo traženi broj. U ovom slučaju porede se ova dva broja i dobijaju se dva nova slučaja, ponovo *True* i *False*. Slučaj *False* (slika 21) treba da omogući ispis poruke "Unesite veći broj. Pokušajte ponovo!" i da omogući dalje izvršavanje programa. Poslednji slučaj je kada je korisnik konačno pogodio generisani broj (slika 22), gde potrebno zaustaviti izvršavanje programa i ispisati poruku korisniku da je broj uspešno pogođen.

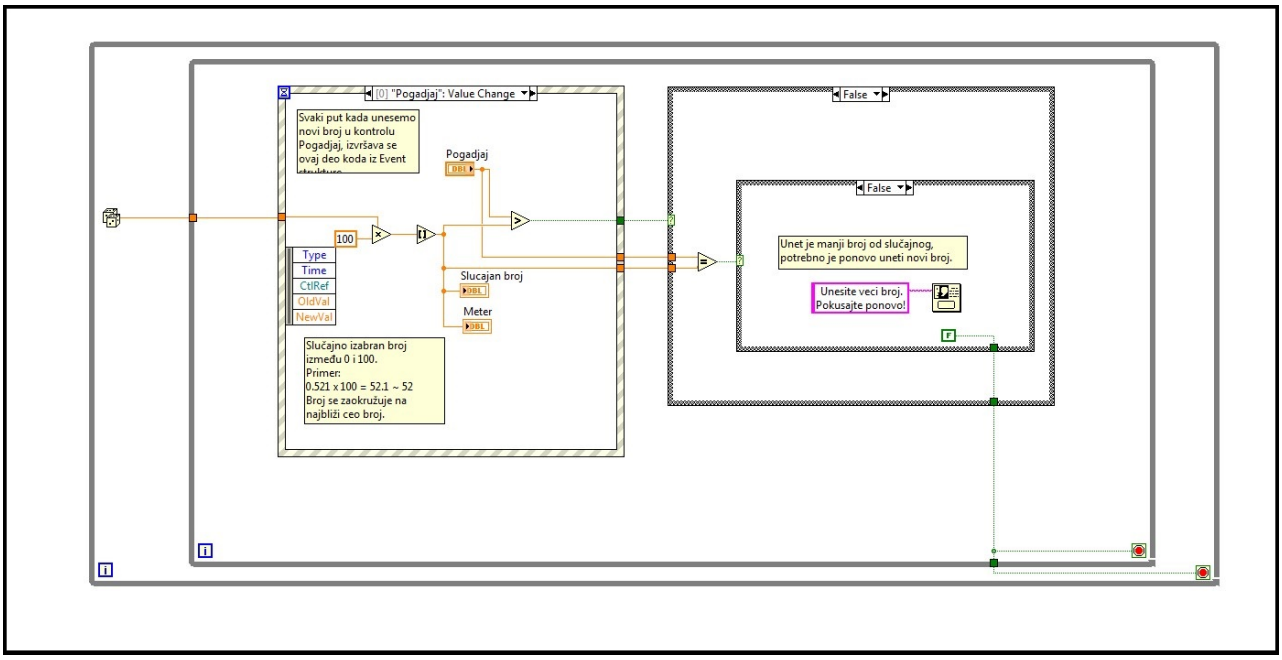


Slika 20: Blok dijagram rešenja.

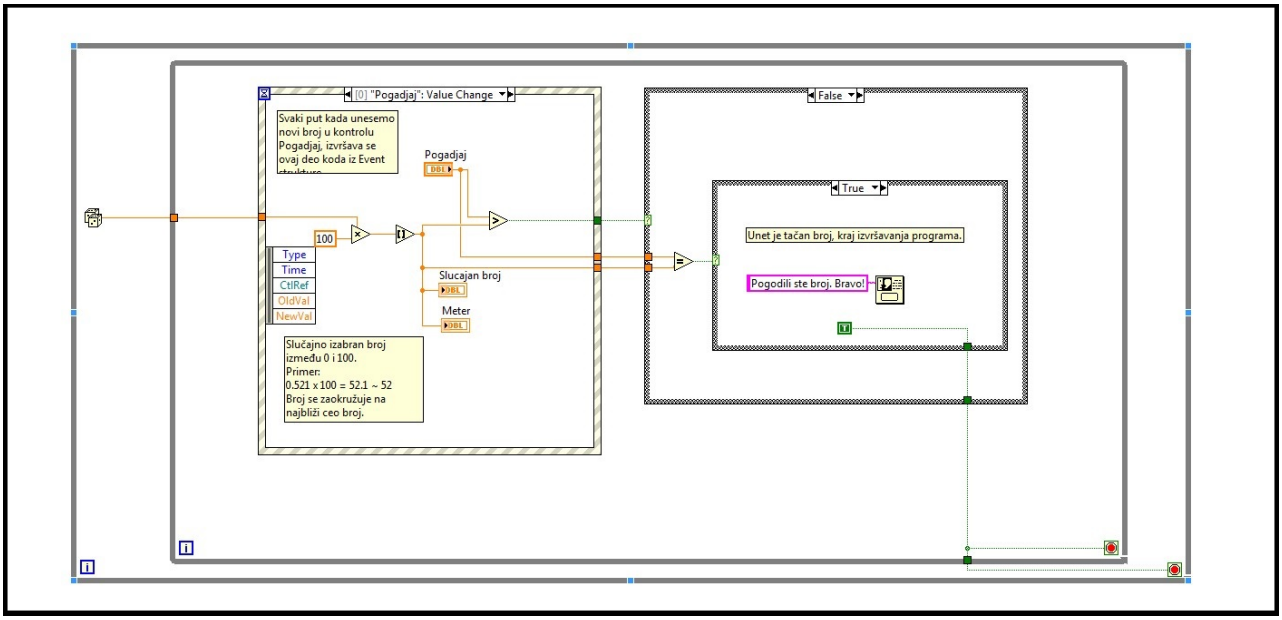
Flat sequence struktura se sastoji od više ramova (*frames*), koji se izvršavaju sekvencijalno. Korišćenjem *flat sequence* strukture obezbeđuje se sigurno sekvencijalno izvršavanje koda. Ramovi se izvršavaju sa leve na desnu stranu i to onda kada su svi potrebni podaci dostupni za njegovo izvršavanje. Kako jedan ram završi sa izvršavanjem, tako podaci napuštaju taj ram. To znači da ulaz jednog rama zavisi od izlaza drugog rama.

Jedan primer sa *flat sequence* strukturom prikazan je na slici 24. Korišćenjem *flat sequence* strukture potrebno je popuniti matricu dimenzija $5x5$ celim brojevima od 0 do 10, a zatim sabrati sve elemente matrice i ispisati zbir.

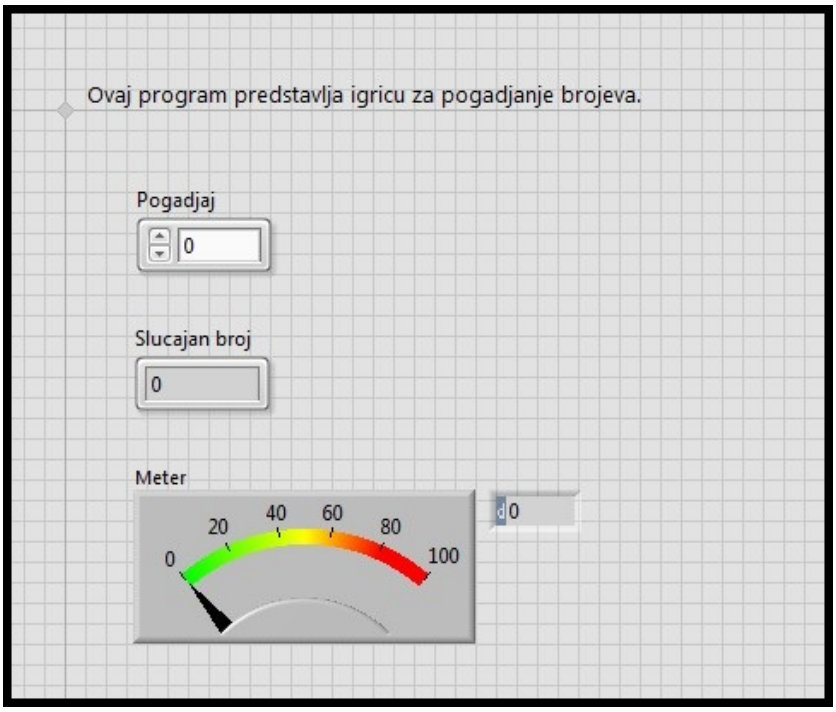
U prvom ramu postavljene su dve *For* petlje, kako bismo kreirali matricu. U oba terminala obe *For* petlje upisan je broj 5, jer matrica ima po 5 vrsta i 5 kolona. Pošto je matricu potrebno popuniti celim brojevima od 0 do 10, u unutrašnjoj *For* petlji iskoristili smo funkciju za slučajan izbor decimalnih brojeva između 0 i 1 (*Random Number (0-1)*), koji ćemo potom pomnožiti sa 10 i nakon toga zaokružiti na



Slika 21: Blok dijagram rešenja.



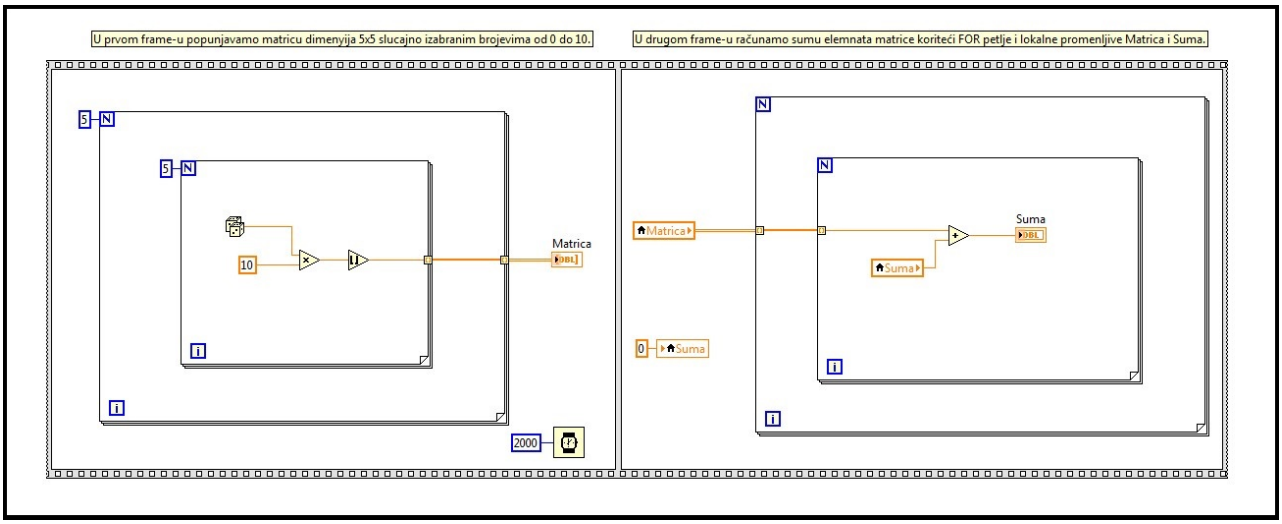
Slika 22: Blok dijagram rešenja.



Slika 23: Prednji panel primera sa *event* strukturom.

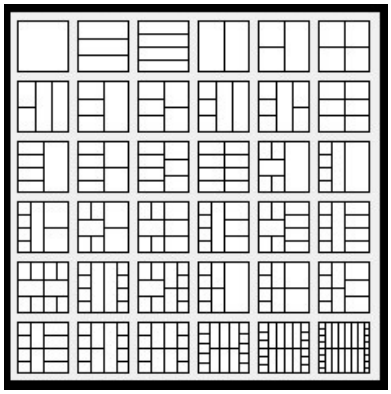
najbliži ceo broj (*Round Toward -Infinity*). Za jednu iteraciju spoljašnje *For* petlje izvršiće se 5 iteracija unutrašnje *For* petlje, odnosno kreiraće se 5 brojeva, odnosno popunićemo jedan red u matrici.

Kada smo popunili matricu potrebno je sada istu tu matricu iskoristimo u sledećem ramu, a to možemo uraditi pomoću lokalnih promenljivih. Od matrice iz prethodnog rama možemo kreirati lokalnu promenljivu i onda tu lokalnu promenljivu moguće je koristiti u drugom ramu. Lokalna promenljiva se kreira desnim klikom na indikator *Matrica* u prethodnom ramu i potom se izabere *Create* → *Local Variable*. Nakon kreiranja potrebno je da izaberemo da li želimo da čitamo ili pišemo u promenljivu. Obzirom da mi želimo da pročitamo elemente matrice, promenićemo režim u čitanje desnim klikom miša i izborom *Change To Read*. Zadatak je da saberemo sve elemente matrice i da tu sumu prikažemo. Ovo takođe možemo rešiti pomoću lokalnih promenljivih, tako što ćemo od indikatora u koji upisujuemo sumu kreirati lokalnu promenljivu. Pre samog sabiranja upisaćemo u lokalnu promeljivu vrednost nula, jer je na samom početku suma matrice 0, a potom ćemo pristupiti svakom elementu matrice uz pomoć dve *For* petlje i na tu sumu dodavati svaki element pojedinačno i na kraju dobiti ukupan zbir elemenata.



Slika 24: Primer korišćenja *flat sequence* strukture.

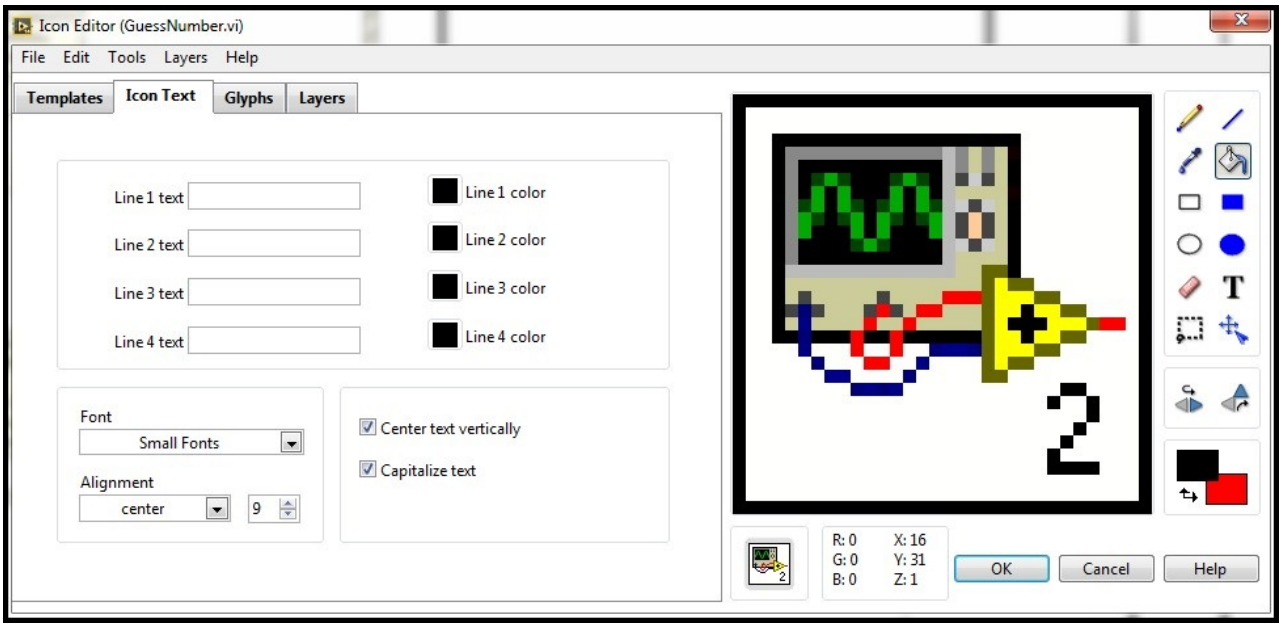
Kreiranje pod VI-a



Slika 25: Mogući izbori šeme pobezivanja.

Pod VI (*sub-VI*) predstavlja VI koji se koristi na blok dijagramu drugog VI-a. Bitnu ulogu igraju ikonica i šema povezivanja. Pomoću ikonice pod VI se identifikuje kada se pojavi na blok dijagramu drugog VI-a. Šema povezivanja diktira kako su terminali na ikonici povezani sa kontrolama i indikatorima na prednjem panelu.

Kada se novi VI kreira on ima podrazumevani izgled ikonice i šeme povezivanja. Podrazumevani prikaz ikonice može se videti na slici 26. Na istoj slici se takođe vidi i editor koji služi za kreiranje ikonice za pod VI. Ulazi i izlazi pod VI-a se definišu u šemi povezivanja. Postoji mnogo izbora za izgled šeme, što se može videti na slici 25. Svaki od njih se može rotirati, omogućavajući tako još više izbora. Podrazumevana šema poveivanja je oblika $4x2x2x4$. Šeme koje imaju više terminala za povezivanje mogu biti teške za povezivanje, dok one koje imaju malo terminala koji se vrlo brzo mogu popuniti. Prilikom biranja izgleda šeme, bitno je izabrati neku koja ima više terminala nego što nam je potrebno iz razloga što menjanje izgleda šeme terminala uzrokuje da pri pozivu pod VI u nekom VI žica se prekine. Terminali sa leve strane šeme se obično koriste kao ulazi u VI, a terminali sa desne strane kao izlazi.



Slika 26: Izgled editora za ikonice.

Zadaci za samostalni rad

1. Napraviti program (VI) koji izračunava površinu valjka (cilindra) na osnovu unetog poluprečnika baze i visine valjka.
2. Napraviti program (VI) koji omogućava sabiranje ili odzimanje dva broja u zavisnosti od odabrane operacije.
3. Napraviti program (VI) koji izračunava kvadratni koren pozitivnog broja, a ukoliko se unese negativan broj omogućiti ponovni unos broja uz odgovarajuću poruku na ekranu.
4. Napraviti program (VI) koji broji od 0 do 200 i brojač se uvećava na svaku sekundu. Brojač realizovati uz pomoć *For* petlje.
5. Napraviti program (VI) koji omogućava da se izračuna srednja vrednost elemenata matrice.
6. Napraviti program (VI) koji omogućava da se generiše matrica slučajno izabranih celih brojeva između 90 i 100. Matrica treba da bude dimenzija 5x5. U procesu generisanja matrice treba omogućiti da se svaki novi element matrice prikazuje na ekranu na svakih 300ms, te svaka vrsta matrice u promenljivu tipa niza na svakih 1,5 sekundi i na kraju da se prikaže cela matrica.

Rad sa fajlovima

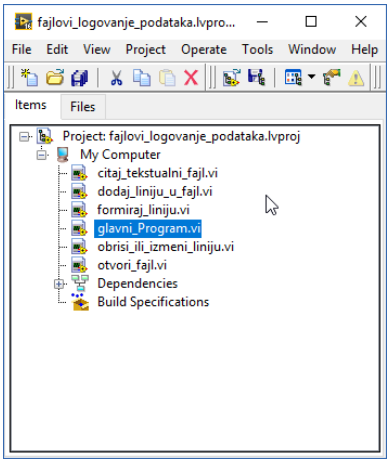
Rad sa fajlovima predstavlja jedan od najvažnijih koncepata u programiranju. U računarskom svetu fajlovi nam obezbeđuju mogućnost trajnog čuvanja podataka nakon završetka izvršavanja programa. Jasno je da savremeni računari ne bi mogli funkcionisati bez postojanja fajlova. Kada se govori o fajlovima u smislu programiranja, najčešće se spominje rukovanje tekstualnim i binarnim fajlovima. Na ovaj način se obezbeđuje najjednostavniji vid razmene informacija između procesa, bili oni na istom ili na različitim računarima. Zbog svega navedenog, jasno je da svaki programski jezik mora posedovati mogućnost rukovanja fajlovima. Naravno, i *LabView* poseduje mogućnost rada sa fajlovima. U ovom poglavlju bavićemo se radom sa tekstualnim fajlovima, međutim, *LabView* poseduje mogućnost jednostavnog rukovanja tabelarnim (*spreadsheet*) fajlovima.

Kada se govori o rukovanju tekstualnim fajlovima, moraju se razmotrini neke osnovne operacije za rukovanje fajlovima. Pre svega programski jezik mora da obezbedi mogućnost otvaranja i zatvaranja fajla. Što se tiče same sigurnosti prilikom otvaranja fajlova, obično se programski jezici oslanjaju na sloj operativnog sistema koji je zadužen za rukovanje datotekama. Odnosno, obično je davanje dozvole za otvaranje fajla i dalje rukovanje njime u nadležnosti operativnog sistema. Naravno, pored otvaranja i zatvaranja, programski jezik mora da obezbedi osnovne operacije za čitanje iz fajla i pisanje u fajl. Na kraju, očigledno je da mora postojati mogućnost za pozicioniranje u fajlu, da bi se omogućio upis na određenu lokaciju u fajlu, odnosno čitanje sa određene lokacije iz fajla. Navedena funkcionalnost obično se obezbeđuje mogućnošću promene pokazivača koji je zadužen za određivanje lokacije u fajlu. Pokazivač se može pomerati u odnosu na tri osnovne referentne pozicije. Te pozicije su početak fajla, kraj fajla, kao i trenutna vrednost pokazivača. Pokazivač se pomera za određeni broj lokacija (obično bajtova) u odnosu na referentnu poziciju. Broj lokacija za koje je potrebno pomeriti pokazivač programer definiše kroz ulazni parametar funkcije za pomeranje. Kao što je već rečeno, *LabView* poseduje sve navedene funkcionalnosti, a njihove pojedinosti će biti prodiskutovane u nastavku.

U ovom poglavlju prikazan je rad sa tekstualnim fajlovima u okviru *LabView* okruženja. Prikazano je otvaranje teksualnog fajla, pisanje, čitanje i izmena tekstualnog fajla. Sve navedene operacije opisane su kroz primer logovanja vrednost promenljive.

Primer - logovanje vrednosti promenljive

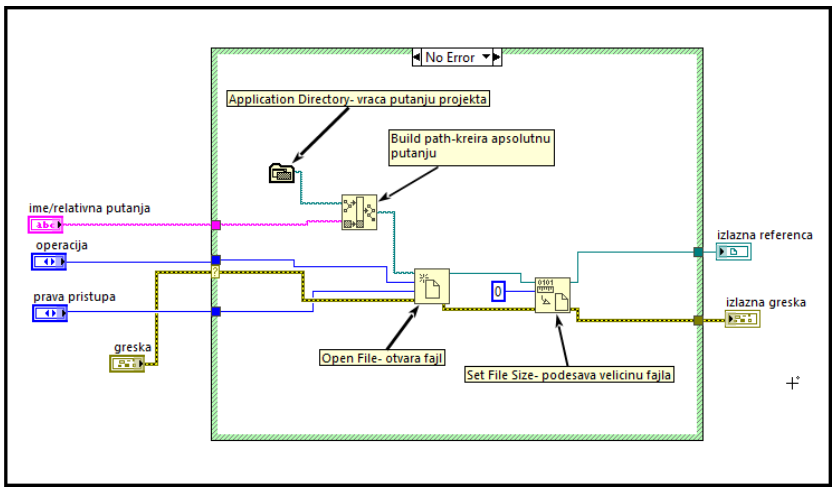
Zadatak je da se napravi *LabView* aplikacija koja upisuje podatke o nekoj promenljivoj u tekstualni fajl. Ti podaci podrazumevaju ime promenljive, vrednost, datum i vreme očitavanja. Po završetku očitavanja i logovanja podataka, pročitati ceo fajl i prikazati ga na jednom indikatoru. Nakon toga, potrebno je izbrisati petu liniju u fajlu i novi string prikazati na drugom indikatoru, i konačno potrebno je izmeniti treću liniju sa tako da na njenom mestu piše „greška".



Slika 27: Prikaz stabla projekta. Sadrži sve pomoćne funkcionalnosti i glavni program.

Da bi se ostvarila navedena funkcionalnost potrebno je implementirati niz operacija. Prvo je potrebno implementirati funkcionalnost za otvaranje tekstualnog fajla. Implementacija navedene funkcionalnosti prikazana je na Slici 28. Ulaz *ime/relativna putanja* predstavlja ime fajla, odnosno relativnu putanju fajla u odnosu na direktorijum gde se nalazi projekat. Ulaz *operacija* predstavlja izbor načina otvaranja fajla. Moguće opcije su otvaranje već postojećeg fajla, kreiranje novog fajla, zamena postojećeg fajla sa novim, otvaranje postojećeg ili kreiranje novog ako ne postoji fajl sa navedenim imenom, zamena postojećeg ili kreiranje novog ako ne postoji fajl sa navedenim imenom. Ulaz *prava pristupa* ima tri opcije: samo čitanje, samo pisanje ili oboje.

Konačno, ulaz *greška* predstavlja standardni *LabView* tip greške sa poljima koja govore da li postoji greška, koji je kod greške ako ona postoji i koji je opis greške. Izlazi iz ovog *subVI*-a su greška i referenca



Slika 28: Prikaz implementacije pomoćne funkcionalnosti čiji je zadatak otvaranje fajla.

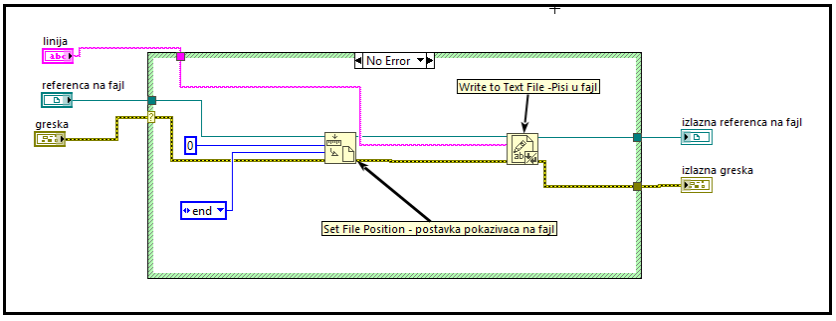
na fajl koji pokušavamo otvoriti.

Ukoliko na ulazu ove funkcionalnosti postoji greška, tada funkcija ne treba da radi otvaranje fajla, odnosno taj slučaj u *Case* strukturi ostaje prazan, potrebno je samo povezati *ulaznu grešku* na *izlaznu grešku* i na *izlaznu referencu* postaviti podrazumevanu(*default*) vrednost. Ukoliko na ulazu nema greške tada je prvo potrebno formirati apsolutnu putanju. To može da se uradi uz pomoć *Build path* funkcije, koja prima putanju do trenutne lokacije projekta i relativnu putanju do fajla. Relativna putanja nam je ulaz kompletnog *subVI*-a pa to možemo povezati na odgovarajuće mesto, dok putanju projekta možemo dobiti uz pomoć ugrađene *LabView* funkcije *Application Directory*. Nakon toga uz pomoć funkcije *Open File* otvara se fajl sa parametrima koji su definisani ulazima *operacija* i *prava pristupa*. Putanja do fajla je putanja koju smo kreirali sa *Build path* funkcijom. Nakon otvaranja fajla, dobijena referenca, koja je izlaz funkcije *Open File*, se povezuje na izlaz *izlazna referenca*. Da bismo lakše debagovali, odnosno prikazivali rezultate izvšavanja ovog programa, svaki put kada se otvori fajl obrisaćemo sve iz njega. To je moguće uraditi upotrebom ugrađene *LabView* funkcije *Set File Size*, postavljanjem veličine fajla na 0.

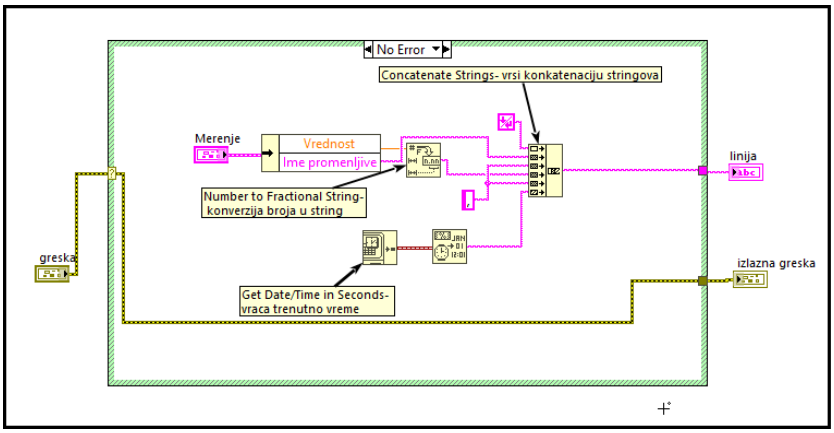
Pisanje u fajl

U ovom zadatku nove vrednosti potrebno je dodavati na kraj fajla. Zbog toga potrebno je kreirati *subVI* koji izvršava navedenu funkcionalnost. Implementacija navedene funkcionalnosti data je na Slici 29 i nazvana je *dodaj_liniju_u_fajl*. Ulaz *referenca na fajl*, kao što joj ime kaže, predstavlja referencu na fajl u koji želimo da upišemo tekst. Ulaz *linija* predstavlja string tip podatka, odnosno tekst koji želimo da dodamo u tekstualni fajl. Ulaz *greška*, kao i ranije, predstavlja standardni *LabView* tip greške. Izlaz iz ovog *subVI*-a treba da bude signal greške, kao i referenca na fajl.

Ukoliko na ulazu ove funkcionalnosti postoji greška, tada funkcija ne treba da se vrši upis u fajl, odnosno taj slučaj u *Case* strukturi ostaje



Slika 29: Prikaz implementacije dodavanja linije u tekstualni fajl



Slika 30: Prikaz subVI-a koji formira liniju za upis u tekstualni fajl

prazan, potrebno je samo povezati *ulaznu grašku* na *izlaznu grešku* i *ulaznu referencu* na *izlaznu referencu*. Ukoliko na ulazu ne postoji greška, potrebno je upisati tekst u fajl. Da bi tekst dodali na kraj fajla, potrebno je da postavimo trenutnu poziciju u fajlu na kraj fajla. To je moguće uraditi uz pomoć funkcije *Set File Position*, koja ima ulazne parametre *refnum*(referenca fajla), *offset* i *from*. Uz pomoć parametara *offset* i *from*, moguće se pozicionirati bilo gde u fajlu. Na ulaz *from* moguće je proslediti *start*, što predstavlja početak fajla, *end* što predstavlja kraj fajla i *current*, što predstavlja našu trenutnu poziciju u fajlu. Na ulaz parametra *offset* unosimo celi broj koji zapravo predstavlja za koliko bajtova hoćemo da se pomerimo u odnosu na parametar koji smo doveli na ulaz *from*. Dakle, ako izaberemo *start* i 5, pozicionirali bi se na peti bajt u fajlu. Pošto mi želimo da dodajemo na kraj fajla, potrebno je postaviti ulaz *from* na *end* i naravno *offset* na 0. Na izlazu funkcije dobijamo referencu sa podešenom pozicijom na kraj fajla. Nakon postavljanja pozicije na kraj fajla potrebno je upisati naš tekst u fajl. To se radi uz pomoć funkcije *Write to Text File*. Na ulaz ove funkcije potrebno je dovesti referencu na fajl, kao i tekst koji je potrebno upisati. Izlaz ove funkcije je referenca na fajl u koji je upisan tekst, kao i signal greške.

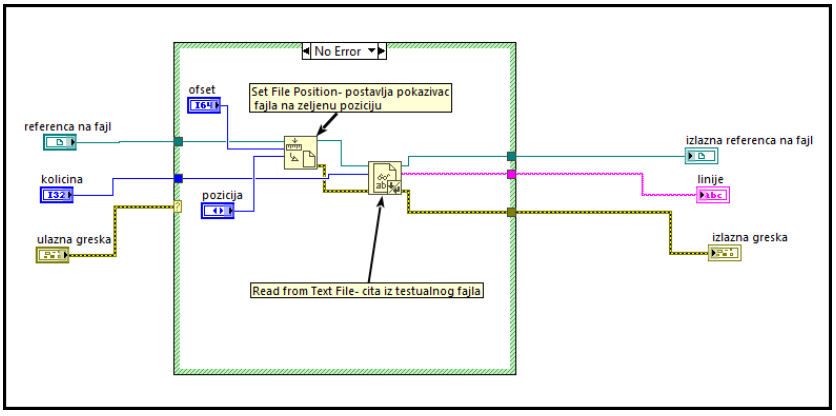
Implementirana je još jedna pomoćna funkcionalnost, formiraj liniju. Implementacija navedenog *subVI*-a data je na Slici 30. Zadatak ovog *subVI*-a je da procita podatke iz ulaza *merenje*, konvertuje ih u string tip podataka i formira jednu liniju koja će biti upisana u tekstualni fajl. Konverzija brojne vrednosti promenljive u string se vrši uz pomoć ugrađene *LabView* funkcije *Number To Fractional String*, dok se konkatenacija stringova takođe vrši pomoću ugrađene *LabView* funkcije *Concatenate Strings*. Radi lepšeg prikaza dodani su zarezi između svakog stringa. Pored imena i vrednosti promenljive, potrebno je upisati i vreme očitavanja promenljive. Trenutno vreme moguće je dobiti

uz pomoć *LabView* funkcije *Get Date/Time In Seconds*. Naravno, dobijeno vreme potrebno je konvertovati u string, što se može uraditi uz pomoć *LabView* funkcije *Format Date/Time String*. Izlaz iz ovog *subVI*-a je formirana *linija* koja će kasnije biti upisana u fajl, kao i signal greške. Naravno, ukoliko na ulazu postoji greška, ništa od navedenog postupka se ne izvršava. Potrebno je samo povezati grešku sa ulaza na izlaz i upisati prazan string na izlaz *linija*.

Čitanje iz fajla

Naredna pomoćna funkcionalnost predstavlja *subVI čitaj_tekstualni_fajl*. Implementacija je prikazana na Slici 31. Ulazi ovog *subVI*-a su *referenca na fajl* iz kojeg je potrebno čitati, *ofset* i *pozicija* koji nam služe da se pozicioniramo na određenu poziciju u fajlu, *količina* koji predstavlja broj karaktera ili linija koje želimo pročitati i *signal greške*. Izlaz je *referenca na fajl*, *signal greške* i *linije* (string koji sadrži sve pročitane linije).

Kao i kod svih ostalih funkcionalnosti, ukoliko postoji greška na ulazu potrebno je povezati *ulaznu referencu* na *izlaznu referencu*, a *ulaznu grešku* na *izlaznu grešku*. Pritom, u izlazni string *linije* se upisuje prazan string. Ukoliko ne postoji greška, potrebno je pokazivač fajla postaviti na poziciju koja je definisana ulazima *ofset* i *pozicija*. Navedeni postupak je identičan kao i kod dodavanja linije u fajl, odnosno radi se uz pomoć ugrađene *LabView* funkcije *Set File Position*. Nakon pozicioniranja u fajlu, potrebno je pročitati sadržaj fajla počevši od postavljene pozicije u fajlu. To je moguće uraditi sa ugrađenom *LabView* funkcijom *Read from Text File*. Ova funkcija kao ulazne parametre prima referencu na fajl iz kojeg se čita, signal greške, kao i broj karaktera/linija koje je potrebno pročitati. Ukoliko se funkciji proslijedi -1 na ulaz koji definiše količinu koju je potrebno pročitati, funkcija će pročitati kompletan sadržaj fajla od postavljene pozicije pa sve do kraja fajla. Pored *referen-*



Slika 31: Prikaz implementacije čitanja iz tekstualnog fajla.

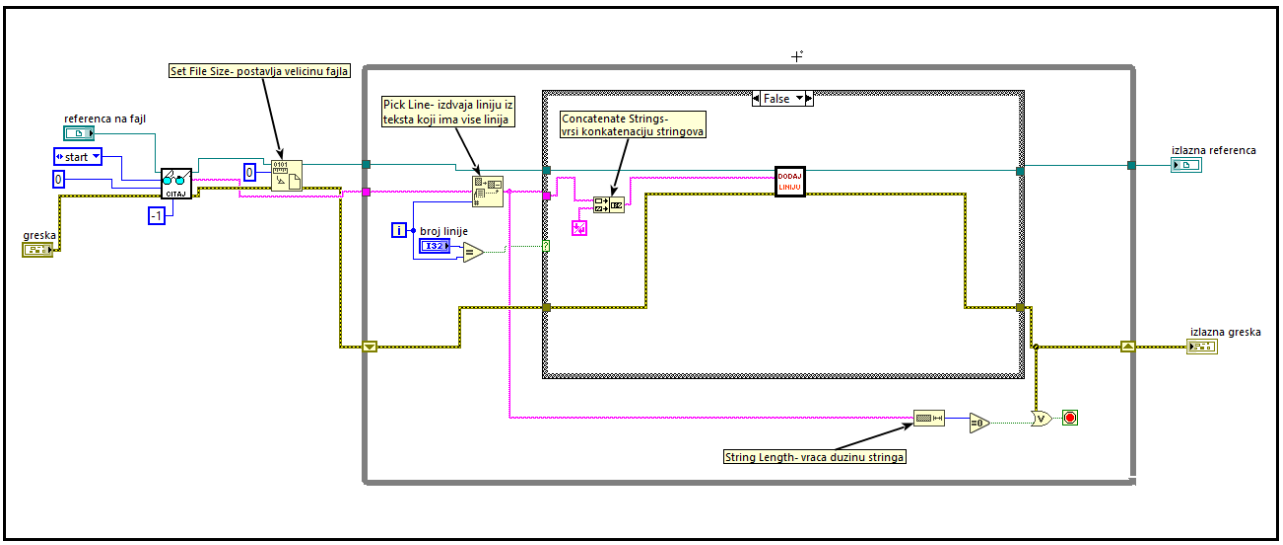
ce i *signala greške*, izlaz ove funkcionalnosti je i string koji predstavlja pročitani tekst iz fajla. Ovaj tekst potrebno je proseditи na izlaz *linije*.

Brisanje iz fajla (izmena sadržaja fajla)

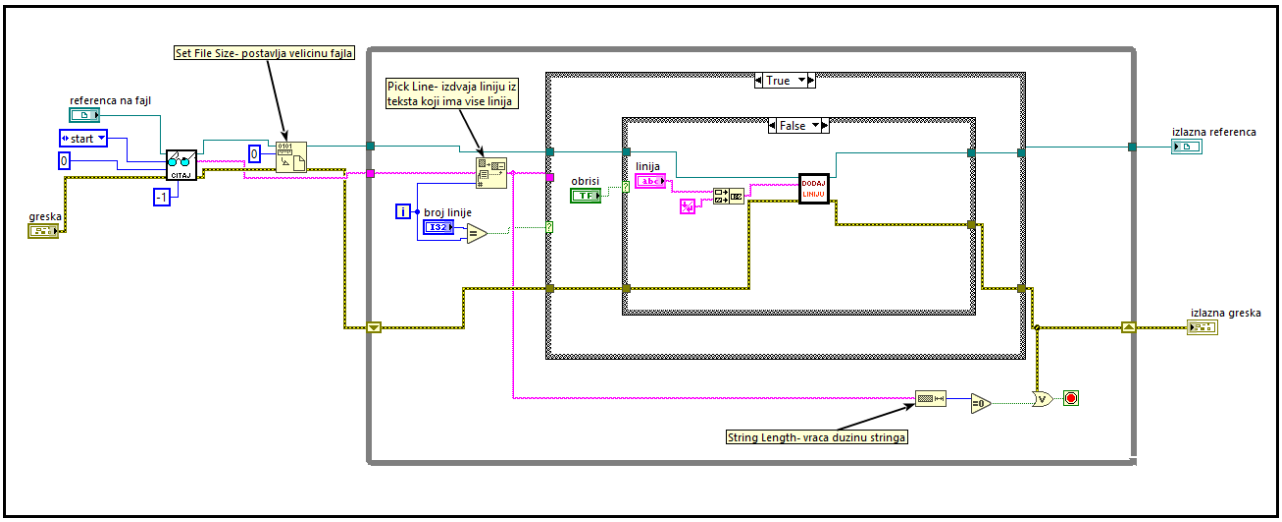
Poslednja pomoćna funkcionalnost ima zadatak da obezbedi mogućnost izmene ili brisanja određene linije u fajlu. Implementacija navedene funkcionalnosti prikazana je na Slikama 32 i 33. Prvi ulaz ovog *subVI*-a je *referenca na fajl* u kojem je potrebno izmeniti, odnosno izbrisati liniju. Ulaz *broj linije* predstavlja indeks linije koju je potrebno izmeniti, odnosno obrisati, ulaz *linija* predstavlja tekst koji je potrebno upisati umesto linije koja se menja, dok ulaz *obriši* govori da li se linija briše ili menja. Naravno, kao i kod svih ostalih *subVI*-a potrebno je imati *signal greške* na ulazu. Izlazi iz ovog *subVI*-a su *izlazna referenca na fajl* i *izlazna greška*. Opis implementacije dat je u nastavku.

Prvo je potrebno pročitati čitav sadržaj fajla, što radimo uz pomoć funkcionalnosti *čitaj_tekstualni_fajl*, čija je funkcionalnost opisana ranije. Da bi pročitali čitav fajl potrebno se pozicionirati na početak fajla, što se može uraditi tako što će se na ulaze *pozicija* i *ofset* funkcionalnosti *čitaj_tekstualni_fajl* postaviti *start* i 0, respektivno. Ovim se postiže da čitanje krene od početka fajla sa pomeranjem od 0 bajtova. Takođe, kao što je opisano ranije, na ulaz *količina* potrebno je proslediti -1 da bi pročitali kompletan fajl. Na izlazu *subVI*-a *čitaj_tekstualni_fajl* dobićemo čitav sadržaj fajla u okviru jednog stringa.

Ideja je da se prolazi kroz čitav sadržaj tako što se pročita linija, ukoliko ona nije linija koju treba izmeniti ili obrisati, linija se upisuje nazad u fajl. Ukoliko broj linije odgovara željenom indeksu, tada se



Slika 32: Prikaz implementacije brisanja i izmene linije u tekstualnom fajlu.



Slika 33: Prikaz implementacije brisanja i izmene linije u tekstualnom fajlu. Unutar *True* slučaja unutrašnje *Case* strukture potrebno je samo direktno povezati referencu na fajl, kao i signal greške.

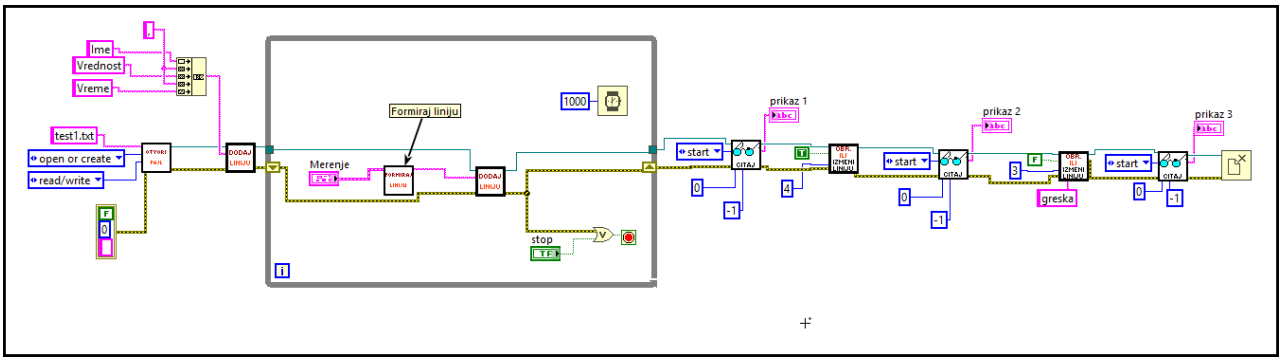
linija menja i upisuje u fajl, ukoliko je potrebno izmeniti liniju. Ukoliko je potrebno obrisati liniju, ona se jednostavno preskače i prelazi se na narednu liniju. Da bi se implementirala navedena ideja, prvo je neophodno „izbrisati" sve iz fajla. To se radi uporebom ugrađene *LabView* funkcije *Set File Size*, tako što se veličina fajla postavi na 0. Nakon toga potrebno je izdvajati liniju po liniju i izvršavati prethodno opisani algoritam. Izdvajanje linije se ovde radi upotrebom *While* petlje. Odnosno indeks koji govori koja je trenutno iteracija petlje se iskorištava da se izvuče linija iz teksta. Prekid izvršavanja *While* petlje se dešava kada se pročitaju sve linije, odnosno kada dužina pročitane linije bude 0. Linija iz teksta se izvlači upotrebom ugrađene *LabView* funkcije *Pick Line*, koja kao ulaze prima tekst koji ima više linija i indeks linije koju je potrebno izdvojiti. Paralelno sa izdvajanjem linije, vrši se provera da li je indeks linije koja se trenutno izdvaja jednak željenom indeksu linije koju je potrebno izbrisati, odnosno izmeniti. Ukoliko nije, tada je izdvojenu liniju potrebno upisati nazad u fajl. To se vrši upotrebom ranije implementiranog *subVI*-a *dodaj_liniju_u_fajl*. Pre upisa, potrebno je na kraj linije dodati karakter za kraj linije, da bi format fajla ostao nepromenjen. Navedena implementacija prikazana je na Slici 32. Ukoliko se indeks izdvojene linije podudara sa indeksom linije koju želimo izmeniti, odnosno izbrisati, potrebno je prvo proveriti da li je potrebno izmeniti ili izbrisati liniju. To se vrlo jednostavno može implementirati, jer je jedan od ulaza našeg *subVI*-a *bool* kontrola *obriši* koja govori da li se linija menja ili briše. Dovoljno je navedeni ulaz povezati na *Case* strukturu i time će oba slučaja biti pokrivena. Ukoliko je na ulazu obriši *True* konstanta, to znači da je liniju treba izbrisati, te u tom slučaju preskače se upis izdvojene linije, odnosno u

fajl se ne upisuje ništa. Implementacija navedene funkcionalnosti prikazana je na Slici 33. Poslednji slučaj je izmena linije, odnosno upis nove linije na mesto izdvojene linije. Taj upis je sličan upisu kada se treba vratiti stara linija u fajl, što je opisano ranije. Jedina razlika je u tome što se u ovom slučaju ne upisuje linija koja je pročitana iz fajla, nego string *linija*, koji zapravo predstavlja novi sadržaj date linije. Jasno je da je string *linija* zapravo ulaz u kompletan *subVI*. Dakle, upis je izvršen upotrebom ranije implementiranog *subVI*-a *dodaj_liniju_u_fajl*. Implementacija navedene funkcionalnosti prikazana je na Slici 33. Nakon što petlja prođe kroz celi fajl, na izlazu se dobija referenca na novi, izmenjeni, fajl. Naravno potrebno je biti pažljiv prilikom čitanja fajla. Ukoliko je fajl velik, nije najbolje rešenje da se čitav fajl pročita odjednom. Međutim, ovde je fokus na upoznavanju sa osnovnim pristupom rada sa tekstualnim fajlovima u *LabView* programskom okruženju, te je ovaj pristup zadovoljavajući.

Glavni program

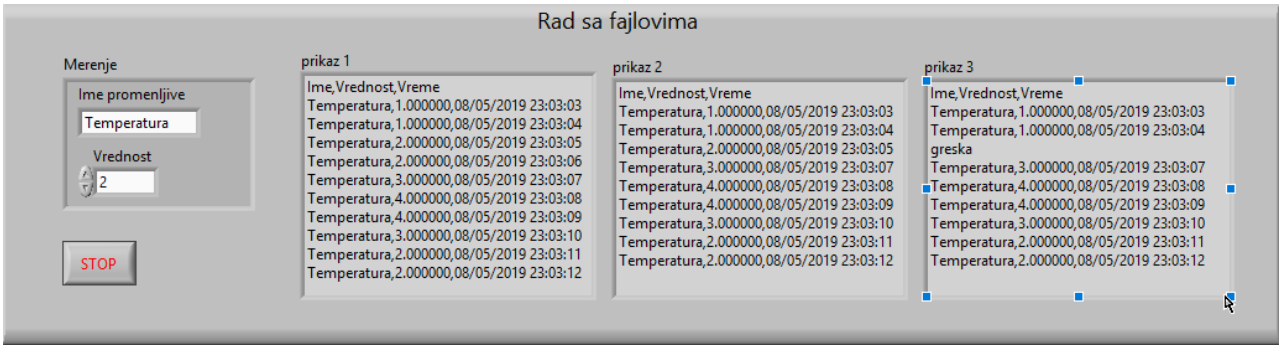
Nakon implementacije svih pomoćnih funkcionalnosti, može se pristupiti implementaciji glavnog programa. Implementacija glavnog programa prikazana je na Slici 34. Kao što je već rečeno, prvo je potrebno otvoriti fajl u koji će se upisivati podaci. To je izvršeno upotrebom *subVI*-a *otvori_fajl*. Na ulaz *ime/relativna putanja fajla* ovog *subVI*-a povezan string koji predstavlja ime tekstualnog fajla. Na ulaz *prava pristupa* povezan je način pristupa (*Read/Write* jer je potrebno i čitanje i pisanje), a na ulaz *operacija* povezan je način otvaranja fajla. U ovom slučaju biramo opciju *Open or create* koja će otvoriti fajl ukoliko postoji, a ukoliko ne postoji kreiraće novi fajl sa navedenim imenom. Radi lepšeg izgleda samog fajla, na početak je dodano zaglavlje. Upis zaglavlja u fajl izvšen je upotrebom *subVI*-a dodaj liniju u fajl, čija je implementacija opisana ranije.

Nakon upisa zaglavlja, izvršava se glavna *While* petlja u kojoj se u



Slika 34: Prikaz implementacije glavnog programa.

toku svake sekunde upiše ime i vrednost promenljive u fajl. Vrednosti iz *cluster*-a *Merenje* se prosleđuju u *subVI formiraj_liniju*, gde se podaci prebacuju u format koji je prilagodljiv za upis u fajl. Potom, upotrebom *subVI*-a *dodaj_liniju u fajl*, formirana linija se dodaje u fajl. Čekanje isteka jedne sekunde se obezbeđuje uz pomoć ugrađene *LabView* funkcije *Wait (ms)* kojoj se na ulaz prosleđuje konstanta 1000, što predstavlja 1000 milisekundi, odnosno jednu sekundu. Klikom na *Stop* petlja se završava. Takođe, ukoliko se desi greška, petlja treba da se zaustavi. Ova dva uslova su obezbeđena upotrebom logičke operacije *ili* između njih. Po završetku *While* petlje, potrebno je pročitati kompletan sadržaj fajla i prikazati ga na indikatoru. To je izvršeno upotrebom ranije opisanog *subVI*-a *čitaj_tekstualni_fajl*. Naravno, potrebno je poziciju u fajlu postaviti na početak fajla i staviti -1 na ulaz koji govori količinu podataka koji će biti pročitani da bi se pročitao kompletan sadržaj fajla. Izlaz iz *subVI*-a će biti string u kojem će se nalaziti kompletan sadržaj fajla. Ovaj string potrebno je povezati na string indikator *prikaz1* da bi sadržaj bio vidljiv na *Front panel*-u. Posle prikaza čitavog fajla, potrebno je obrisati petu liniju u fajlu. To je izvršeno upotrebom *subVI*-a *obrisi_ili_izmeni_liniju*. Na ulaz ovog *subVI*-a treba proslediti broj linije, u ovom slučaju to je broj 4 pošto indeksiranje kreće od 0, dok na ulaz *obrisati* treba proslediti *True* konstantu. Nakon brisanja linije, provere radi potrebno je na drugi indikator prikazati novi sadržaj fajla. Čitanje i prikaz se vrši na identičan način kao i pre izmene. Konačno, potrebno je izmeniti treću liniju u fajlu, što se radi na sličan način kao i brisanje koje je prethodno opisano. Razlika je tome što na ulaz *obrisati* potrebno postaviti *False* konstantu, a na ulaz *linija* upisati željeni tekst, u ovom slučaju „greška". Narvno, potrebno je proslediti odgovarajući redni broj linije. Nakon izmene linije, potrebno je prikazati novi sadržaj fajla na novom indikatoru. To se radi na identičan način kao i kod prethodna dva prikaza. Izgled *Front panel*-a nakon jednog izvršavanja kompletnog programa prikazan je na Slici 35.

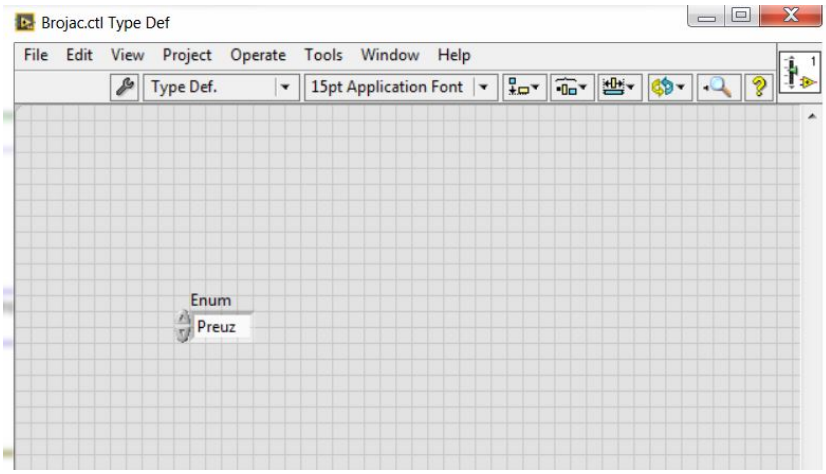


Slika 35: Prikaz *Front Panel*-a glavnog programa.

Brojač

Brojač je svakako jedna od najčešće korišćenih struktura u programiranju. Klasičan brojač može biti izveden na više načina. Ovde ćemo prvo prikazati kako bi izveli brojač kao **pogon akcije** (*action engine*). Svakako osnovne karakteristike **pogona akcije** su:

1. pogon akcije sadrži petlju koja poseduje neinicijalizovani **pomerajući registar** (*shift register*) ili **povratni čvor** (*feedback node*) koji služe kao elementi za čuvanje podataka
2. ovakva petlja se izvršava samo jednom.

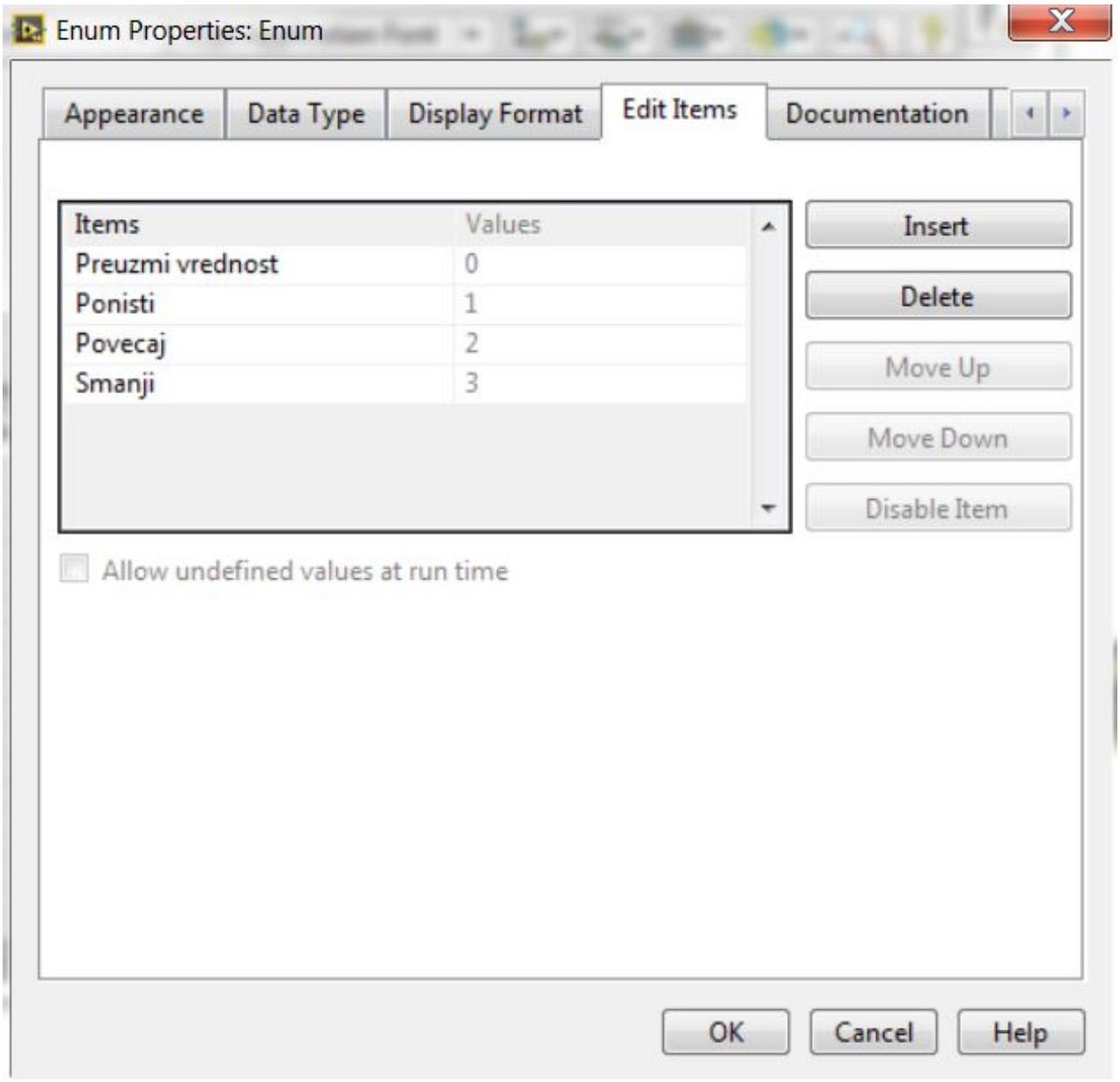


Slika 36: *Front panel* kontrole brojača

Na Slici 36 je ilustorovano kako izgleda kontrola brojača. Ona treba da sadrži *enum* kontrolu koja treba da sadrži sve nazive (a ne i definicije) svih metoda koje brojač treba da poseduje. U osnovnoj izvedbi to su metode: očitaj vrednost brojača, povećaj, te smanji vrednost brojača, kao i vrati brojač na 0 (odnosno resetuj brojač). Struktura *enum* kontrole data je na Slici 37.

Front panel brojača je prikazan na Slici 38. On sadrži jednu kontrolu čiji sadržaj je već objašnjen. Takođe sadrži celobrojni indikator, koji predstavlja vrednost brojača, kao i kontrolu i indikator signala greške koji omogućuju praćenja signala greške.

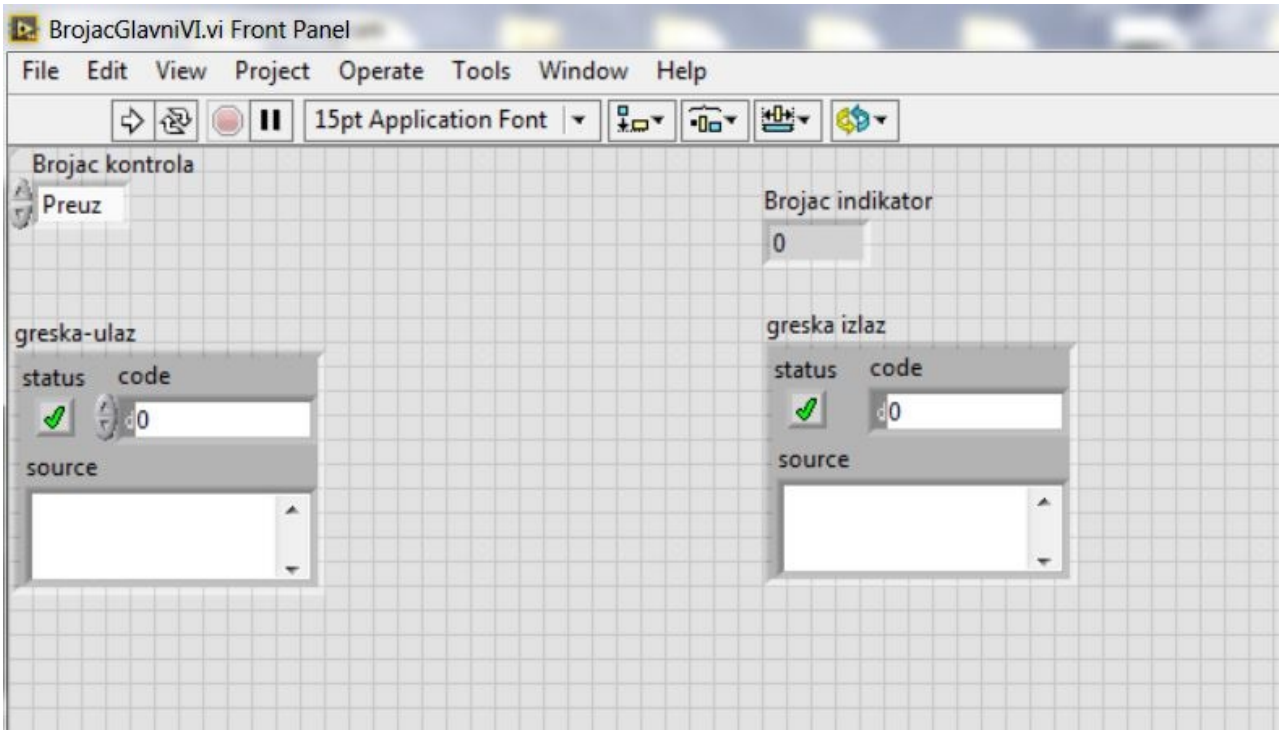
Pri kreiranju *custom control*-e potrebno je kao vrstu kontrole izabrati *Type Def.* iz menija *front panel*-a kontrole pored ikonice ključa. Ovo omogućava definiciju kontrole kao novog tipa i omogućava da se promene strukture kontrole automatski ažuriraju svuda u programu gde postoji instanca te kontrole.



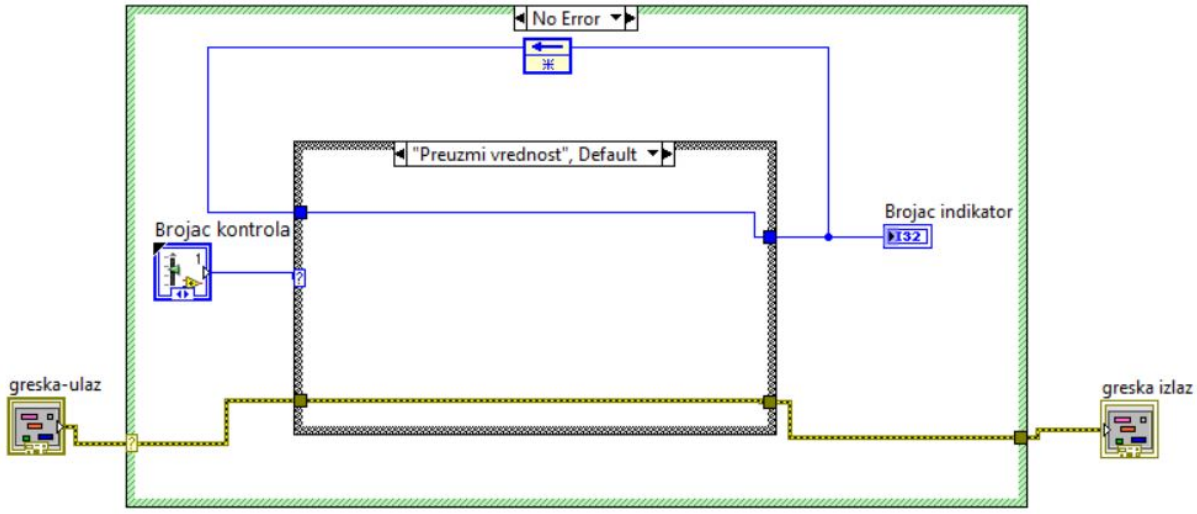
Slika 37: Enumerator kontrole brojača

Sama implementacija brojača je data na slikama 39, 40, **??**, 42. Karakteristično za svaku od ovih slika je da su one vezane za jedan blok dijagram, a samo u zavisnosti od izbora *Case* strukture na čiji ulaz je dovedena brojač kontrola izvršavaće se drugačiji slučaj. Svaki od ova 4 slučaja predstavlja metodu brojača.

Metoda *preuzmi vrednost* omogućava da se prikaže trenutna vrednost brojača, stoga kroz *Case* strukturu samo prolazi žica od ulaza do izlaza. Naravno, da bi ovakva implementacija imala smisla potrebno je ta 2 terminala (ulazni i izlazni) spojiti **povratnim čvorom** (*Feedback no-*



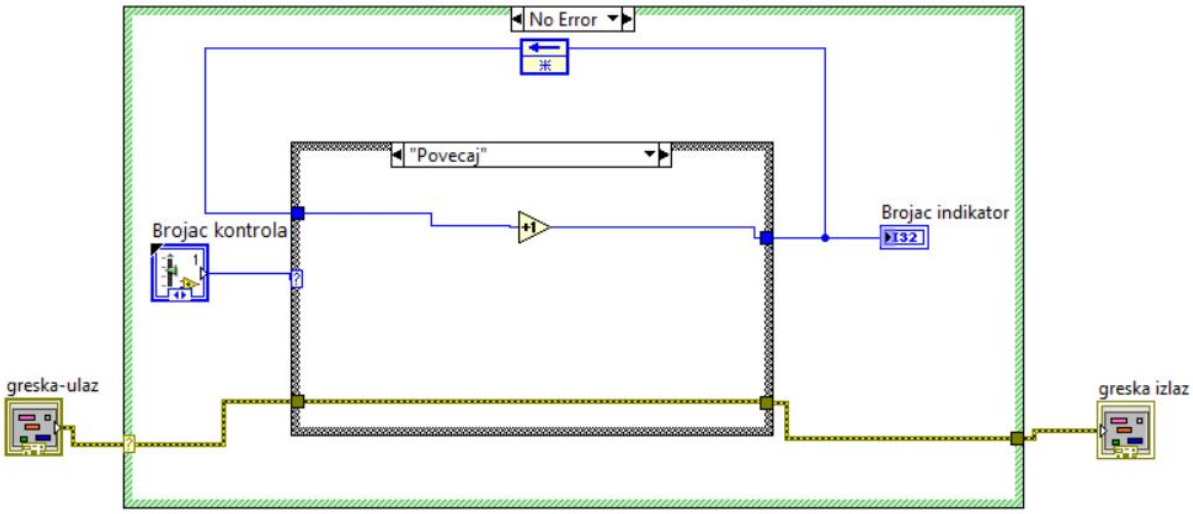
Slika 38: *Front panel* brojača



Slika 39: Metod brojača - očitaj vrednost brojača

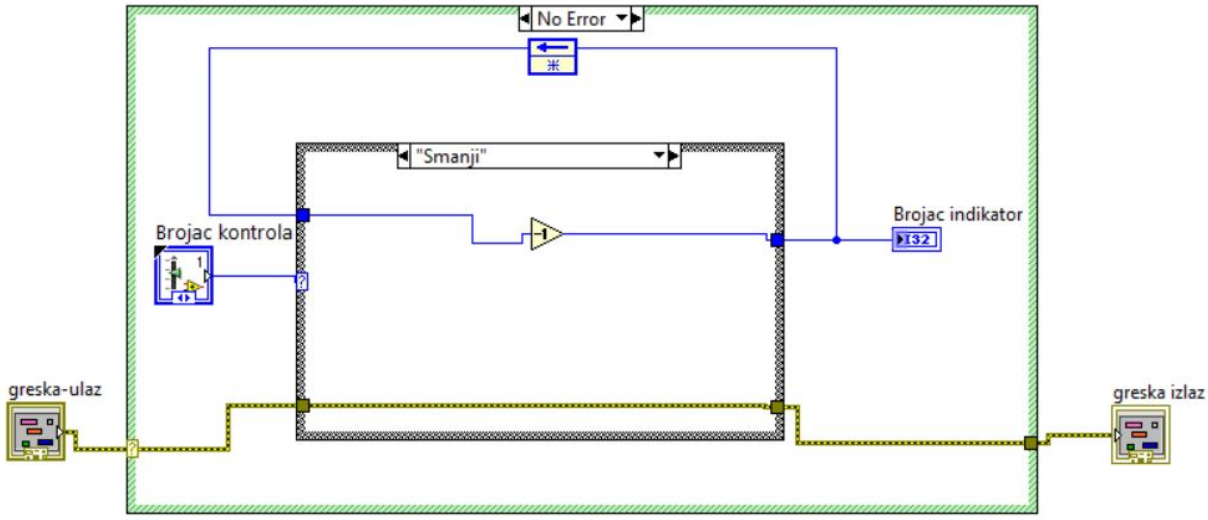
de-om) da bi se omogućilo da funkcija pamti vprethodnu vrednost do narednog poziva. Isti način implementacije je naravno u slučaju svih metoda.

Metoda *Povećaj* povećava vrednost brojača za 1 i ilustrovana je na Slici 40, dok metoda *Smanji* smanjuje vrednost brojača za 1 i ilustrova-



Slika 40: Metod brojača - povećaj vrednost brojača

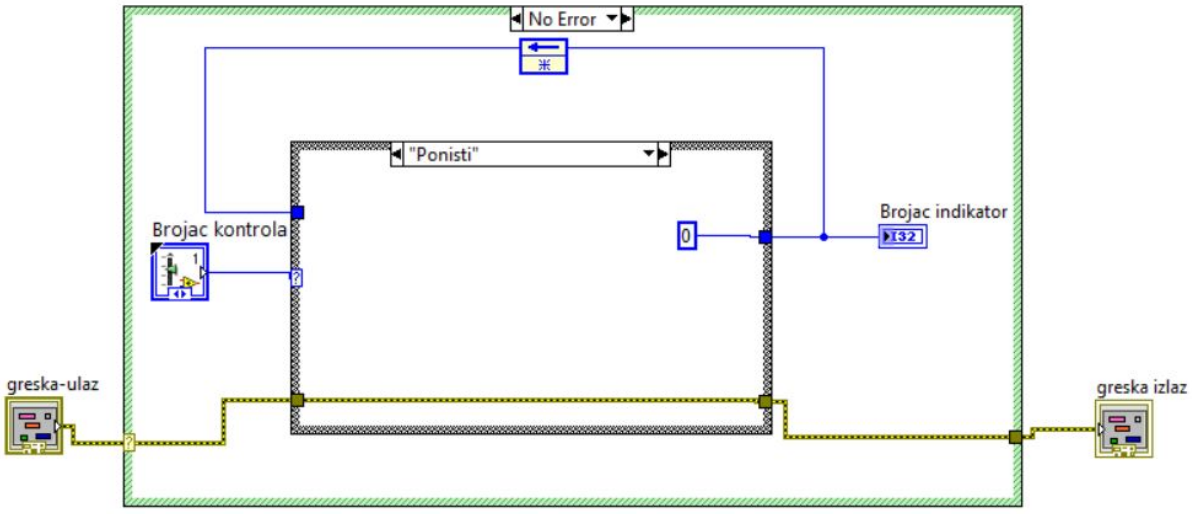
na je na Slici 41. Realizacija oba ova stanja je izvedena na taj način što je promenljiva stanja iz tekuće iteracije povećana, odnosno smanjena za 1 i ta vrednost je prosleđena u narednu iteraciju.



Slika 41: Metod brojača - smanji vrednost brojača

Poslednja metoda koju sadrži *Brojač* je medota *Poništi* koja resetuje brojač na 0 i prikazana je na Slici 42. Ona je relizovana na taj način što se promenljiva stanja u svakoj iteraciji kada je pozvana ova metoda resetuje na 0.

Svakako, kao što se vidi na ilustraciji svih ovih slika, prikazani su



Slika 42: Metod brojača - poništi vrednost brojača na 0

samo *no error Case*-ovi. U slučaju *error Case*-a samo se prosleđuje signal greške sa ulazne kontrole greške na izlaznu. Takva realizacija omogućava da se uhvati greška u bilo kom stanju *Brojača*.

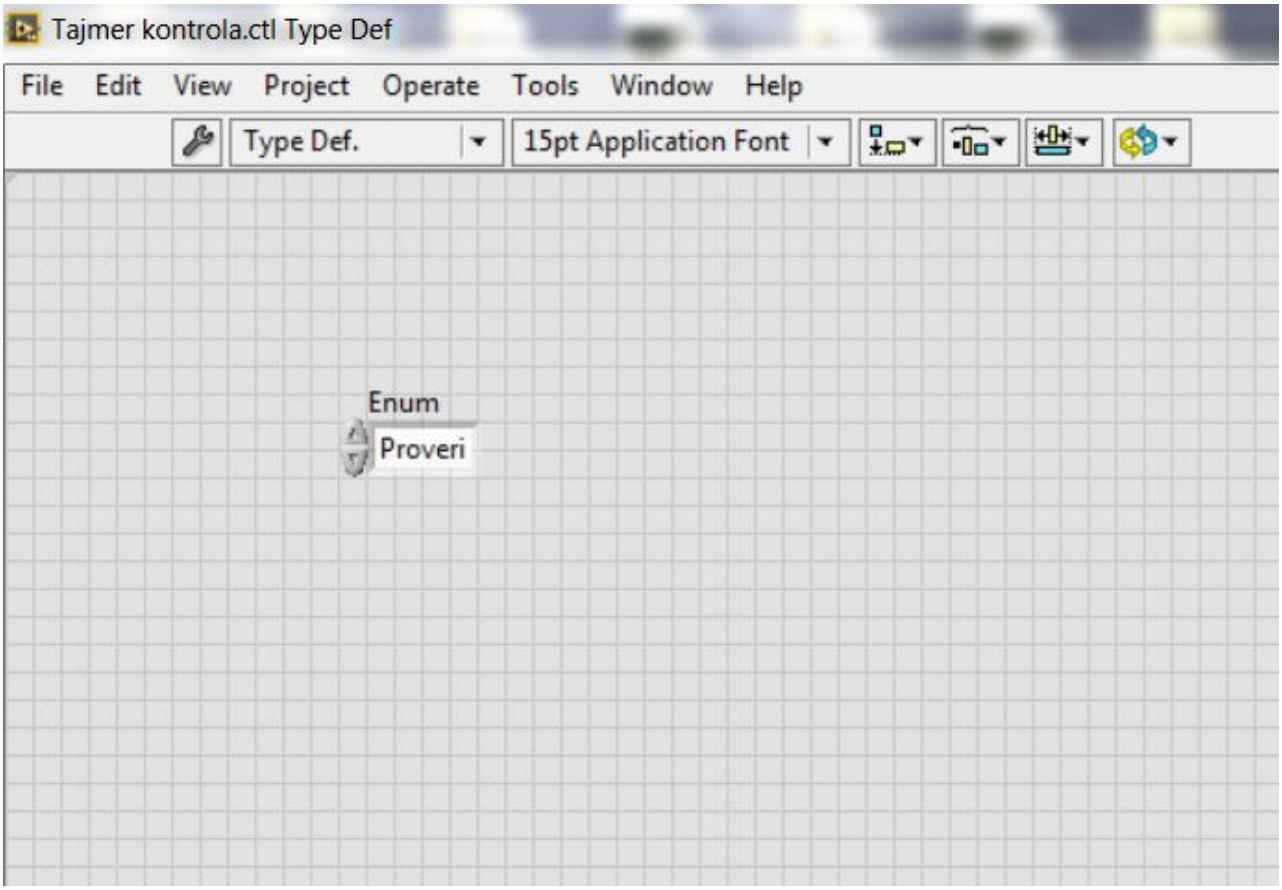
Postoje više alternativnih načina za realizaciju pogona akcije - realizacija preko povratnog čvora, *While* ili *For* petlje sa *shift* registrima. Svi ovi pristupi omogućavaju identičnu funkcionalnost.

Vežno je napomenuti da kod svih realizacija pogona akcija ili *shift* registri ili povratni čvor moraju biti neinicijalizovani.

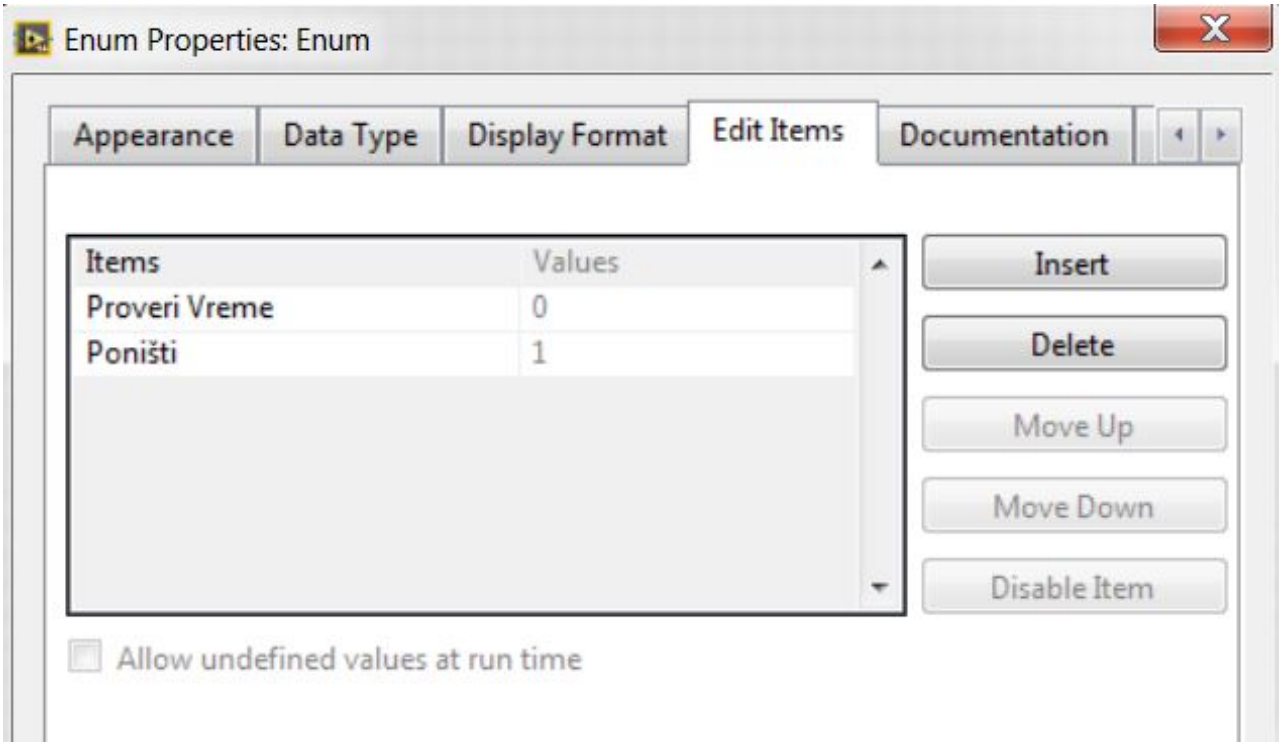
Tajmer

Kao što smo već napomenuli jedna od osnovnih metoda projektovanja u LabView-u je projektovanje pomoću pogona akcija. Na ovaj način možemo izvesti veliki broj funkcija. Recimo tajmer, koji predstavlja jednu od najčešće korišćenih funckija u bilo kom vidu programiranja, može biti izveden na taj način. U ovoj implementaciji još detaljnije struktuiramo **pogon akcija**.

Na Slici 43 dat je izgled *Front panel*-a kontrole koja sadrži sve metode *Tajmera*.

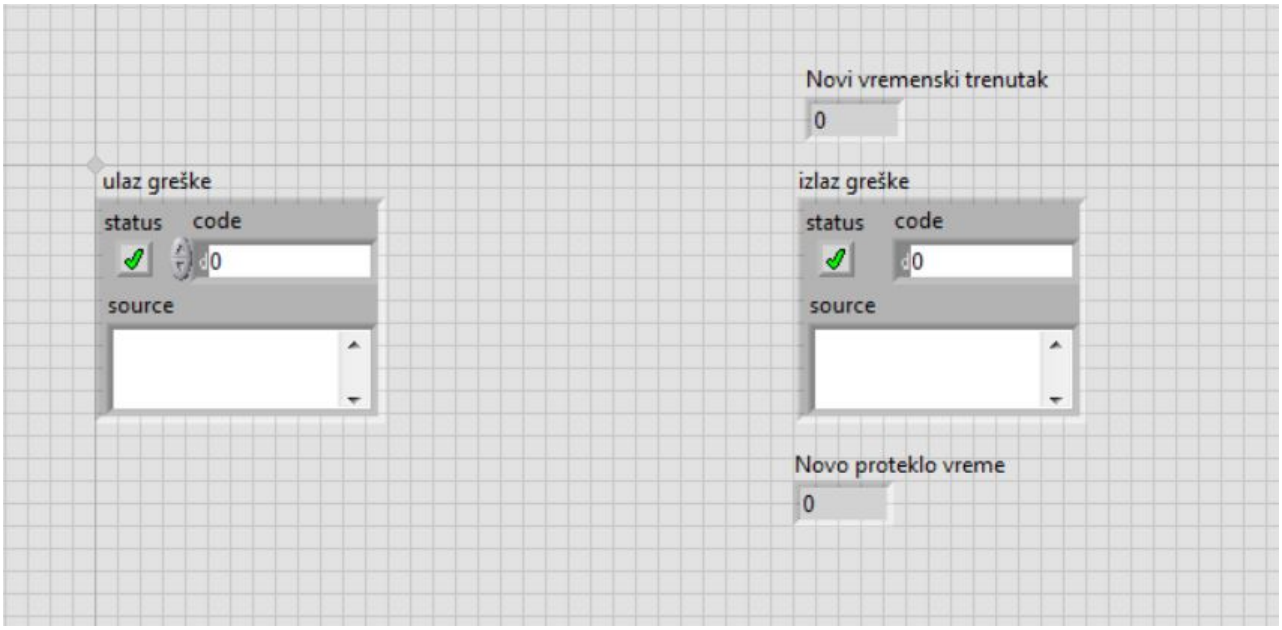


Slika 43: *Front panel* kontrole tajmera



Slika 44: Enumerator kontrole tajmera

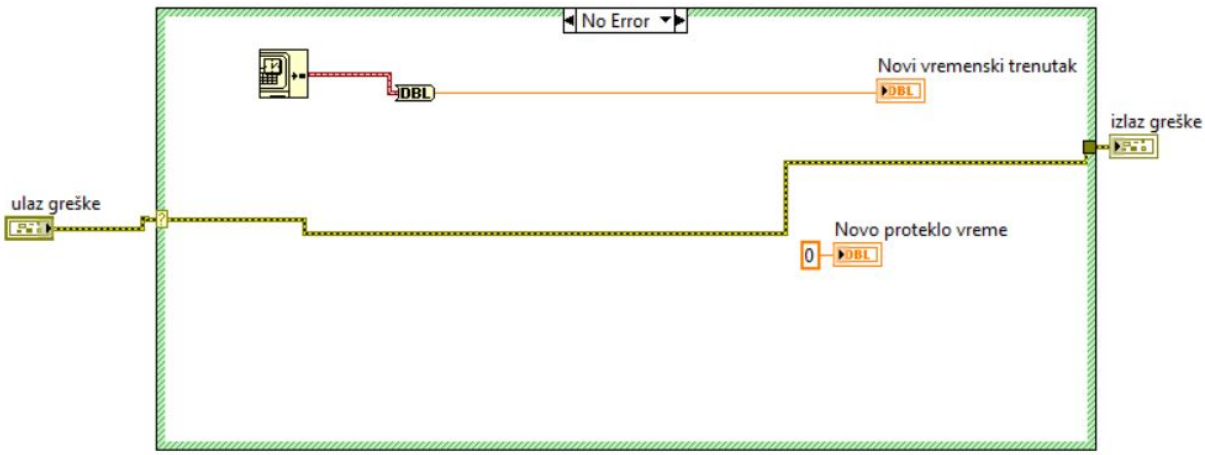
S druge strane, na Slici 44 data je struktura *enum* kontrole. *Tajmer* treba da omogući dve osnovne funkcije, a to je da broji vreme koje je proteklo između njegovih poziva i to dodaje na sveukupno vreme od početka aktivacije *tajmera* i da resetuje (poništi) to ukupno vreme.



Slika 45: *Front panel* metode ponisti tajmer

Metoda *Poništi tajmer* ima relativno jednostavnu realizaciju. Za sada nećemo spominjati promenljive stanja, ali treba uzeti u obzir sve elemente koji omogućavaju *tajmeru* da obavlja svoju funkciju kako treba. Ti elementi su svakako oni čija vrednost mora da se čuva od jednog do drugog poziva funkcije *Tajmer*. Ono što svakako mora da se čuva je trenutna vrednost sistemskog vremena, kao i trenutno vreme koje je proteklo od aktivacije tajmera (odnosno deaktivacije tajmera u slučaju njegovog resetovanja).

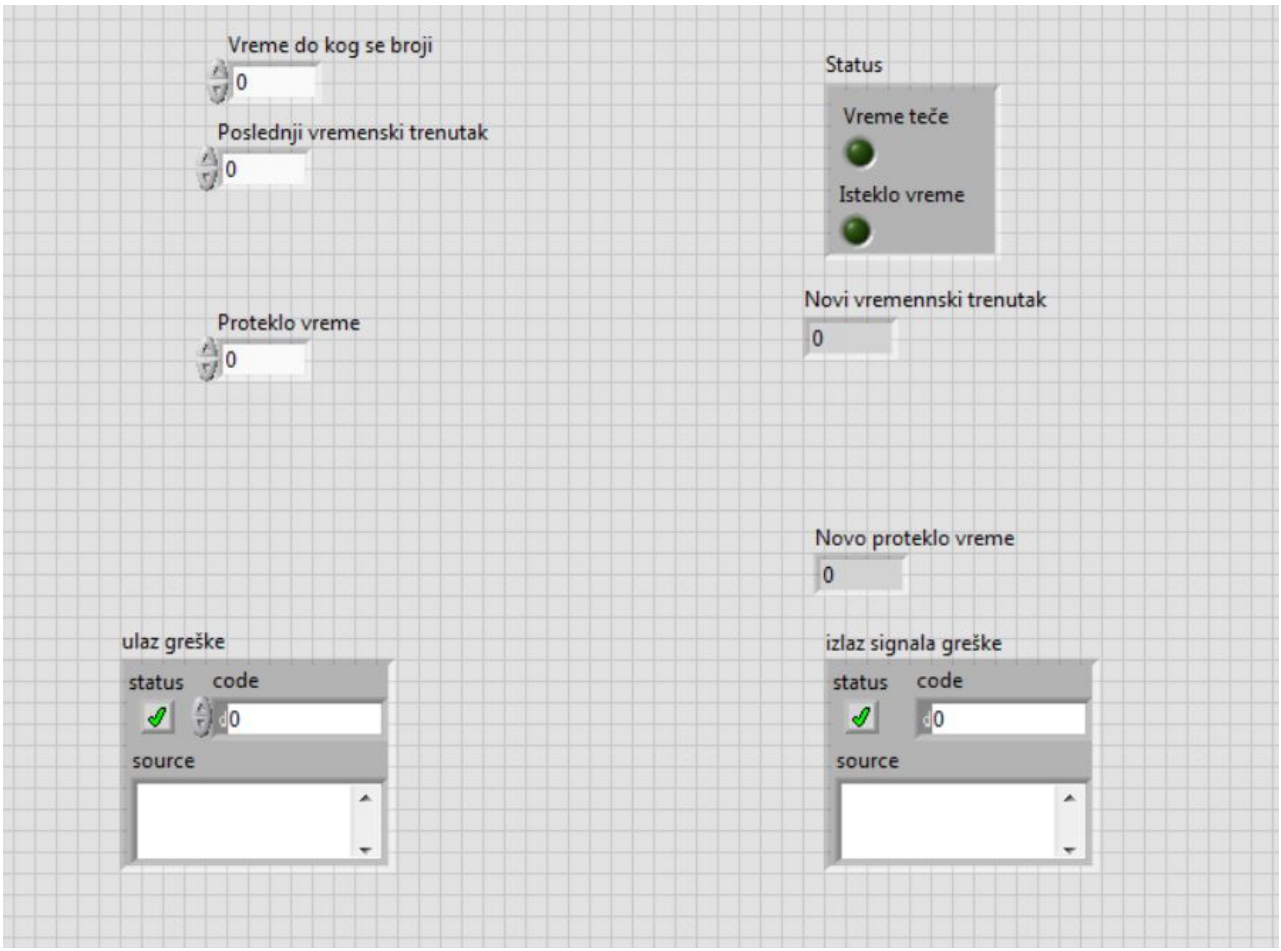
Čuvanje ova dva podatka je i omogućeno u metodi *Poništi*. Kao što se vidi na Slici 46 omogućeno je čitanje sistemskog vremena i njegovo upisivanje u indikator *Novi vremenski trenutak*. Ova realizacija omogućava da se u svakom pozivu metode *Poništi* ima evidencija u kom sistemskom trenutku je taj poziv ostvaren. Svaki sledeći put kada se pozove bilo koja metoda *Tajmera* znaće se kada je prema sistemskom vremenu poslednji put resetovan *Tajmer*. U indikator *Novo proisteklo vreme* se upisuje vrednost o svaki put kada se pozove metoda *Poništi*. *Front panel* ove metode ilustrovan je na Slici 45.



Slika 46: Metod tajmera - poništi tajmer

Sa druge strane metoda *Proveri vreme* mora da vodi računa o više detalja. Izgled njenog blok dijagarama je ilustrovan na Slici 48, dok je *Front panlel* dat na Slici 47. Ova metoda treba da vodi računa koliko je vremena proteklo od njenog poslednjeg poziva, kao i da se trenutno sistemsko vreme zapamti da bi bilo jasno koliko je vremena prošlo do narednog poziva. Informacija o trenutnom sistemskom vremenu se prosleđuje na indikator *Novi vremenski trenutak*. Jasno je da če u narednom pozivu ove funkcije informacija sa indikatora *Novi vremenski trenutak* biti prosleđena na *Poslednji vremenski trenutak*, stoga možemo ovo smatrati jednom promenljivom stanja sa kojom ćemo postupati

kroz proceduru akcije pogona.

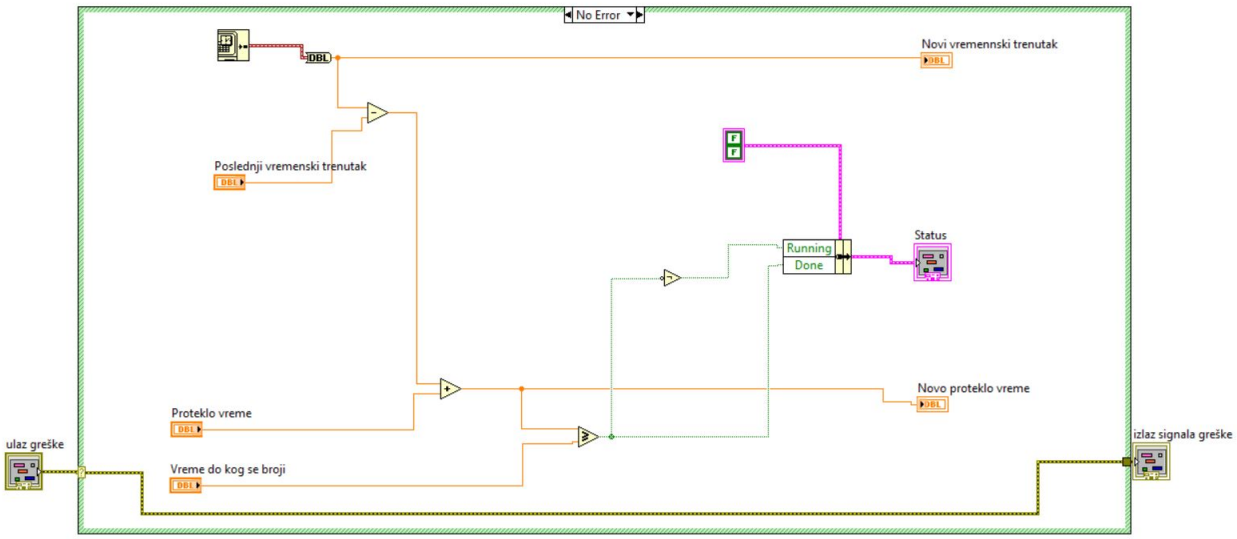


Slika 47: *Front panel* metode proveri vreme tajmera

Dalje je potrebno da se omogući da se izračuna koliko je vremena proteklo od poslednjeg ternutka kada je ova metoda bila pozvana. Ovo je realizovano na sledeći način: Od trenutnog sistemskog vremena oduzima s vremenski trenutak kada je bila poslednji put pozvana ova metoda (ova informacija je dostupna kroz kontrolu *Poslednji vremenski trenutak*) i taj rezultat se dodaje na kontrolu *Proteklo vreme*. Ova kontrola sadrži vreme koje je proteklo do trenutnog poziva metode. Dalje se rezultat ove operacije prosleđuje na indikator *Novo proteklo vreme*. Odavde postaje jasno da informaciju koja se nalazi na indikatoru *Novo proteklo vreme* treba proslediti u narednom pozivu ove metode na kontrolu *Proteklo vreme*. Odatle indikacija da će ovaj parametar potencijalno biti još jedna promenljiva stanja.

Pored ovoga, metoda vodi i računa da li je novo proteklo vreme veće ili jednako od zadatog vremena koliko *Tajmer* treba da broji. Ukoliko jeste na izlaz treba da se prosledi informacija da je *Tajmer* odbrojao - to se čini i kroz postavlje *boolean* indikatora *Done* na vrednost *true*, dok u

paraleli sa tim *boolean* indikator *Running* bude na inverznoj vrednosti *false*. U slučaju da je situacija obrnuta, tj. da je novo proteklo vreme manje od zadatog vremena koliko *Tajmer* treba da broji, na ova dva *boolean* indikatora se šalju suprotne vrednosti.

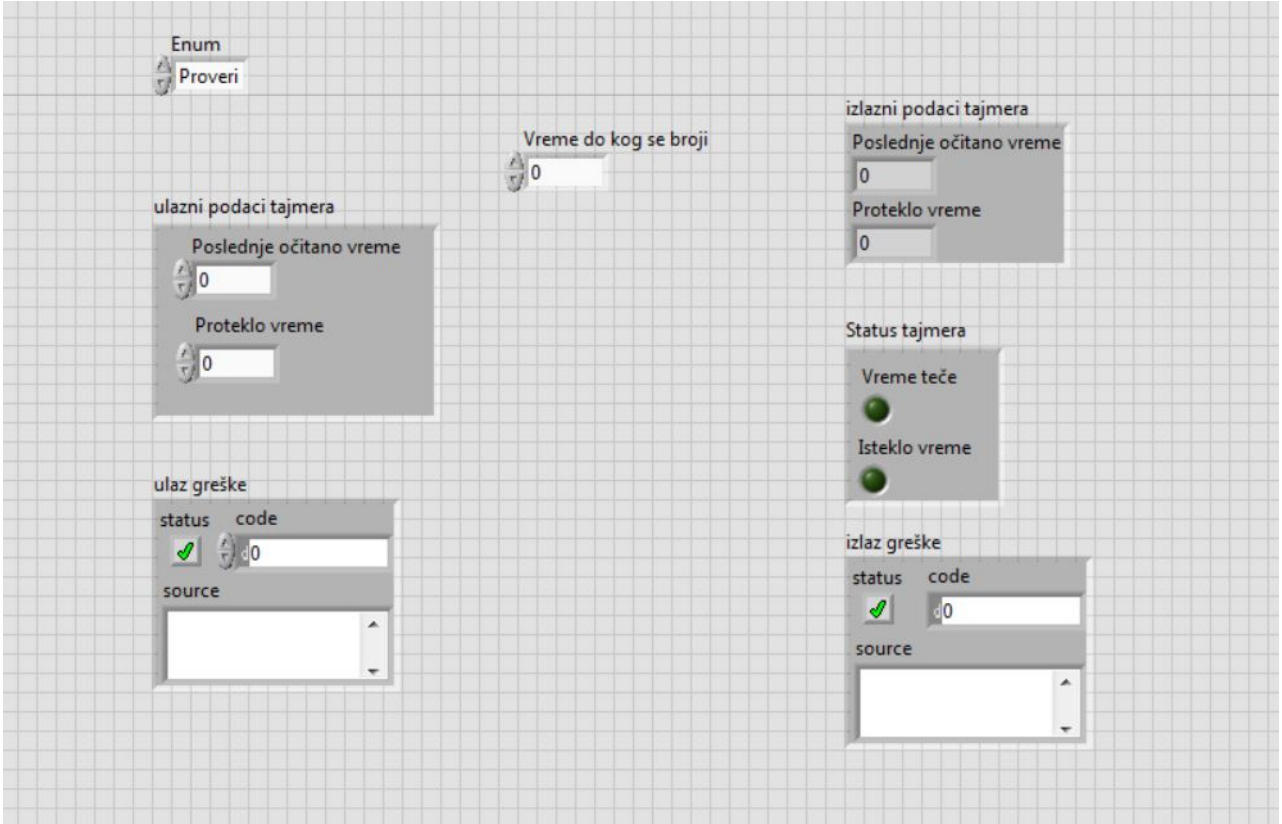


Slika 48: Metod brojača - proveri vreme tajmera

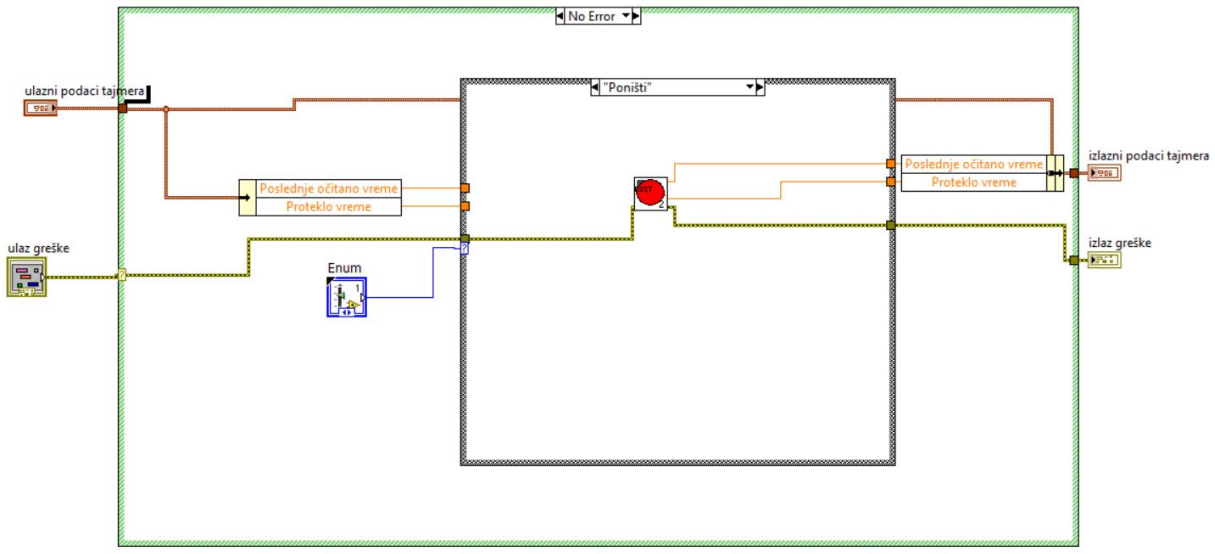
Dalje je potrebno ove dve metode integrisati u jezgro *Tajmera* čiji je blok dijagram ilustrovan na Slikama 50 i 51. Svaka od ovih slika daje reprezentaciju jezgra u slučaju da je aktivna jedna ili druga gore navedena metoda. Jezgro *Tajmera* služi da se u njemu definišu promenljive stanja, koje ćemo koristiti da bi sačuvali važne informacije iz jednog poziva u drugi. U ovom konkretnom slučaju su to *Poslednje očitano vreme* koje se mapira na ulaz *Poslednji vremenski trenutak* i na izlaz *Novi vremenski trenutak* i *Proteklo vreme* koje se mapira na ulaznu kontrolu *Proteklo vreme* i izlazni indikator *Novo proteklo vreme*. Ove dve promenljive stanja su spojene u *cluster* da bi se omogućila lakša manipulacija sa njima. U principu, ovo je način kako da omogućite bilo kom *pogonu stanja* da vodi računa o promenljivima koje su bitne da se čuvaju pri prelasku iz jedne iteracije u drugu.

Svakako, jezgro *Tajmera* sadrži ulaznu kontrolu *Enum* koja služi za odabir metode *Tajmera*. U slučaju da je izabrana metoda *Proveri vreme* potrebno ja na ulaz dovesti i informaciju do kada tajmer treba da broji. To je u ovom slučaju urađeno kroz kontrolu *Vreme do kog se broji*. Takođe, metoda *Proveri vreme* treba da omogući u jezgru da se dobiju informacije uo statusu *Tajmera* - da li radi ili je istekao. Ova informacija je dostupna kroz izlazni *cluster Status tajmera*.

Napomenimo da je jezgro *Tajmera* je podeljeno na dva slučaja - slučaj koji je do sada bio opisan je *no error* slučaj koji sadrži manipulaciju *Tajmerom* (resetanjem i proverom vremena), i na slučaj kada se javi



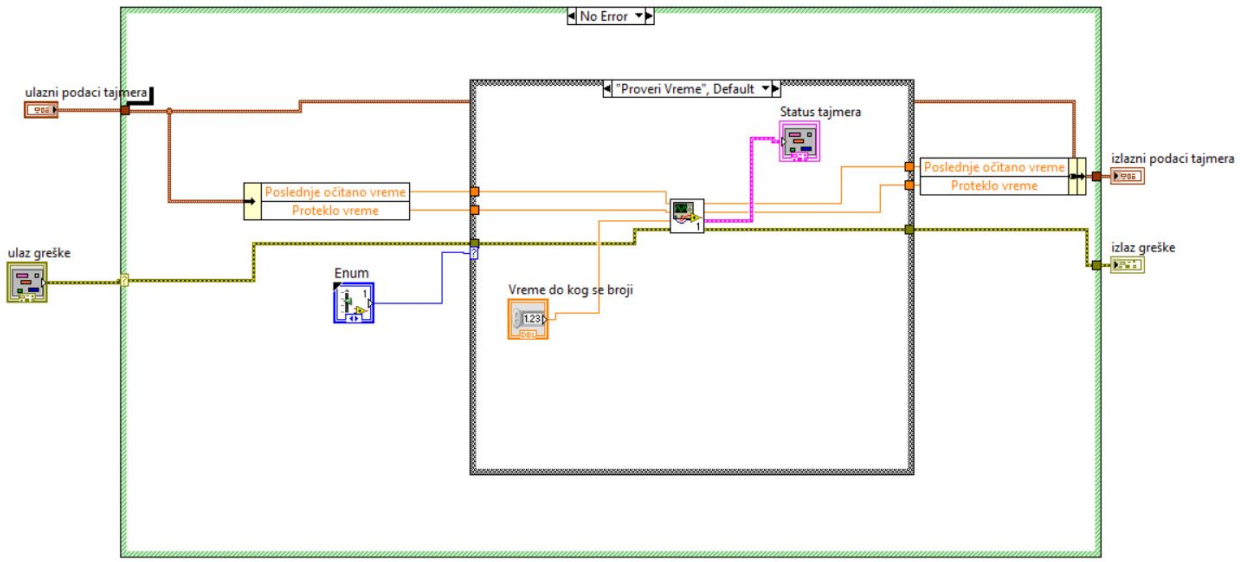
Slika 49: *Front panel* jezgra tajmera



Slika 50: Jezgro tajmera - poziv ponisti metode

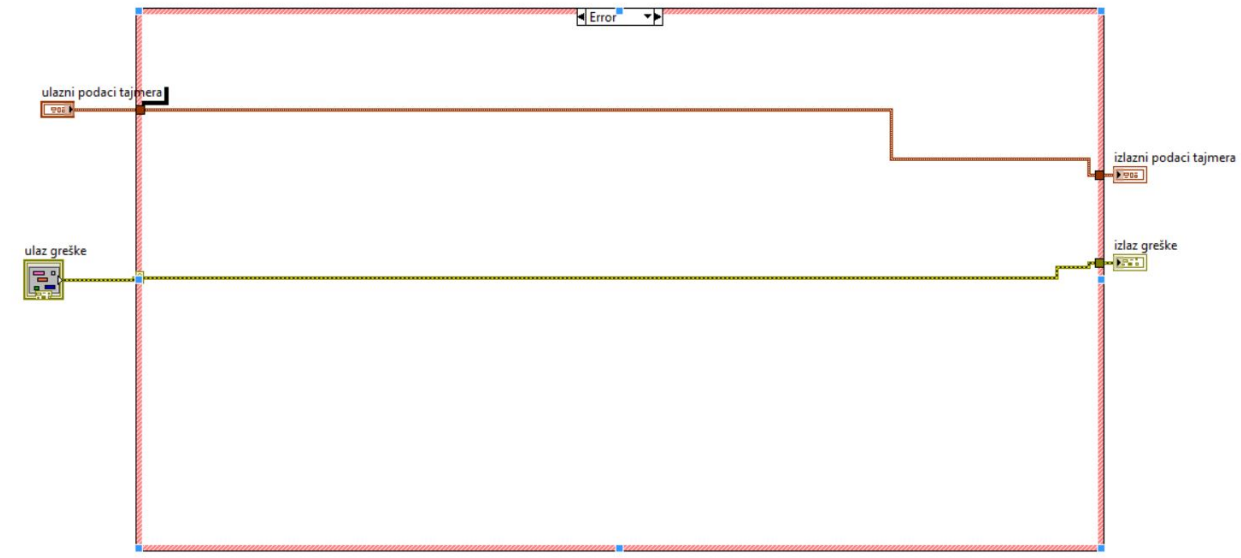
neka greška u programu - taj slučaj je ilustrovan na Slici 52.

Poslednji korak u projektovanju *Tajmera* je omogućiti tajmeru da

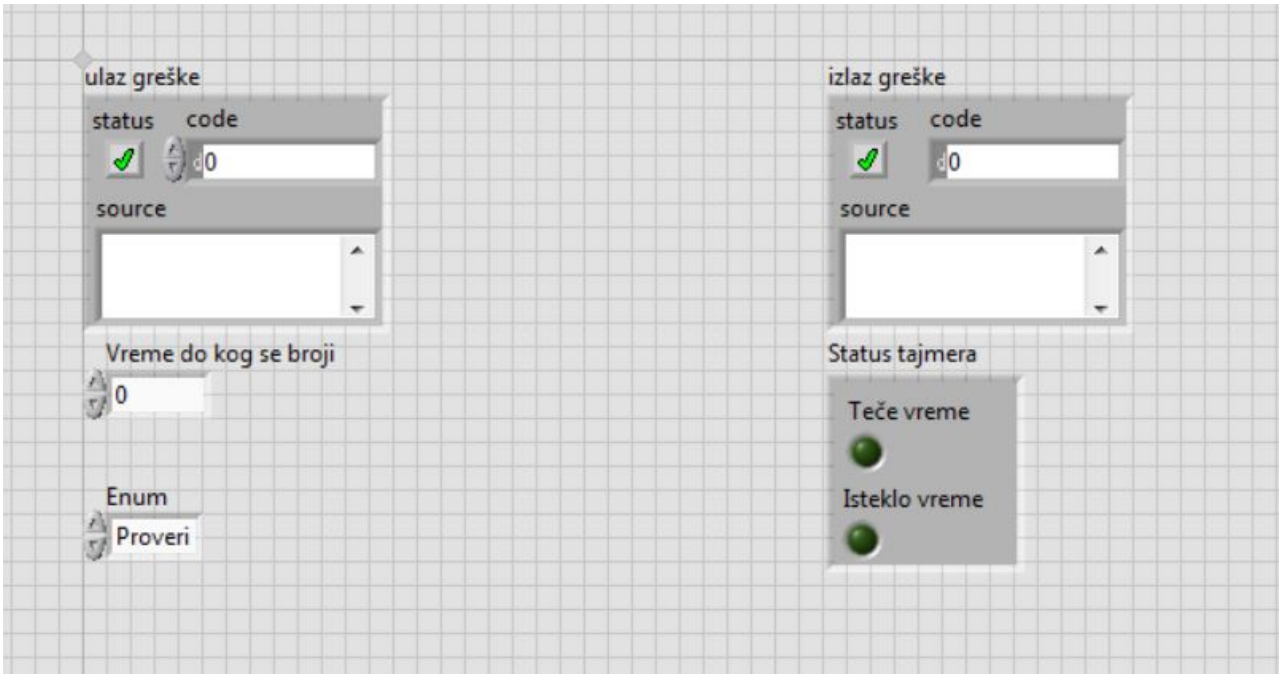


Slika 51: Jezgro tajmera - poziv proveri vreme metode

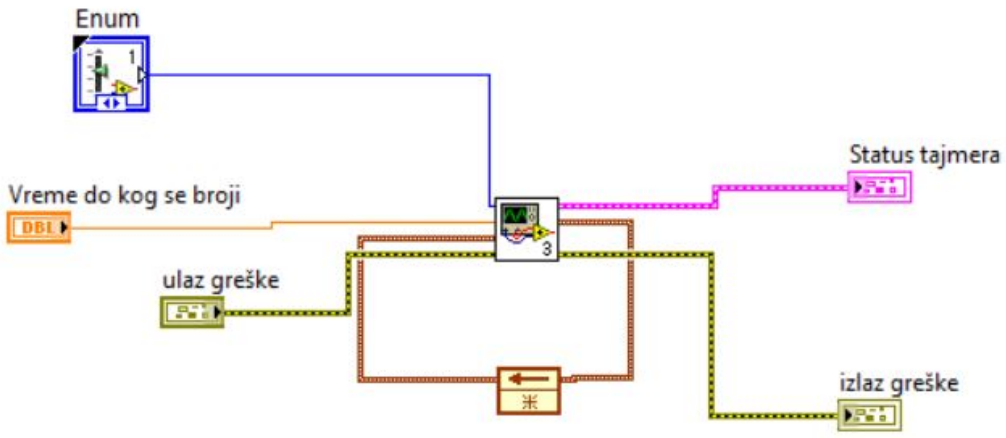
funckioniše kao pogon akcije. Dizajniran je blok dijagram koji poziva funkciju jezgra *Tajmera* i omogućava rad na principu pogona akcije preko *Feedback node*-a. Ovaj blok dijagram je dat na Slici 53. Svi elementi koji su ulazni i izlazni u ovom pogonu akcija su ožičeni tako da budu ulazni i izlazni parametri u funkciji *Tajmera*. Jasno je ovaj pogon akcija ustvari zaista predstavlja funkciju *Tajmer*.



Slika 52: Jezgro tajmera - slučaj kada postoji greška



Slika 53: *Front panel* pogona akcija tajmera



Slika 54: Blok dijagram pogona akcija tajmera

Automati stanja u LabVIEW

Dizajn obrasci (ili softverski dizajn obrasci) u *LabVIEW* su obično jedan od najčešće korišćenih obrazaca dizajna za implementaciju automata stanja.

U *LabView*-u postoje mnogi pristupi automatima stanja, kao što je pristup klasičnog automata stanja [2], [3], događajima vođenim automati stanja, automati stanja sa redovima i još mnogi drugi. Svaki od prethodnih pristupa je vrsta nadogradnje prethodne, na primer, automati stanja vođenim događajima rešavaju neke nedostatke klasičnog automata stanja, i tako dalje [4].

U ovom poglavlju biće obajšnjena struktura klasičnog pristupa automatima stanja kroz primere koji predstavljaju simulaciju rada određenih sistema.

Klasični automati stanja

Arhitektura klasičnog automata stanja organizuje VI u stanja i prelaze. Stanja predstavljaju sposobnosti VI, odnosno šta VI može da uradi, a prelazi mapiraju načine na koje VI može da se kreće iz jednog stanja u drugo.

Izgled blok dijagrama jednog klasičnog automata stanja može se videti na slici 55. Svako stanje odgovara posebnom slučaju (*case*) u glavnoj strukturi slučaja (*main case structure*) i svaki slučaj sadrži sav kod potreban za izvršavanje funkcija njegovog stanja, koji odlučuje koje stanje je sledeće. Stanja se obično navode u enumerisanoj konstanti (*type-defined enumerated constant*) i ova konstanta se čuva u *shift* registru na glavnoj petlji. Glavna struktura slučaja se postavlja unutar glavne *while* petlje i svako stanje mora odlučiti da li se *while* petlja nastavlja ili završava. Najčešće rešenje predstavlja povezivanje logičke konstante, kao izlaz iz glavne strukture slučaja, na uslovni terminal glavne *while* petlje. Na slici 55 prikazani su delovi automata stanja sa njihovim imenima [5] i u nastavku su data njihova objašnjenja:

- Ulazna oblast (*Input area*) - Ova oblast se obično nalazi sa leve strane glavne *while* petlje. Kod koji se nalazi sa leve strane *while* petlje

[2] Boris Jakovljević; Stefana Jocić; Tomislav Novak; Živko Kokolanski; Bodan Velkovski; Dariusz Tefelski; Angelika Tefelska; Milica Janković; Marko Barajaktarović; Kosta Jovanović; Nikola Knežević; Petar Atanasijević; Marija Novičić. *Control, virtual instrumentation and signal processing use cases practicum*. Fakultet tehničkih nauka, Novi Sad, Serbia, 2019

[3] Rick Bitter; Taqi Mohiuddin; Matt Nawrocki. *LabView Advanced Programming Techniques*. CRC Press, Taylor and Francis Group, second edition, 2007

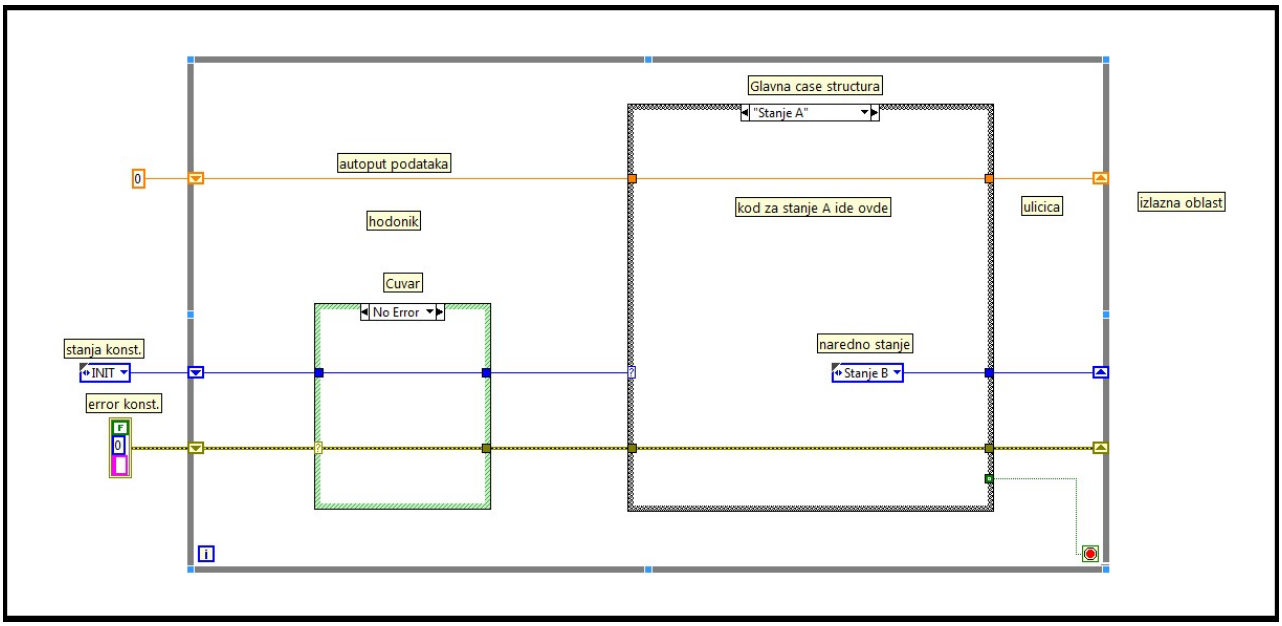
[4] Thomas Bress. *Effective LabView programming*. National Technology and Science, 2013

[5] Thomas Bress. *Effective LabView programming*. National Technology and Science, 2013

i povezan je na nju izvršiće se pre nego što petlja počne sa izvršavanjem. Žice ulaze u *while* petlju preko tunela ili *shift* registra. Ova oblast se najčešće koristi za inicijalizaciju *shift* registara i objekata sa prednjeg panela, postavljanje konstanti koje će se koristiti u petlji i sl.

- Hodnik (*Lobby*) - Obično se nalazi između leve ivice glavne *while* petlje i leve ivice glavne *case* strukture. Najčešće se koristi za proveru postojanja greške, kao lokacija za ulaze koji se koriste u svakoj iteraciji petlje, itd.
- Čuvar (*Guard Clause*) - Njegova uloga je da pošalje automat stanja u stanje isključenja (*shutdown*) u slučaju pojave greške ili ako npr. korisnik pritisne stop dugme. Čuvar predstavlja *case* strukturu na čiji je terminal povezana *error* konstanta, gde se na taj način dobijaju dva moguća slučaja u ovoj strukturi. Jedan slučaj je obojen zelenom bojom i predstavlja *No Error* stanje, dok je drugi, crveno obojen, predstavlja *Error* stanje. Kada nema greške trenutno stanje automata se samo prosleđuje kroz slučaj *No Error* do terminala glavne *case* strukture.
- Autoput podataka (*Data Highway*) - Pod ovim se podrazumeva grupa žica koja se kreće od jednog do drugog kraja *while* petlje.
- Glavna *case* struktura (*Main Case Structure*) - Ova struktura predstavlja srce automata stanja, jer sadrži kod za svako od stanja automata. Svako stanje predstavljeno je posebnim slučajem u ovoj strukturi.
- Glavna *while* petlja (*Main Loop*) - Predstavlja petlju koja vodi automat stanja iz stanja u stanje. Glavna petlja poseduje *shift* registre koji čuvaju podatke sa autoputa podataka. Takođe, postoje i *shift* registri koji prenose informaciju o sledećem stanju i o postojanju i tipu greške.
- Uličica (*alley*) - Nalazi se između desne ivice *case* strukture i desne ivice *while* petlje. Najčešće se koristi kao lokacija za indikatore i logiku povezanu na stop terminal *while* petlje.
- Izlazna oblast (*Output area*) - U ovoj oblasti se obično nalaze izlazi iz glavne petlje automata. Najčešće se nalazi desno od *while* petlje.

Prvi korak prilikom kreiranja automata stanja jeste identifikovati stanja, sve moguće prelaze između stanja i uslove koji uzrokuju prelaze. Kao što je već navedeno stanja se obično navode u enumerisanoj konstanti, koja se može kreirati desnim klikom na projekat i izbor opcije *New Control* na što je prikazano na slici 56. Na taj način dobijamo prozor sličan kao na slici 57. Potrebno je iz padajućeg menija u *toolbar*-u izabrati *Type Def*, zatim postaviti *Enum Constat* na prednji panel



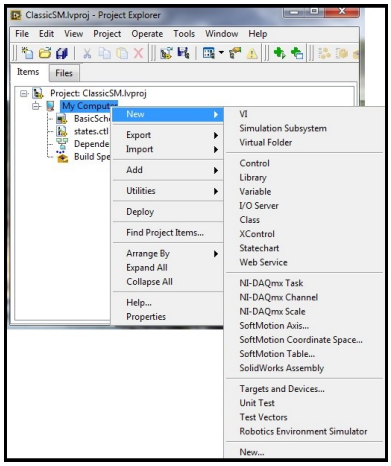
Slika 55: Izgled blok dijagrama jednog klasičnog automata stanja.

i pritiskom desnog klika miša izabrati *Edit Item*, što otvara prozor prikazan na slici 57. Tu je moguće upisati sva stanja automata. Takođe, mogu se videti vrednosti koje *LabVIEW* vezuje za svako stanje, što je objašnjeno u uvodnom poglavlju.

Primer rada drinkomata

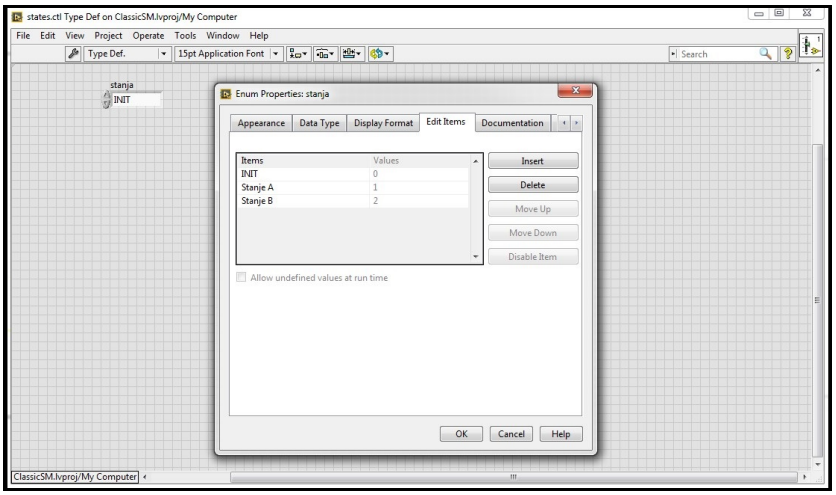
U ovom delu biće prikazan primer rada jednog drinkomata primenom klasičnog automata stanja. Drinkomat izbacuje sok koji košta 20 dinara, gde je za kupovinu soka moguće koristiti apoene od 5 i 10 dinara. Takođe, drinkomat ima mogućnost vraćanja kusura. Postoji inicijalno (početno) stanje drinkomata gde se vrše početna podešavanja, posle kog drinkomat ulazi u stanje iščekivanja apoena.

Postoje dva pristupa za rešavanje ovakvog problema, i oni su redom prikazani na slikama 58 i 59. Jedan pristup je da u automatu stanja imamo između ostalog i stanja koja predstavljaju moguće apoene i cenu soka (*5DIN*, *10DIN* i *20DIN*), koji je prikazan na slici 58. U tom slučaju postoje sledeća stanja: *INIT*, *ČEKANJE*, *5DIN*, *10DIN*, *20DIN*, *KUSUR* i *IZBACI_SOK*. Ovakav pristup predstavlja automat sa minimalnim brojem stanja, gde npr. ukoliko dođe do promene cene soka broj stanja se ne menja, samo uslovi za prelazak između stanja. Drugi pristup (slika 59), tzv. neminimalni automat stanja, predstavlja loš pristup rešavanju problema, jer kao što je prikazano na blok dijagramu sa promenom cene soka menja se i broj stanja automata (otud i ime neminimalni), odnosno broj "između" stanja. Pod "između" sta-

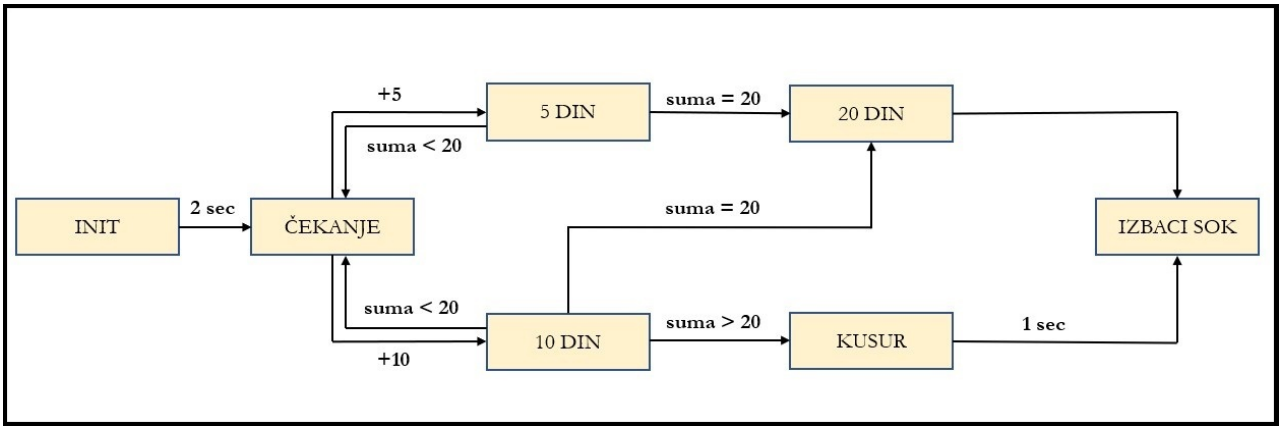


Slika 56: Kreiranje nove kontrole u projektu.

njima podrazumevaju se stanja koja se nalaze između *5DIN*/*10DIN* (mogućih apoena) i *20RSD* (cene soka). Tako u ovom slučaju u odnosu na prethodni imamo jedno stanje više, a to je *15DIN*. Vidimo da ako je cena soka porasla na *25DIN*, postojalo bi još jedno stanje više (upravo tih *25DIN*). Ovakvim pristupom pri svakoj promeni cene soka morali bismo menjati automat stanja, što svakako nije cilj, zato će u nastavku biti objašnjenjo samo rešenje sa minimalnim brojem stanja.



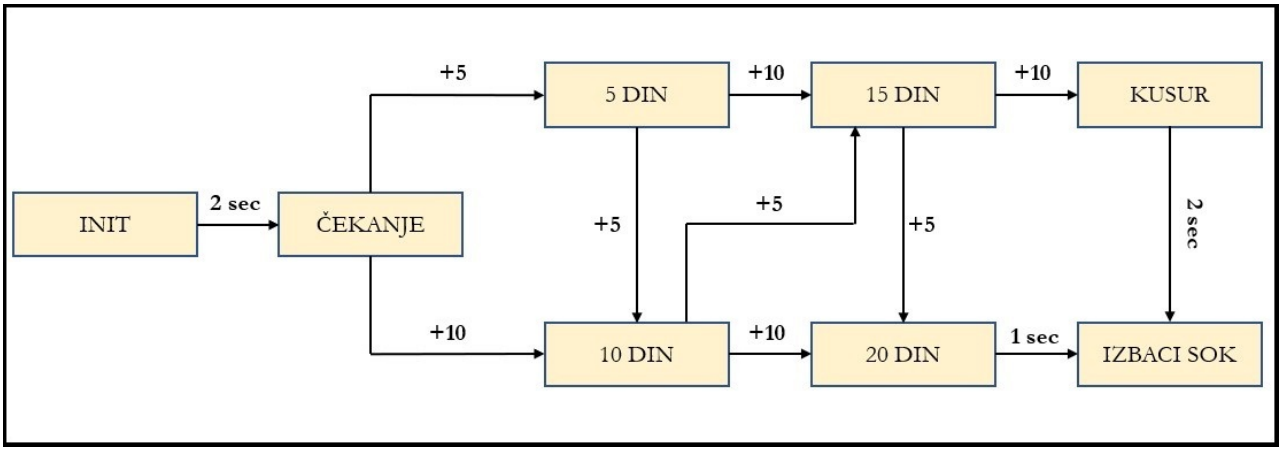
Slika 57: Definisanje stanja.



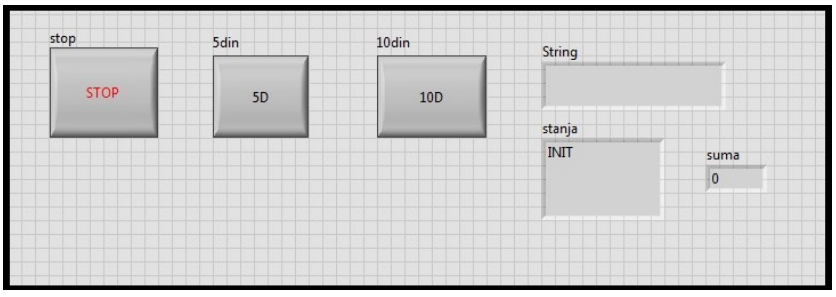
Slika 58: Automat sa minimalnim brojem stanja.

Za simulaciju rada drinkomata potrebno je prvo kreirati novi projekat. Stablo projekta prikazano je na slici 61. Kao što se može videti sa slike za rešavanje problema postoje samo dva fajla, VI nazvan *drinkomatSM.vi*, koji sadrži kod za simulaciju rada drinkomata i kontrola pod nazivom *stanja.ctl*, gde su definisana sva stanja (slika 62).

Izgled prednjeg panela može se videti na slici 60. Tu se nalaze dva dugmeta *5D* i *10D* koja su ekvivalnet ubacivanju apoena od 5 i 10 di-



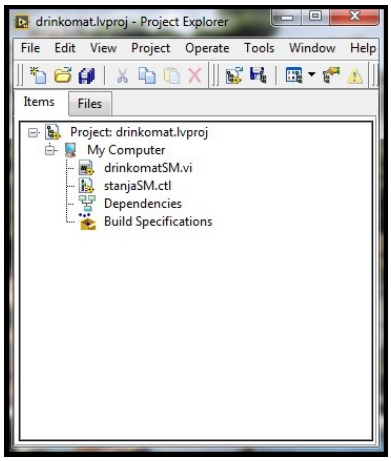
Slika 59: Automat sa neminimalnim brojem stanja.



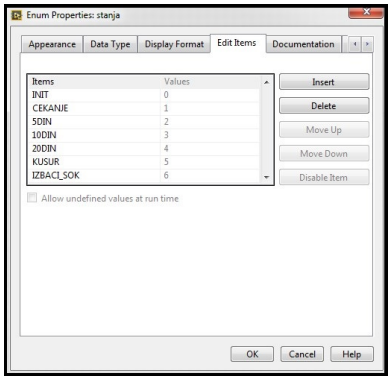
Slika 60: Izgled prednjeg panela automata stanja koji simulira rad drinkomata.

nara u drinkomat respektivno. *Stop* dugme služi za zaustavljanje celog virtuelnog instrumenta. Numerički indikator nazvan *suma* predstavlja sumu novca ubačenu u drinkomat, trenutno stanje drinkomata prikazuje se na indikatoru *stanja* i *String* prikazuje koliki kusur je potrebno vratiti, ukoliko postoji.

Kao što je objašnjeno na početku, blok dijagram ovog automata stanja sastoji se od glavne *while* petlje koja sadrži čuvara i glavnu *case* strukturu, gde je svako stanje dodeljeno posebnom slučaju. *Error* konstanta je povezana sa selektor terminalom čuvara, kreirajući tako dva moguća slučaja - *Error* i *No Error*. U slučaju greške automat ide pravo u stanje inicijalizacije (INIT stanje). Konstanta stanja je povezana sa terminalom glavne *case* strukture, što povlači kreiranje sedam različitih slučajeva koji će biti detaljno opisani u nastavku:



Slika 61: Izgled stabla projekta za simulaciju rada drinkomata.



Slika 62: Spisak svih stanja za simulaciju rada drinkomata.

- *INIT* - Svaki put pri pokretanju programa, VI bi trebao proći kroz ovo stanje kako bi se svaki element postavio na podrazumevanu vrednost. Kod koji se izvršava u ovom stanju može se videti na slici 63. Nakon dve sekunde automat prelazi u sledeće stanje, *ČEKANJE*.

- *ČEKANJE* - U ovom stanju automat čeka akciju od korisnika. Dva ulaza su povezana na glavnu *case* strukturu, dugmad *5D* i *10D*. Ova dugamad se mora nalaziti u hodniku automata stanja, jer u suprotnom njihova vrednost (promena stanja) neće biti uhvaćena dok drinkomat radi. VI proverava da li su dugmad pritisnuta, i to pomoću dve *case* strukture. Dugme *5D* je povezano sa terminalom prve *case* strukture, što znači ukoliko je dugme stvarno pritisnuto, drinkomat će preći u sledeće stanje (*5DIN*). U suprotnom se proverava da li dugme *10D* pritisnuto (koje je povezano na terminal druge *case* strukture) i ukoliko je odgovor potvrdan automat prelazi u stanje *10DIN*. Ukoliko ništa od navedenog nije ispunjeno drinkomat ostaje u stanju čekanja. Blok dijagram koji delimično prikazuje opisano stanje može se videti na slici 64.

- *5DIN* - Svaki put kada je dugme *5D* pritisnuto, broj 5 se doda na promenljivu *suma*. Nakon toga proverava se vrednost sume, i ukoliko je ta vrednost manja od 20, automat i dalje ostaje u stanju čekanja, jer nema dovoljno novca za izbacivanje soka. U slučaju da je vrednost sume 20, onda drinkomat prelazi u stanje *20DIN*. Blok dijagram koji delimično prikazuje blok dijagram upravo opisanog stanja može se videti na slici 65.

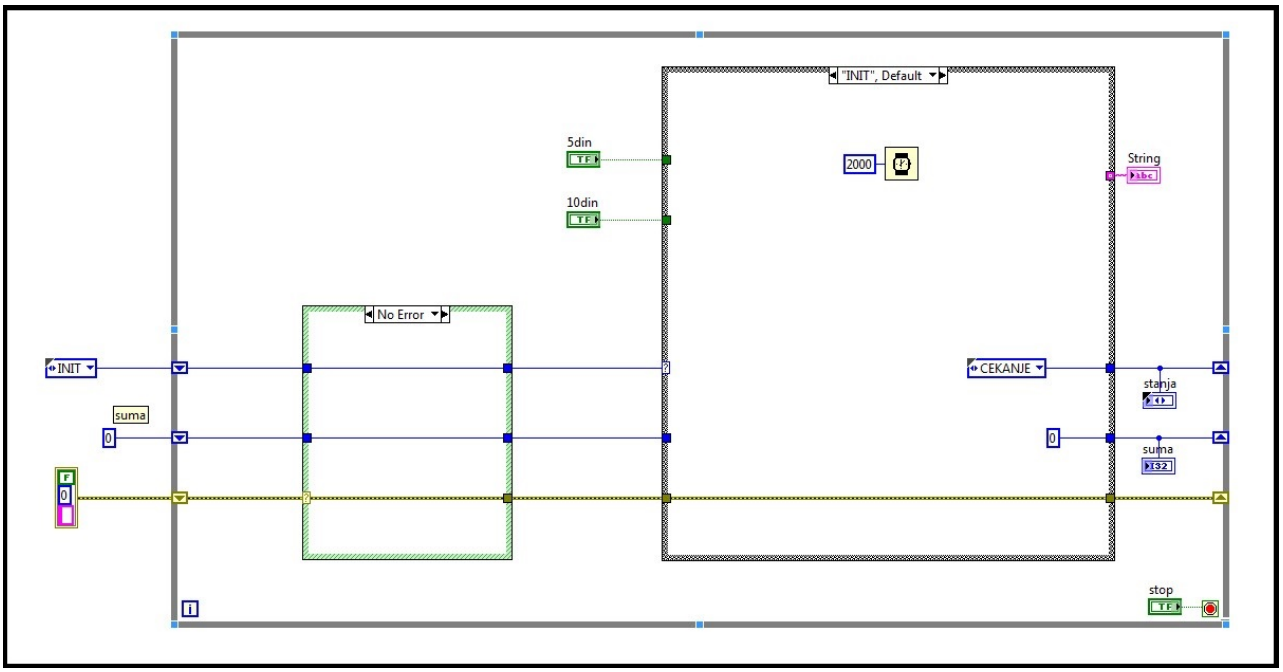
- *10DIN* - Iz ovog stanja automat može otići ili u stanje *20DIN* ili u *KUSUR*. Slično kao kod stanja *5DIN*, i ovde svaki put kada korisnik pritisne dugme *10D*, vrednost 10 se dodaje na trenutnu vrednost sume. Ukoliko je vrednost sume 10 i korisik pritisne dugme *10D*, to znači da postoji dovoljno novca za kupovinu soka i drinkomat

prelazi u sledeće stanje, *20DIN*. Međutim, ukoliko je trenutna vrednost sume 15 (što se može postići stiskanje dugmeta *5D* tri puta ili kombinacijom dugmadi *5D* i *10D*) i korisnik pritisne dugme *10D*, onda je korisniku potebno vratiti kusur, pa automat prelazi u stanje *KUSUR* (što je u ovom slučaju 5 dinara). Prelazak u stanje *KUSUR* moguće je samo iz ovog stanja. Blok dijagram koji delimično prikazuje blok dijagram upravo opisanog stanja može se videti na slici 66.

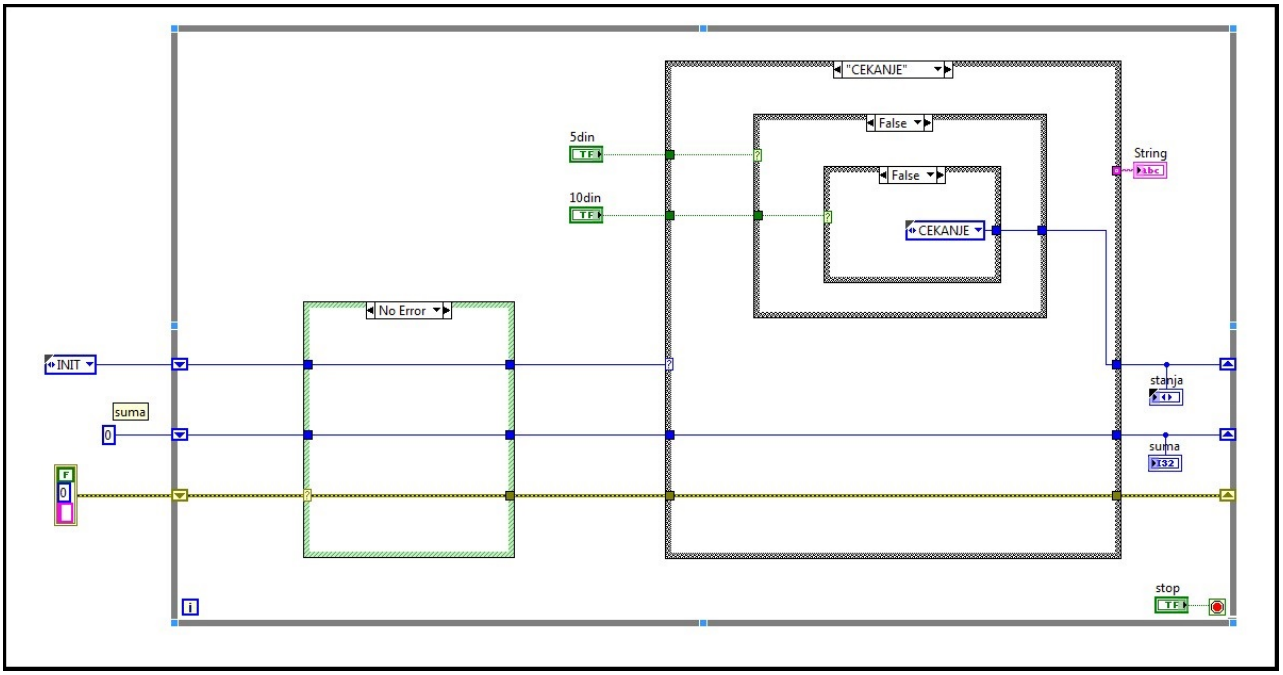
- *20DIN* - Ukoliko automat dođe do ovog stanja to znači da je vrednost sume 20 i ima dovoljno novaca u automatu da pređe u sledeće stanje *IZBACI_SOK*. Drinkomat u ovom stanju ostaje samo dve sekunde, dovoljno da se prikaže na indikatoru na prednjem panelu u kom je trenutno stanju automat. Nakon što prođe jedna sekunda, automat prelazi u sledeće stanje *IZBACI_SOK*. Kod koji izvršava upravo opisano stanje prikazan je na slici 67.

- *KUSUR* - U ovom stanju ispisuje se samo koliki je kusur potrebno vratiti (što se ispisuje u *String* indikatoru na prednjem panelu) i nakon toga drinkomat prelazi u stanje *IZBACI_SOK*. Blok dijagram ovog stanja može se videti na slici 68.

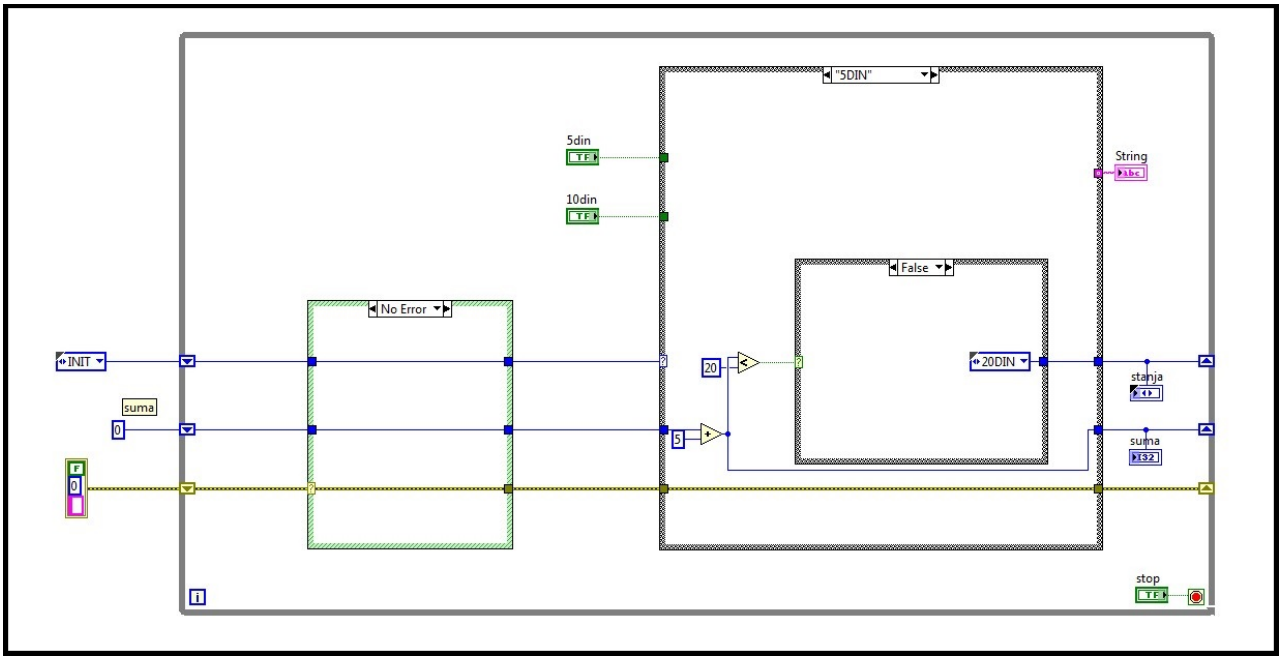
- *IZBACI_SOK* - Kada drinkomat izbaci sok, prelazi u stanje *INIT*, gde se priprema za sledeće usluživanje korisnika. Slično kao i u prethodnim stanjima, i u ovom stanju se automat zadržava samo dve sekunde kako bi se na ekranu ispisalo trenutno stanje drinkomata. Blok dijagram upravo opisanog stanja nalazi se na slici 69.



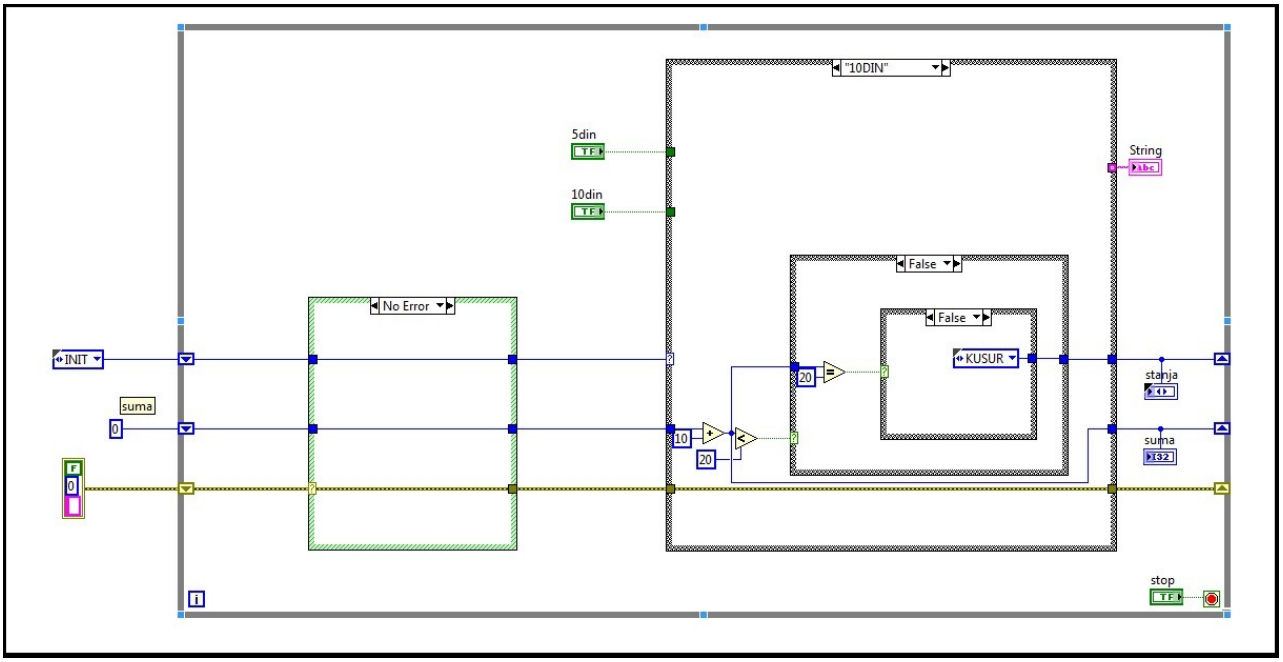
Slika 63: Blok dijagram *INIT* stanja drinkomata.



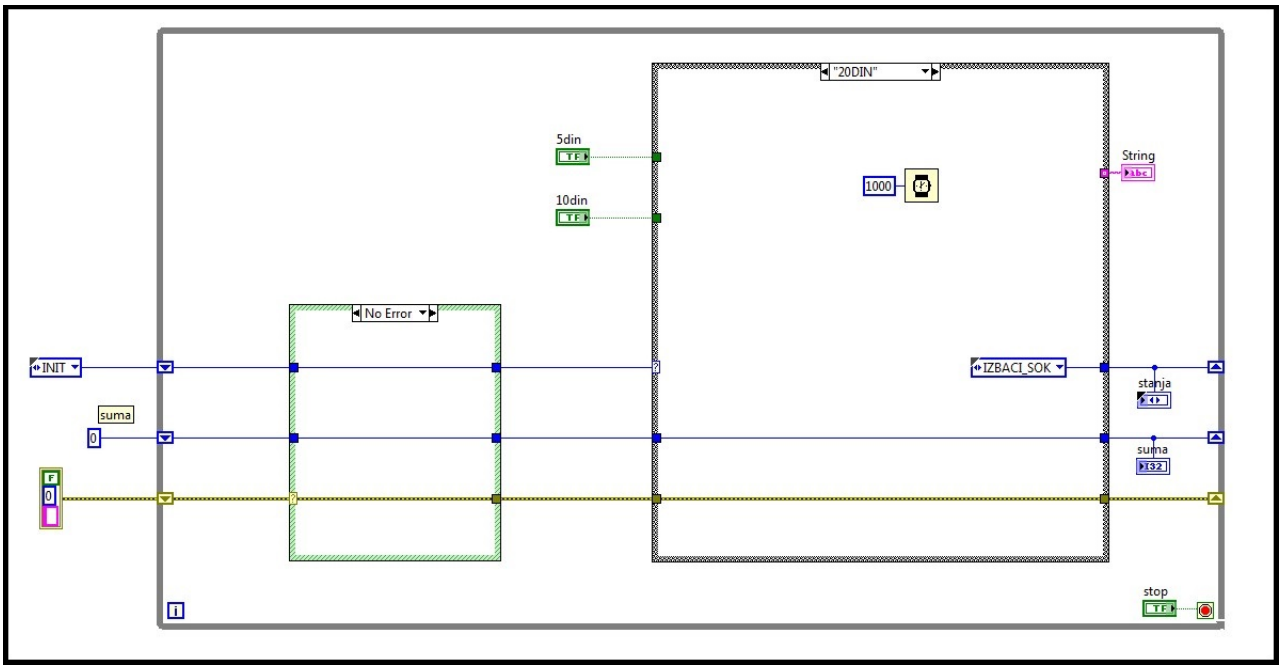
Slika 64: Delimični prikaz blok dijagrama stanja *ČEKANJE*.



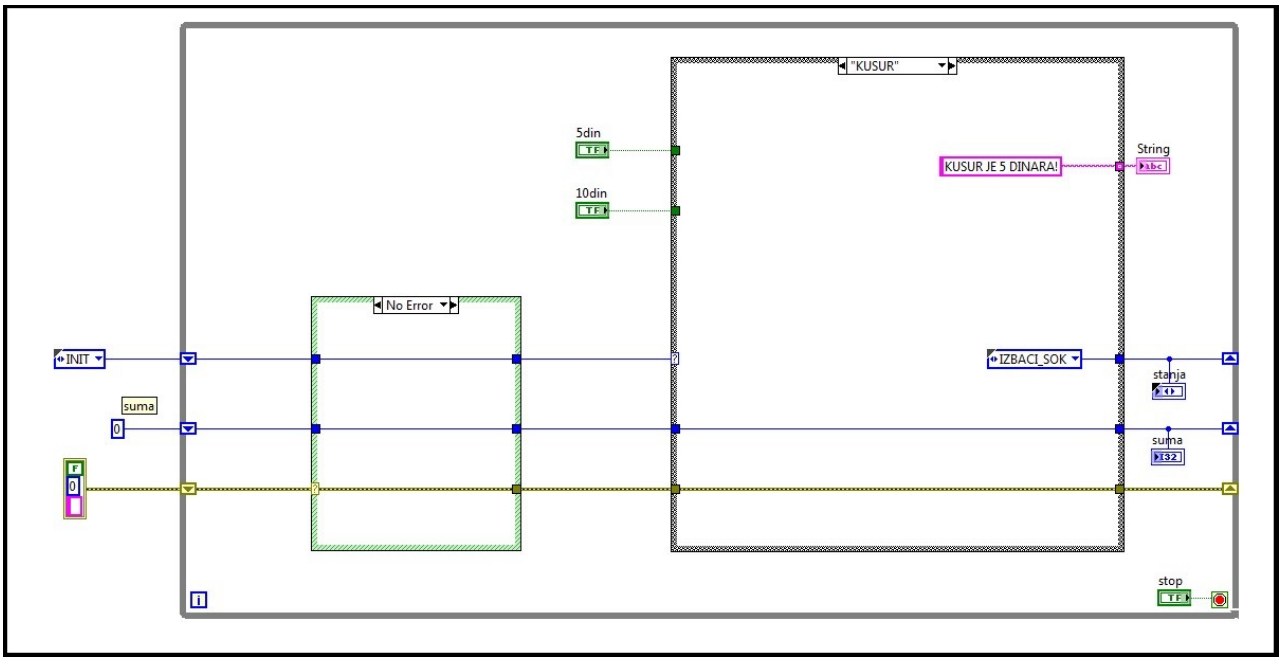
Slika 65: Delimični prikaz blok dijagrama stanja *5DIN*.



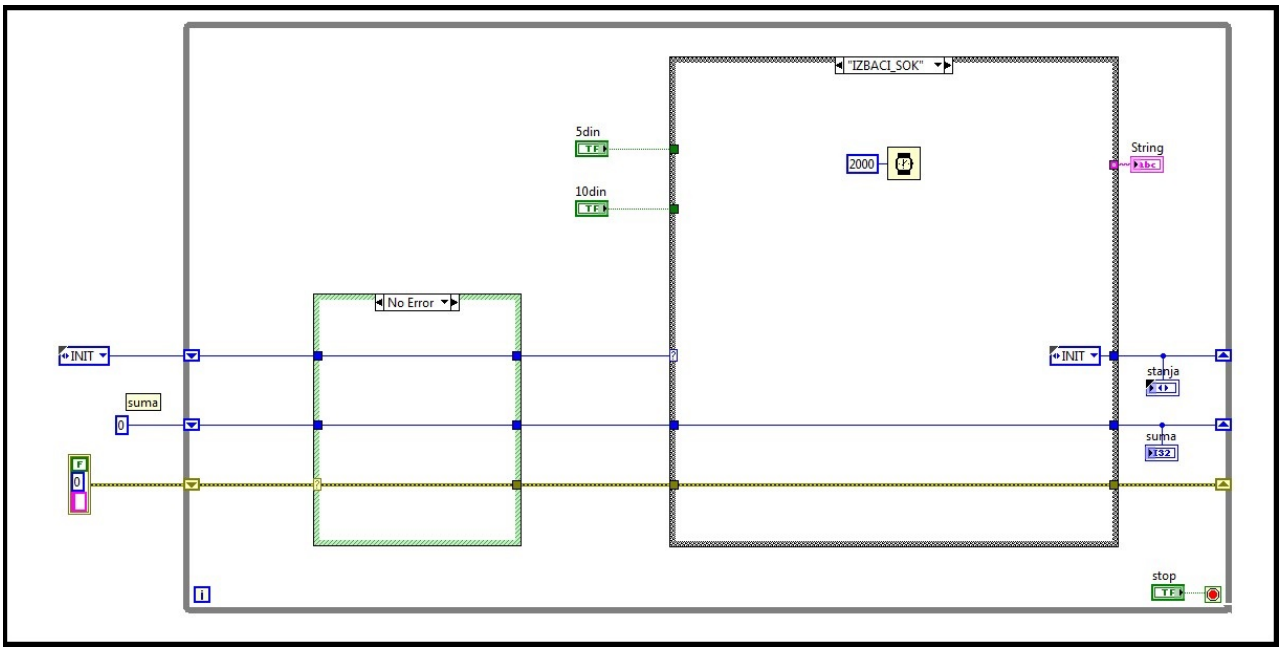
Slika 66: Delimični prikaz blok dijagrama stanja *10DIN*.



Slika 67: Blok dijagram stanja *20DIN*.



Slika 68: Blok dijagram stanja *KUSUR*.



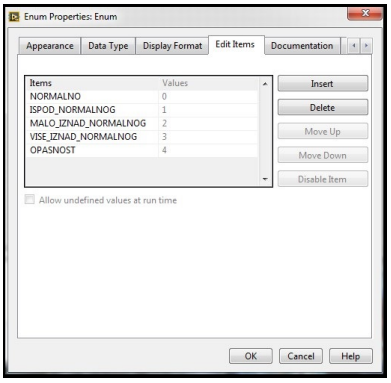
Slika 69: Blok dijagram stanja *IZBACI_SOK*.

Primer rada veštačkog pankreasa

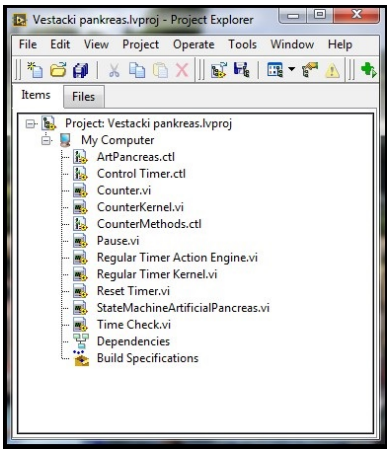
Gušterača, poznata i pod nazivom pankreas, je veoma važna žlezda u sistemu organa za varenje. Dugačka je otprilike 15 cm, duguljastog oblika i pljosnata. Pankreas obavlja dvostruku ulogu: izlučuje sokove potrebne za preradu hrane u crevima, koji se mešaju sa žuči iz jetre i luči hormone u krv, koji deluju u drugim delovima tela, što predstavlja endokrinu funkciju gušterače.

U ovom delu biće prikazana simualcija rada veštačkog pankreasa pomoću klasičnog automata stanja. Sistem treba da funkcioniše na sledeći način:

U svakom trenutku treba proveravati nivo šećera u krvi. Ako je nivo šećera u granicama 3,5-6,1 mmol/l sistem treba da ostane pripravan i da ne radi ništa. Ukoliko vrednost šećera poraste iznad granice od 6,1 mmol/l, a manja je od 10 mmol/l, treba dodavati insulin uključivanjem insulinske pumpe na 4 sekunde. Po isteku ove 4 sekunde potrebno je opet proveriti vrednost šećera, pa postupak ponoviti 2 puta ako se ne može vratiti nivo šećera u normalu. Ako je vrednost veća od 10 mmol/l treba dodavati insulin uključivanjem insulinske pumpe na 5 sekundi. Po isteku ovih 5 sekundi potrebno je opet proveriti vrednost šećera, pa postupak ponoviti 3 puta ako se ne može vrati nivo šećera u normalu. Ukoliko se posle ovih postupaka šećer ne vrati u normalu, oglašava se alarm da se pacijent obrati lekaru. Isto se dešava i ako nivo šećera padne ispod 3,5 mmol/l. Ukoliko se nivo šećera vrati u



Slika 70: Spisak mogućih stanja sistema.



Slika 71: Izgled projektnog stabla zadatka.

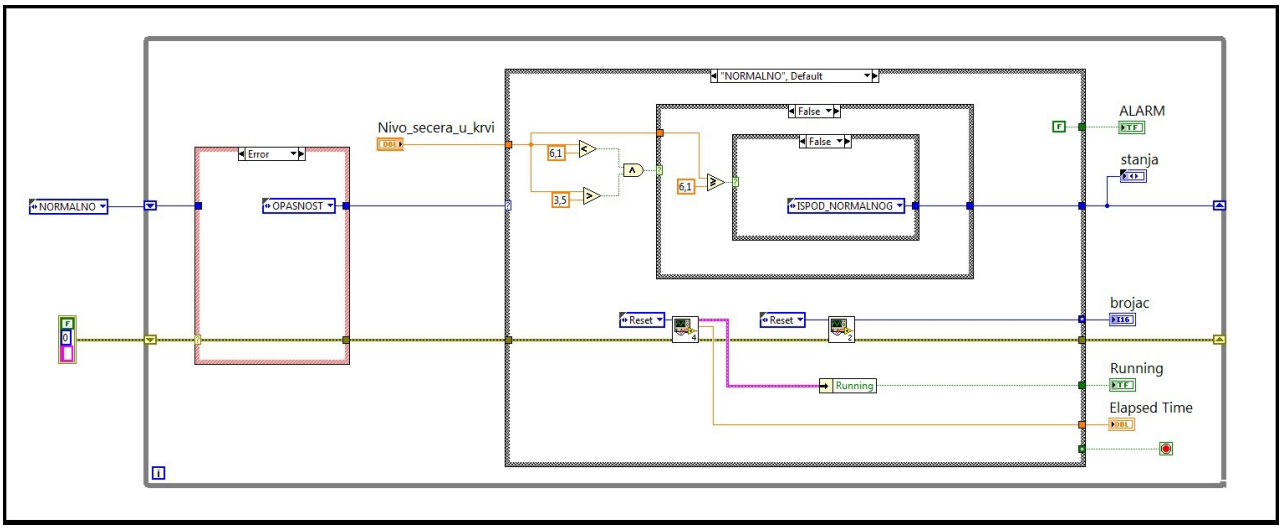
normalu, sistem nastavlja normalno sa radom. Ako se pojavi greška u sistemu potrebno je preći u stanje *OPASNOST* i izaći iz sistema.

Za rešavanje zadatka biće korišćeni brojač i tajmer objašnjenim u prethodnim poglavljima, zato je prilikom kreiranja projekta potrebno u sam projekat uključiti sve potrebne virtuelne instrumente kreirane u prethodna dva poglavalja (slika 71).

Na samom početku potrebno je prvo definisati sva moguća stanja sistema i prelaze između njih. Iz samog teksta zadatka može se zaključiti koja su moguća stanja sistema, ona se mogu videti na slici 70 i svako od njih biće objašnjeno u nastavku.

- *NORMALNO* stanje - Ovo bi trebalo da bude podrazumevano stanje rada pankreasa, što znači da je vrednost šećera u granicama od 3,5 do 6,1 mmol/l. Sve dok je nivo šećera u zadatim granicama sistem treba da ostane u ovom stanju. Ukoliko je vrednost van granica, potrebno je da automat ode u određeno stanje, to znači da se mora proveriti koja je vrednost šećera u krvi. Jednostavan logički izraz doveden je na terminal *case* strukture, koji utvrđuje da li je nivo šećera u zadatim granicama (da li je vrednost kontrole *Nivo_šećera_u_krvi* veća od 3,5 mmol/l i istovremeno manja od 6,1 mmol/l). U slučaju da je uslov ispunjen automat ostaje u istom stanju, na ekranu se ispisuje trenutno stanje, *alarm* je isključen (upisana je logička konstanta vrednosti *False*), tajmer i brojač su resetovani. U suprotnom, proveravamo da li je vrednost šećera u krvi veća od 6,1 mmol/l. Ukoliko je to slučaj, potrebno je da sistem pređe u sledeće stanje *MALO_IZNAD_NORMALNOG*, da se ispita kolika je tačno vrednost šećera. Međutim, ako je ta vrednost manja od 6,1 mmol/l, a nije ni između datih granica, onda sistem treba da pređe u stanje *ISPOD_NORMALNOG*, jer je vrednost ispod 3,5 mmol/l. Blok dijagram koji odgovara ovom stanju može se videti na slici 72.

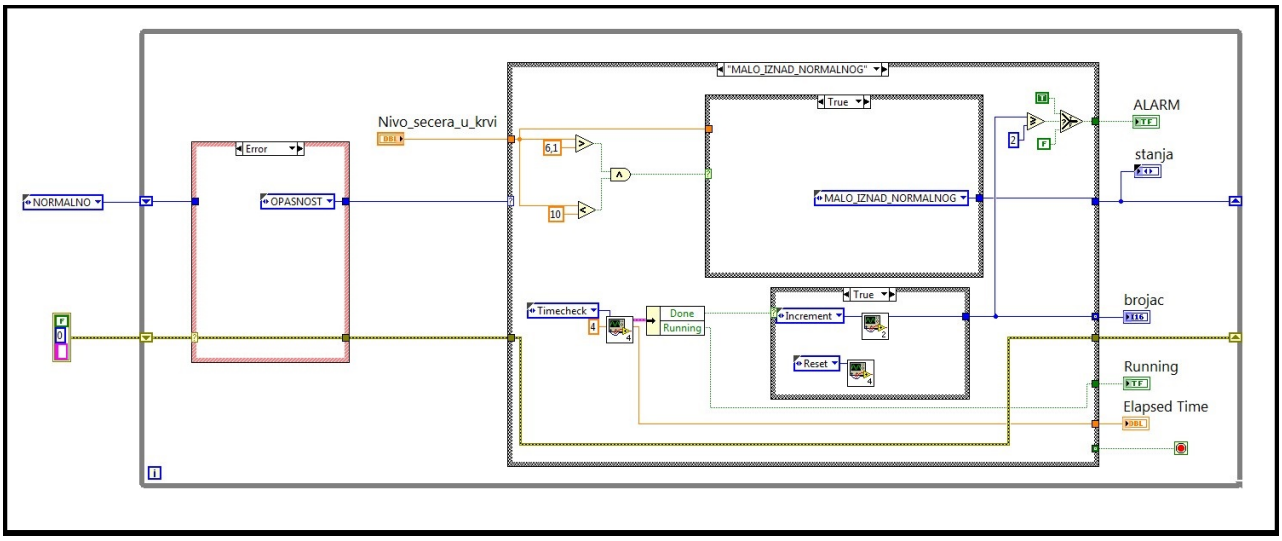
- *MALO_IZNAD_NORMALNOG* stanje - U ovom stanju vrednost nivoa šećera u krvi bi trebala da je u granicama od 6,1 mmol/l do 10 mmol/l. Ukoliko je to slučaj potrebno je da insulinska pumpa radi 4 sekunde i ukoliko nakon isteka te 4 sekunde nivo šećera je i dalje u tom opsegu, potrebno je još jednom uključiti pumpu da radi 4 sekunde. Nakon što je pumpa radila dva puta po 4 sekunde i nivo šećera se nije vratio u normalu potrebno je aktivirati alarm da se pacijent obrati lekaru. Dakle, slično kao i u prethodnom stanju, jednostavan logički izraz je povezan na terminal *case* strukture , gde se utvrđuje da li je vrednost kontrole *Nivo_šećera_u_krvi* u zadatim granica. Ukoliko jeste, potrebno je uključiti pumpu da radi 4 sekunde (indikator *Running* svetli zeleno). To vreme se meri pomoću tajmera kreiranog iz prethodog poglavlja, tako da je potrebno izabrati stanje *Timecheck*, uneti broj sekundi koji je potreban da tajmer izmeri i pro-



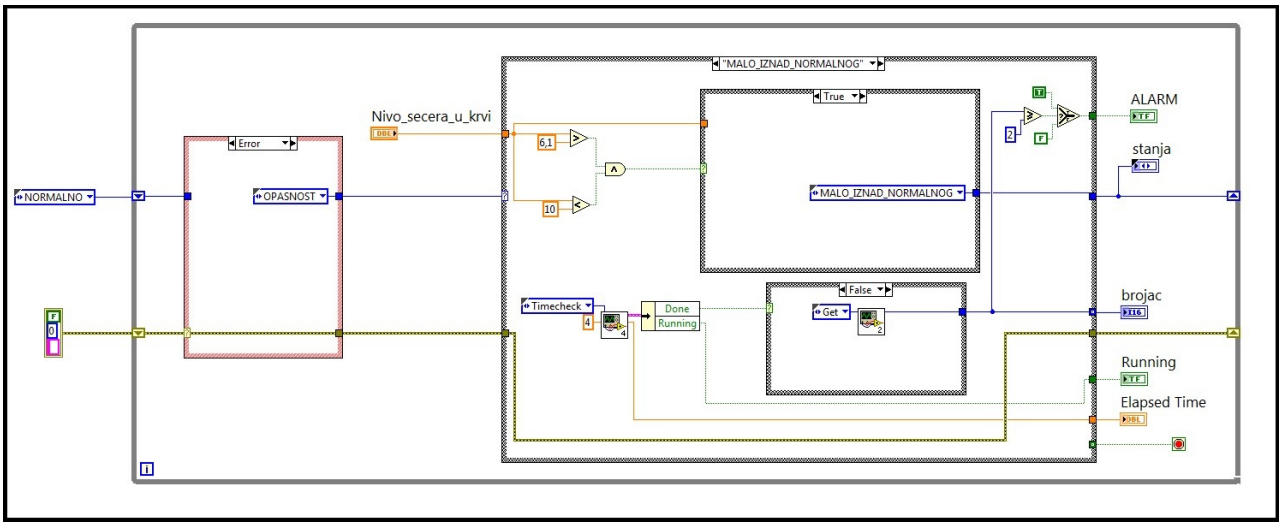
Slika 72: Blok dijagram normalnog stanja veštačkog pankreasa.

slediti informaciju o stanju tajmera, odnosno da li je izbrojao zadato vreme (*Done*) ili još uvek broji (*Running*). Kada je tajmer završio sa merenjem zadatog vremena, logički signal se šalje na terminal još jedne *case* strukture, u kojoj u slučaju *True* potrebno uvećati vrednost brojača za jedan (prethodna vrednost je bila 0, resetovan brojač u prethodnom stanju) i da se resetuje tajmer, da ukoliko je potrebno može ponovo da izmeri traženo vreme. U slučaju da tajmer još uvek meri vreme (*False* slučaj) potrebno je samo pokupiti vrednost brojača (stanje *Get*) i proslediti na izlaz *case* strukture. Potom je potrebno da se vrednost brojača uporedi sa brojem dva i ukoliko je insulinska pumpa bila uključena već dva puta, potrebno je aktivirati alarm, što je urađeno pomoću funkcije *Select*, koja prima kao ulaze logički uslov i dve vrednost povezane na *true* i *false* ulaze i kao izlaz vraća onu odgovarajuću vrednost u zavisnosti od rezultata logičkog izraza. Blok dijagram koji opisuje ovo stanje može se videti na slikama 73 i 74.

- *VIŠE_IZNAD_NORMALNOG* stanje - Ukoliko je vrednost šećera u krvi iznad 10 mmol/l, sistem se nalazi u stanju *VIŠE_IZNAD_NORMALNOG* i potrebno je sada uključiti insulinsku pumpu da radi 5 sekundi i maksimalno 3 puta pre nego što se *alarm* uključi, ako se vrednost šećera u krvi nije smanjila. Slično kao u prethodnim stanjima na terminal *case* strukture doveden je rezultat logičkog izraza koji proverava da li je vrednost šećera u krvi veća od 10 mmol/l. Ukoliko jeste, potrebno je podesiti tajmer da meri 5 sekundi i da brojač svaki put kada tajmer izbroji zadato vreme se uveća za jedan (slično kao i u prethodnom slučaju). Ukoliko brojač ima vrednost 3 ili više potrebno je uključiti alarm da se pacijent javi lekaru. U slučaju *False*

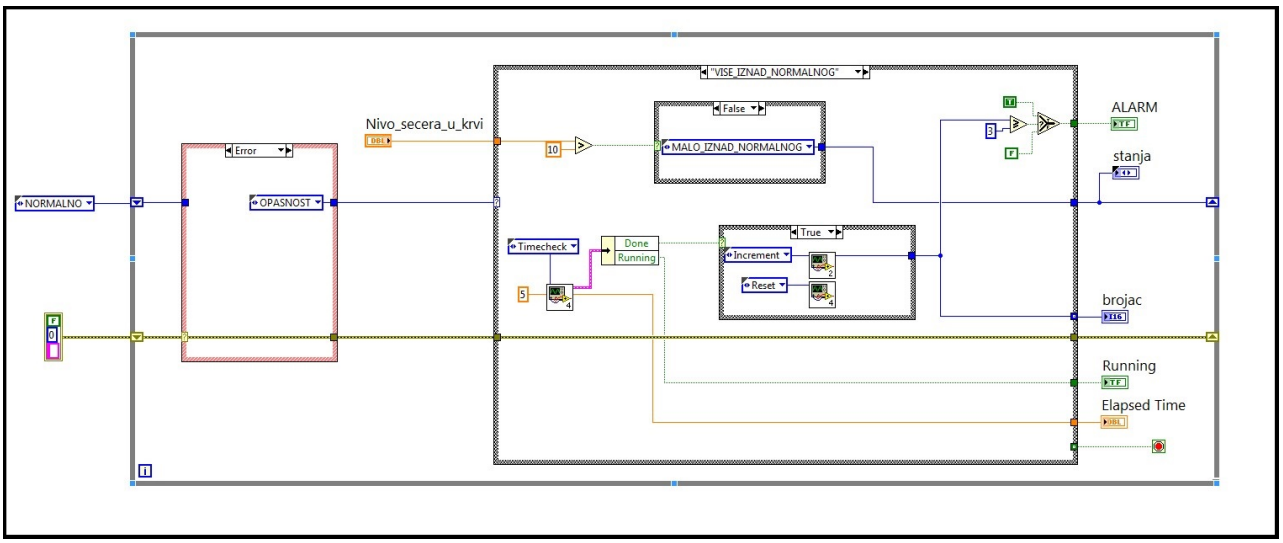


Slika 73: Izgled blok dijagrama stanja *MALO_IZNAD_NORMALNOG*.



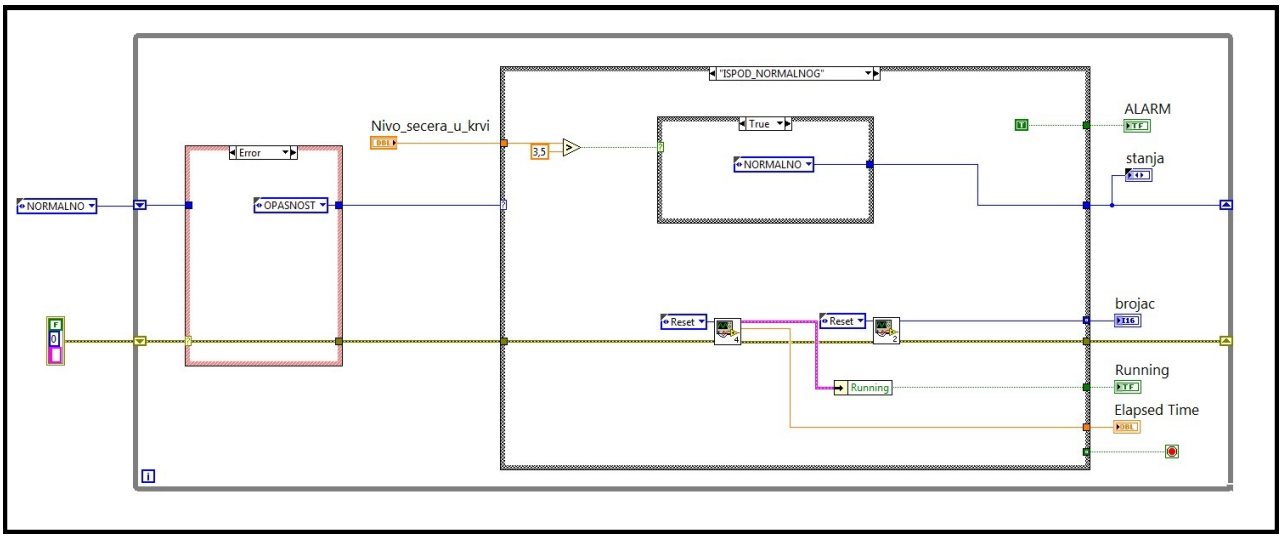
Slika 74: Izgled blok dijagrama stanja *MALO_IZNAD_NORMALNOG*.

(vrednost šećera u krvi nije veća od 10 mmol/l) automat se vraća u prethodno stanja da ustanovi tačnu vrednost šećera.



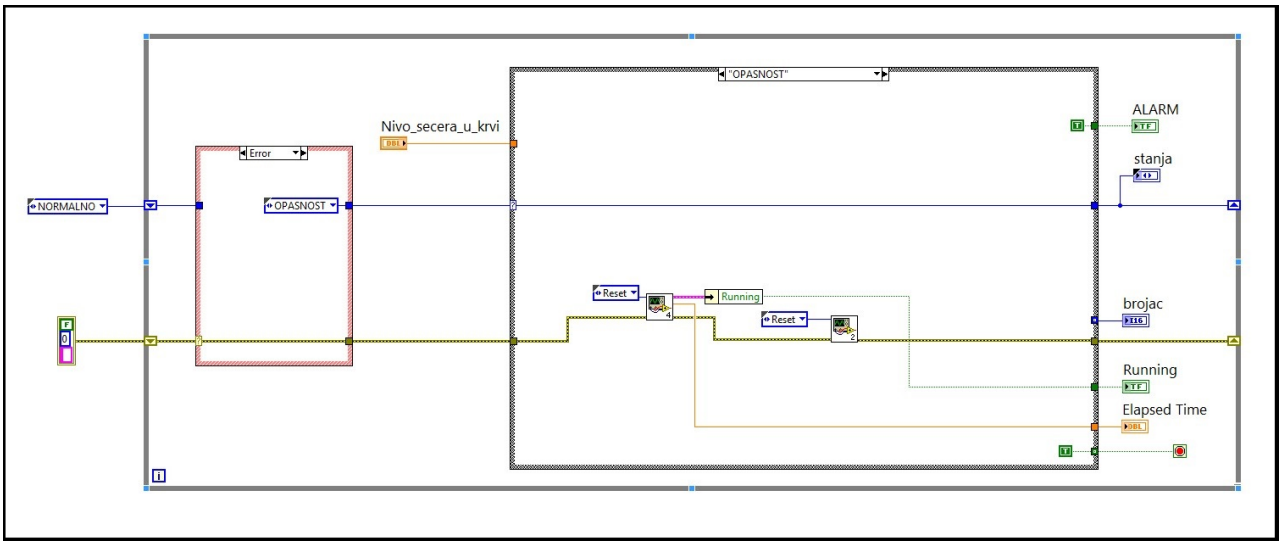
Slika 75: Blok dijagram stanja *VIŠE_IZNAD_NORMALNOG*.

- *ISPOD_NORMALNOG* stanje - U ovom stanju se sistem nalazi ukoliko je nivo šećera u krvi manji od 3,5 mmol/l, gde je potrebno aktivirati alarm da se pacijent javi lekaru, a tajmer i brojač su resetovani. Ukoliko je vrednost šećera u krvi veća pak od 3,5 mmol/l potrebno je da sistem ode u stanje *NORMALNO* kako bi se utvrdila tačna vrednost i sprovela određena akcija. Blok dijagram koji odgovara ovom stanju prikazan je na slici 76.



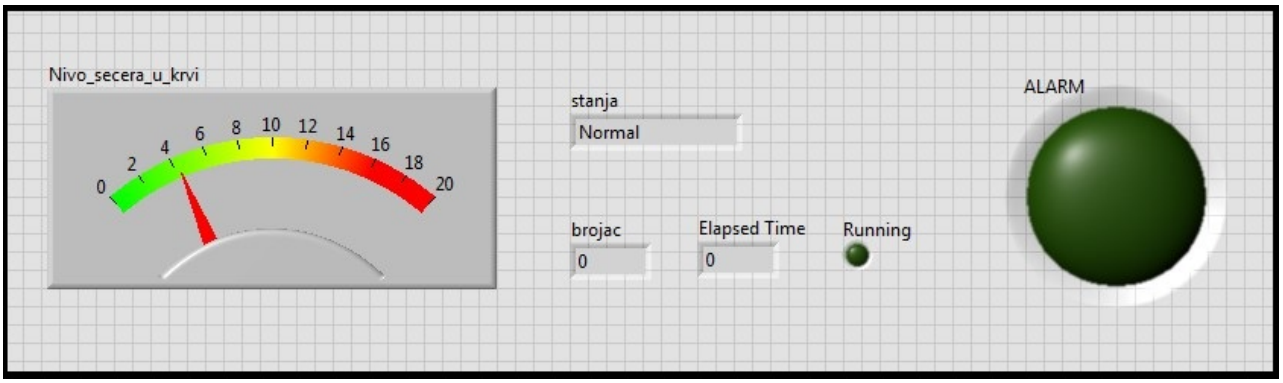
Slika 76: Blok dijagram stanja *ISPOD_NORMALNOG*.

- *OPASNOST* stanje - Ukoliko se javi bilo kakava greška u sistemu sistem treba da ode u ovo stanje, aktivira *alarm* i zaustavi čitav sistem. Blok dijagram koji odgovara ovom stanju prikazan je na slici 77.

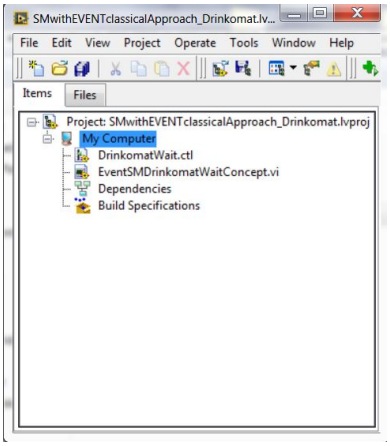


Slika 77: Blok dijagram stanja *OPASNOST*.

Izgled prednjeg panela autoamata stanja može se videti na narednoj slici.



Automati stanja sa Event strukturom - klasičan pristup



Slika 78: "Drvo"projekta automata stanja drinkomata sa *EVENT* strukturom

U ovom poglavlju ćemo objasniti pristup automatima stanja kroz korišćenje *Event* strukture za njihovu realizaciju.

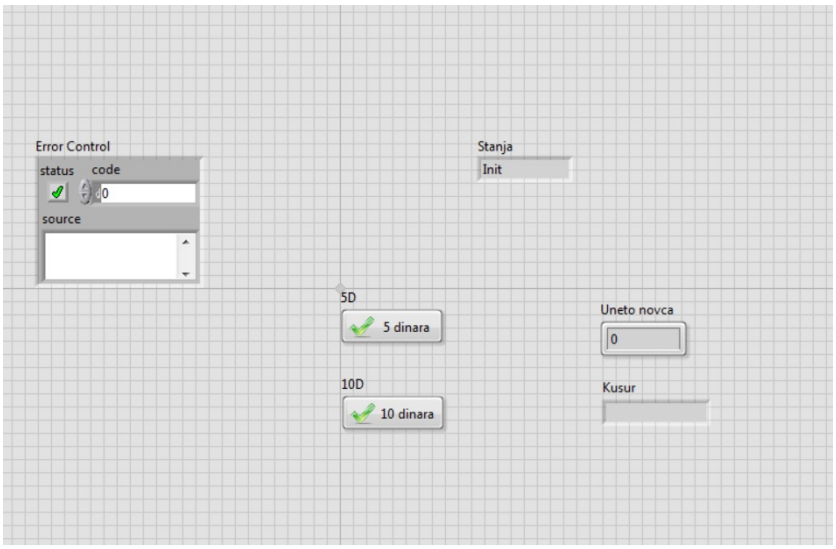
Klasični automati stanja koji su bili relizovani u prethodnom poglavlju su se pokazali kao veoma dobri mehanizmi za rešavanje kompleksnih problema na jednostavan način.

Automati stanja mogu biti realizovani na taj način da omogućavaju izvršavanje određene sekvence funkcija uvek po istom redu. Takvi automati stanja su proceduralno vođeni automati stanja, međutim treba voditi računa i o slučaju kada je pokretanje funkcija prouzrokovano raznim događajima (*event*-ovima).

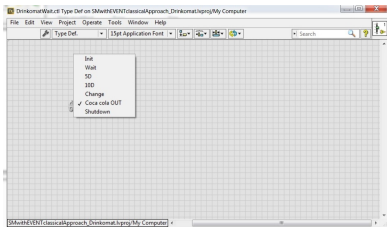
Uopšte, vrste događaja koje mogu prouzrokovati neku promenu stanja mogu biti:

1. Korisnički događaji - događaji koji su uzrokovani promenama na korisničkom panelu,
2. Vremenski događaji - događaji koji su uzrokovani manipulacijom tajmerima,
3. Događaji podataka - događaji koji su uzrokovani nekim podacima, najčešće rezultatom njihove analize.

Primeri ovakvih događaja biće detaljnije objašnjeni u nastavku ovog poglavlja.



Slika 79: *Front panel* automata stanja drinkomata sa *Event* strukturom



Slika 80: Kontrola sa stanjima drinkomata

Klasični automati stanja imaju svoje mane. U ovoj knjizi najuočljivija mana kod klasičnih automata stanja je to što klasični automat stanja u svakoj iteraciji proverava stanje svih kontrola na *Front panel*-u. Ukoliko je broj kontrola veliki, ovo može prouzrokovati bespotrebno opterećenje procesora, kao i kočenje *Front panela*. Naravno, može se izvući analogija i u smislu vremenskih događaja, kao i događaja podataka. Stoga, pristup preko mašina stanja sa *Event* strukturom može umnogome da olakša realizaciju LabView koda.

Osnova pristupa primene automata stanja sa *Event* sturkturom je u tome da se realizacija automata stanja izvede u nekom stanju čekanja na promenu vrednosti kontrola.

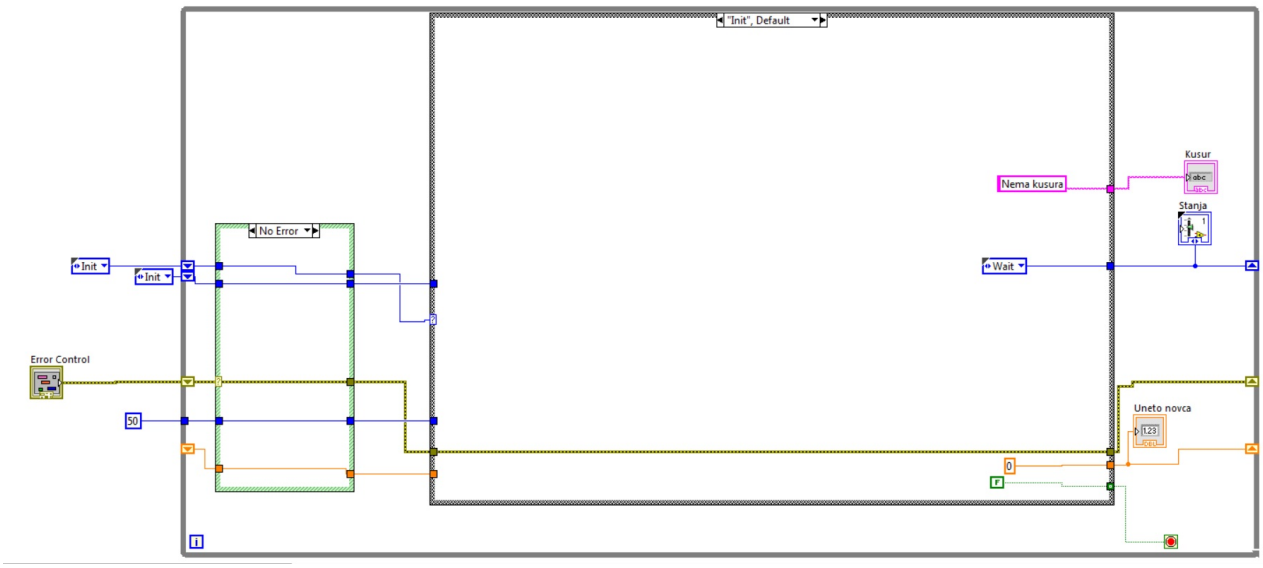
Automati stanja mogu biti realizovani na više načina. Izbor realizacije stanja zavisi od kompleksnoti problema i veština inženjera.

Za realizaciju automata stanja sa *Event* strukturama koriste se kao što samo ima kaže, *Event* strukture. Ovakva struktura može da reaguje na više tipova događaja (*Event*-ova). Postoje dinamički i statički događaji. Statički događaji se odnose na promene *Front panel*-a. Mogu se podeliti na događaje obaveštenja (*Notify*) i filtrirane (*Filer*) događaje. Najčešće korišćeni su statički događaji obaveštenja i oni reaguju direktno na promene na *Front panelu*-u, odnosno obrađuju se pošto deluju na *Front panel*-u. Sa druge strane filtrirani događaji hvataju događaje od strane korisnika pre nego što *LabView* obradi taj događaj, odnosno filtrirani događaj može da se predupredi pre nego što ima efekat na *Front panel*.

Pored ovih događaja postoje još i dinamički događaji koji omogućuju da *Event* struktura reaguje na neki događaj uzrokovan kodom bilo gde na Blok dijagramu.

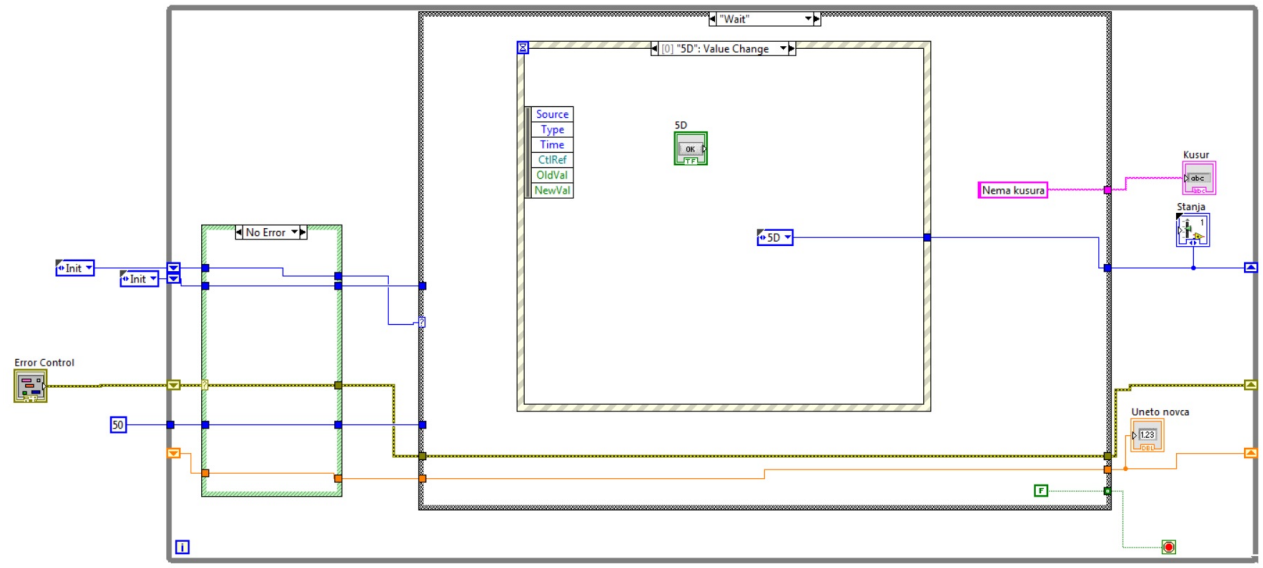
Svakako, u ovoj knjizi, a nadasve u ovom primeru koji ilustrujue primenu automata stanja sa *Event* strukturom, bićemo fokusirani samo

na statičke događaje obaveštenja.



Slika 81: Prikaz *Init* stanja drinkomata sa *Event* strukturom

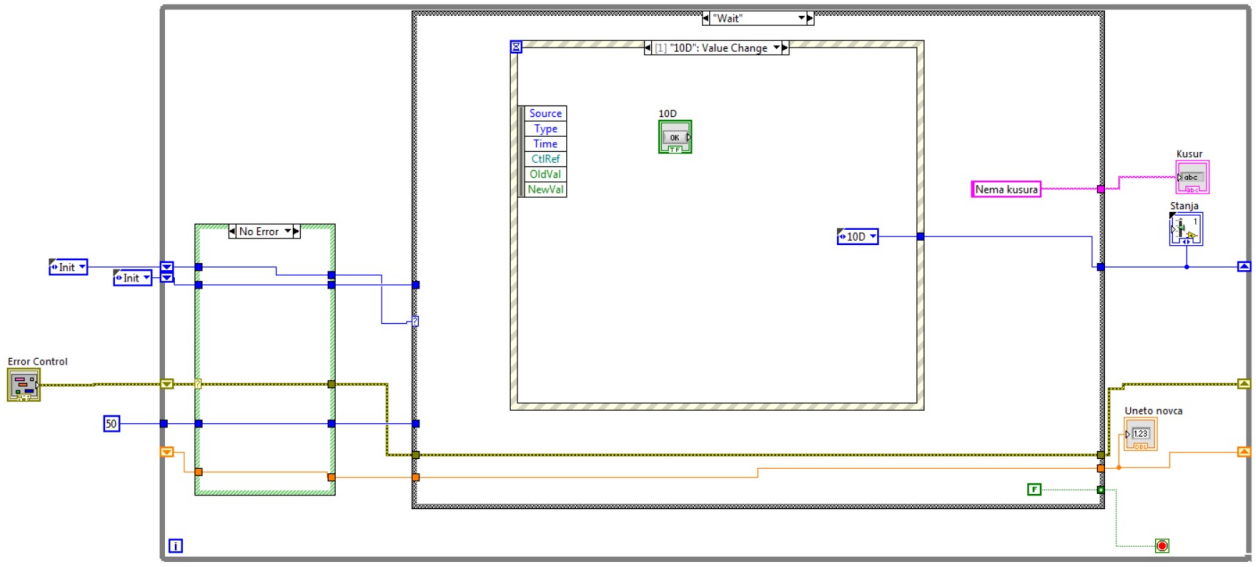
Realizaciju ovakvog automata stanja biće objašnjana kroz primer aparata za izbacivanje konzervi. Ideja je veoma jednostavna, odnosno umnogome uprošćena u odnosu na realni aparat ovakvog tipa. Potrebno je realizovati aparat koji može da izbaci koka-kolu ukoliko primi 20 dinara.



Slika 82: Prikaz *Wait* stanja drinkomata sa *Event* strukturom - slučaj 5 dinara

Neophodno je da automat stanja ima neko početno *Init* stanje, gde se sve vrednosti postavljaju na podrazumevane. Ovo početno stanje je prkazano na Slici 81. Vrednosti koje treba da se vide na korisničkom ekranu se postavljaju na podrazumevane ili na one koje predstavljaju

razumski prvu vrednost. Ovo stanje ima samo tu ulogu - posle obavljene uloge omogućeno je da se iz ovog stanja prelazi direktno u stanje *Wait*.



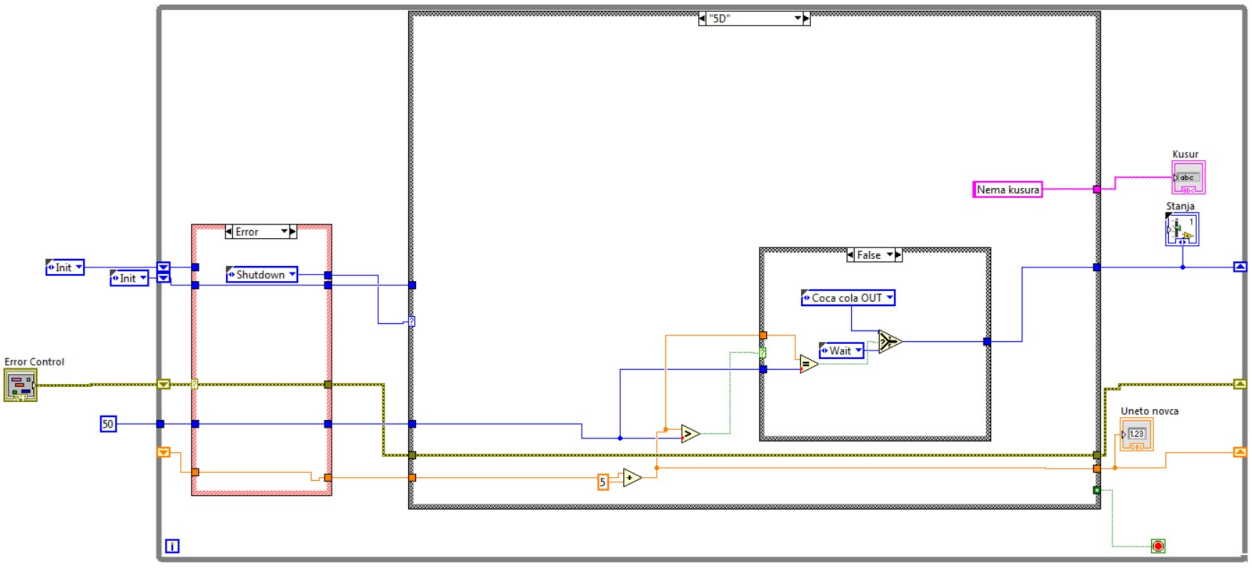
Slika 83: Prikaz *WAIT* stanja drinkomata sa *EVENT* strukturom - slučaj 10 dinara

Kao i svi bazični koncepti automata stanja i ovaj bi trebao da sadrži stanje u kome čeka da se desi neka odgovarajuća promena koja će uzrokovati prelaz u neko od narednih stanja. To stanje se definiše kao satanje čekanja, odnosno *Wait* stanje. U konceptu klasičnog automata stanja bez *Event* strukture u ovom stanju se uglavnom realizuje cela logika prelaska u naredna stanja. Ta logika je bazirana na čitanju velikog broja kontrola u svakoj iteraciji, sve dok se program nalazi u tom stanju. Ovakav pristup troši resurse platforme na kojoj je programa realizovan. Stoga, potrebno je broj čitanja kontrola ograničiti na neki način. To se veoma jednostavno čini ubacivanjem događaja (*Event*-a) za svaku kontrolu koja uzrokuje promenu stanja.

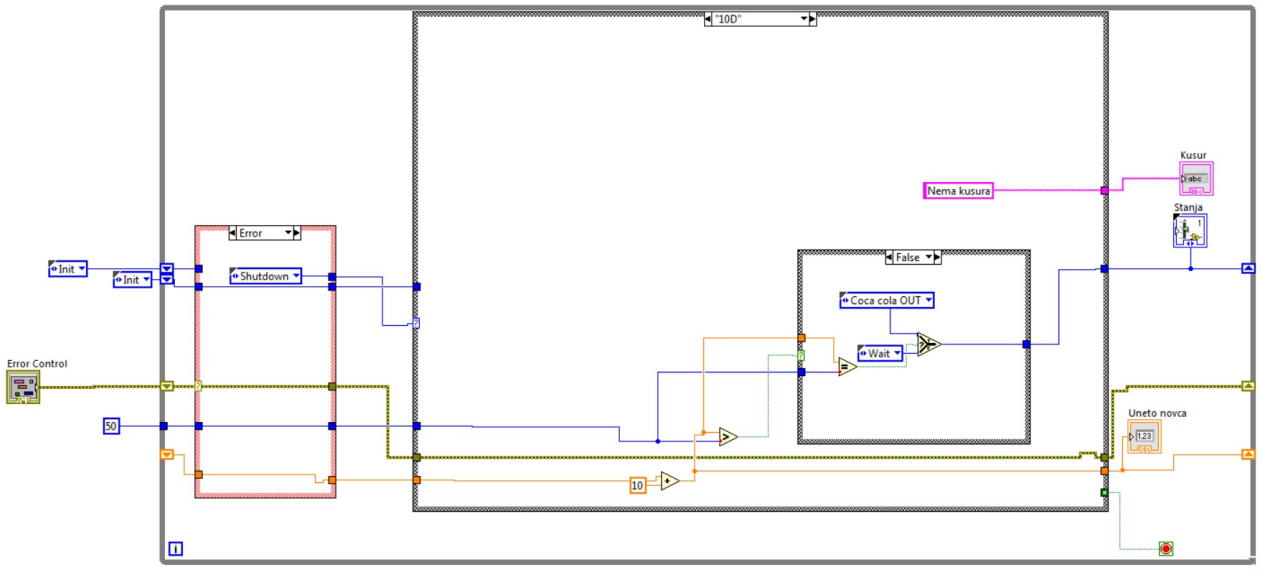
Razlika u ovom pristupu i pristupu kod klasičnih automata stanja je u tome, što je sad procesor oslobođen od proveravanja stanja kontrola u svakoj iteraciji. Ovo stanje omogućava da se detektuje promena vrednosti bilo koje kontrole za koju je događaj definisan, i to se obavalja samo u jednoj iteraciji. Naravno, potrebno je uzeti u obzir da se *Timeout* događaj ne treba realizovati.

Još jedna prednost ovakvog pristupa je čisto vizuelnog karaktera u smislu čitljivosti koda. Kod klasičnih mašina stanja, sve kontorole koje utiču na prelaze između stanja morale su biti postavljene pre leve ivice *Case* strutukre koja definiše stanja. U slučaju automata stanja sa *Event* strukturama, svaka od kontrola je definisana samo u događaju koje se odnosi na nju.

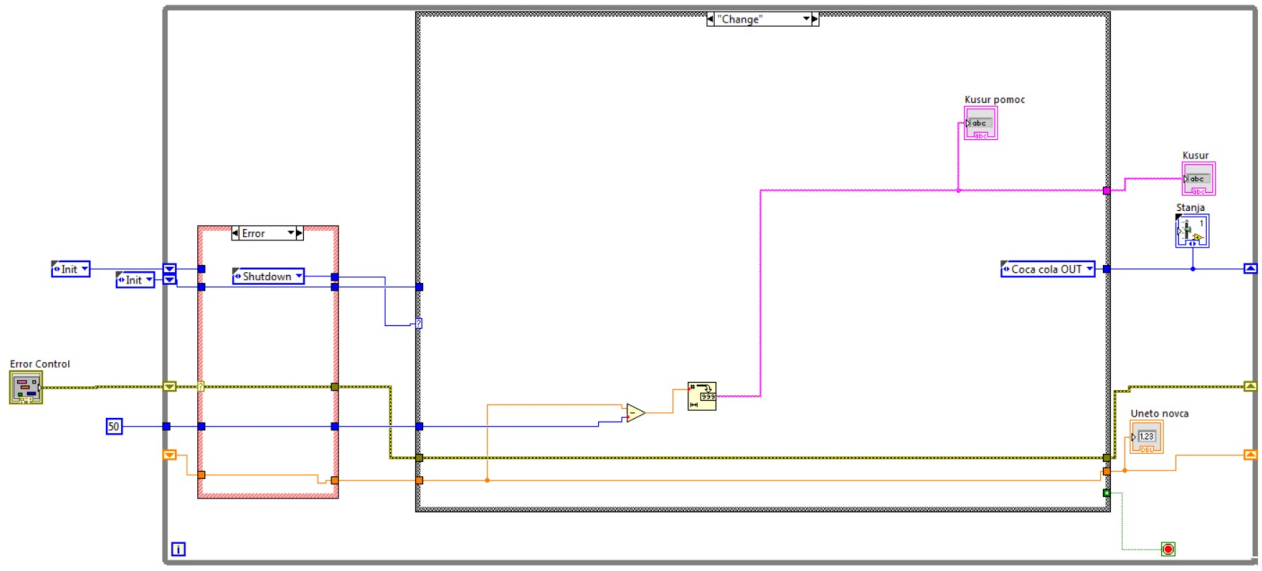
Slike 84 i 85 prikazuju izgled stanja kada se u aparat za izdava-



Slika 84: Prikaz stanja 5 dinara drinkomata sa *EVENT* strukturom

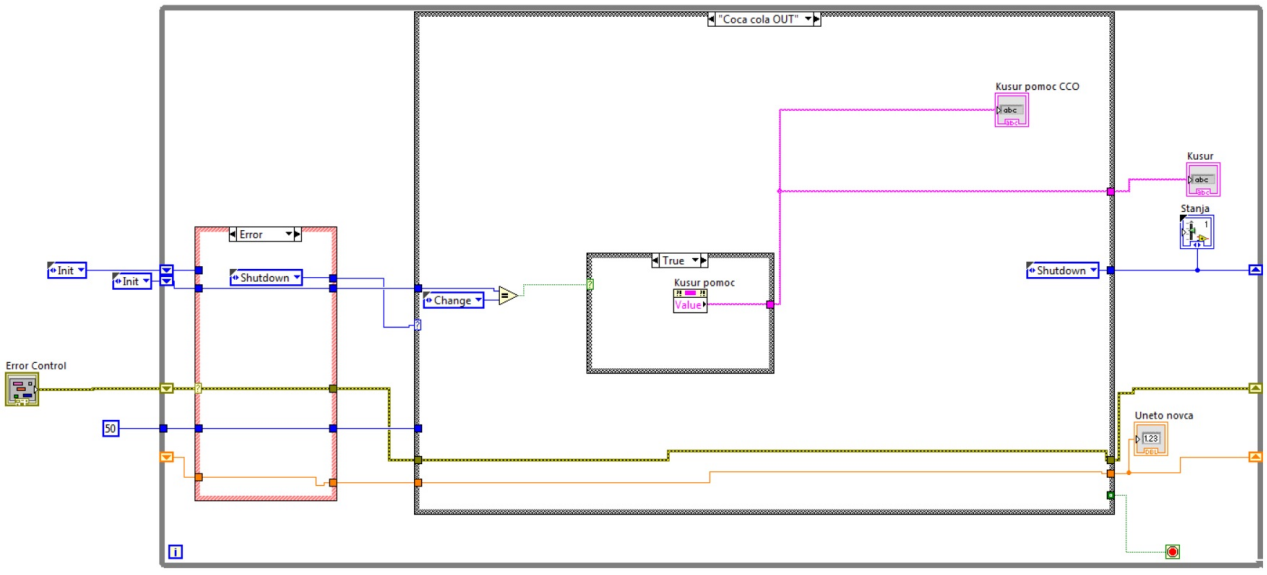


Slika 85: Prikaz stanja 10 dinara drinkomata sa *EVENT* strukturom



Slika 86: Prikaz stanja Kusur drinkomata sa *EVENT* strukturom

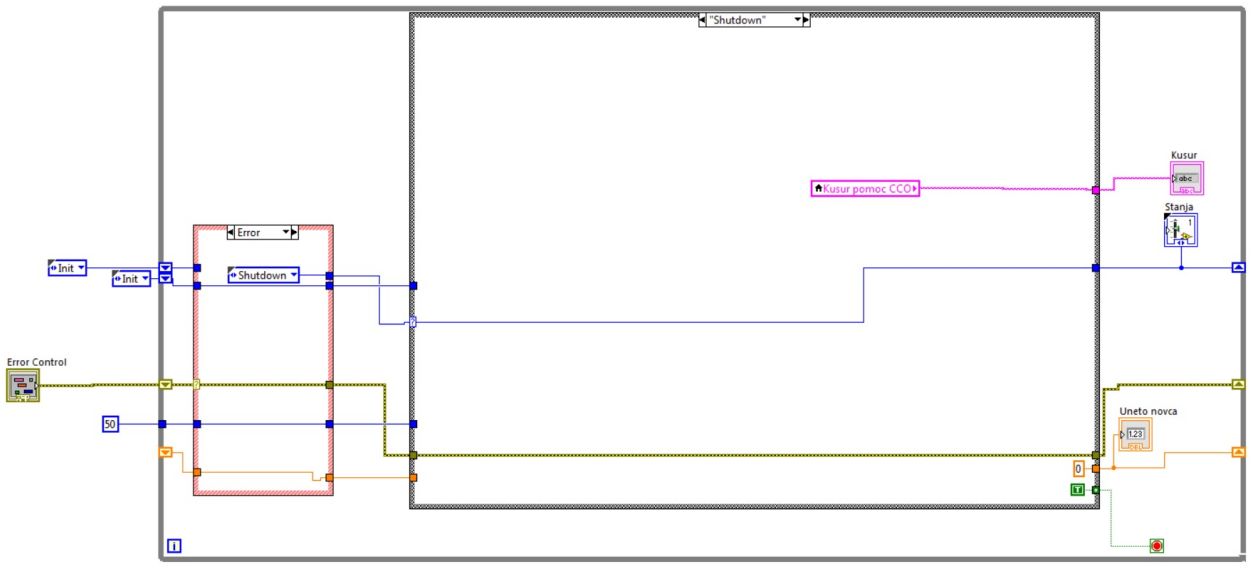
nje koka-kole unese kovanica od 5 ili 10 dinara respektivno. Ova dva stanja su po svojoj strukturi veoma slična. Funkcionišu tako što omogućavaju prelazak nazad u stanje *Wait* ukoliko je ukupna suma novca manja od zahtevane. Ono što ih razlikuje je činjenica da se u stanju 5 dinara može preći sem ovoga stanja, samo još u stanje *Coca-Cola out*. Stanje *Coca-Cola out* omogućava da se izbaci koka-kola iz aprarata.



Slika 87: Prikaz stanja Izbaci koka kolu drinkomata sa *EVENT* strukturom

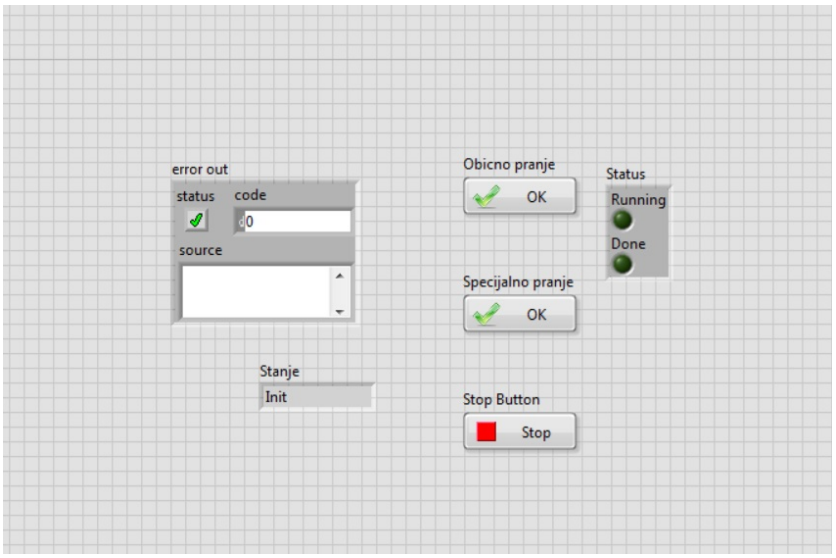
Za razliku od ovog prelaza, ubacivanjem 10 dinara može se preći i u još jedno stanje, a to je *Change* stanje, koje omogućava "vraćanje"kusura, tj. ispisvanje kusura na ekran. Ovo stanje je ilustrovano na slici 86. Pored ovih svih stanja automat stanja sadrži i stanje *Shutdo-*

wn koje predstavlja završno stanje gde je omgućeno da se program zaustavi, kao i da se oslobode svi resursi.



Slika 88: Prikaz stanja *Shutdown* drinkomata sa *EVENT* strukturom

Automati stanja sa redovima (QUEUES) i EVENT strukturom



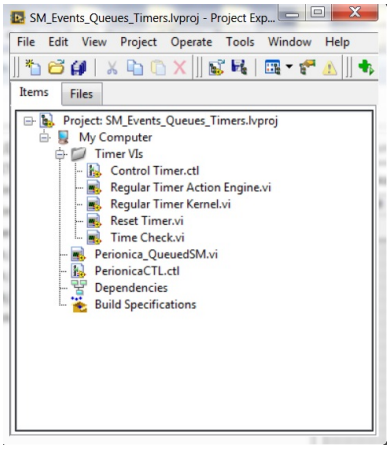
Slika 89: Prikaz *Front panel*-a automata stanja Perionice sa redovima i *Event* strukturom

Klasične mašine stanja sa *Event* strukturama predstavljaju dobar način za rešavanja različitih vrsta problema. Uvođenje *EVENT* strukture u mašine stanja je omogućilo da se ne mora proveravati stanje kontrola u svakoj iteraciji.

Ono što je osnovna odliku klasičnih mašina stanja, a to je da se u svakom stanju mora znati šta predstavlja uslov za prelazaj u naredno, ostaje validno i u slučaju kada se dodaju *Event* strukture.

Problem sa ovako dizajniranim mašinama stanja je to što se samo zna reakcija automata jedno stanje unapred. Problem nastaje kada se automat stanja nađe u stanju iz koga može ući u više stanja, odnosno iz koga ima više prelaza u različita stanja. To stanje predstavlja svojevrsnu tačku granjanja. U tom slučaju se funkcionalnost (logika) koja dovodi do tih prelaza mora razviti u tom jednom stanju.

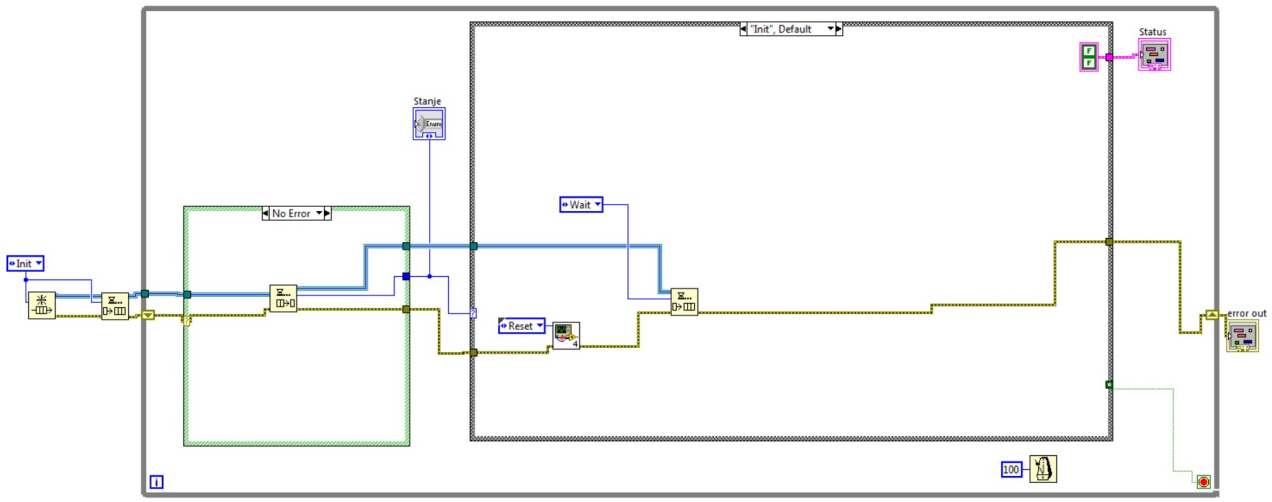
Ako su prelazi stanja zavisni od neke kontrole, onda se mora voditi



Slika 90: Projektna struktura automata stanja Perionica

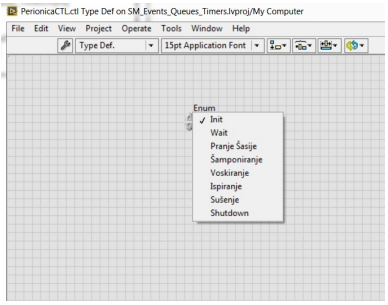
računa da u svakom stanju gde se to dešava da se zna stanje te kontrole. To se kosi sa pristupom sa *EVENT* strukturama. Onda bi opet u svakoj iteraciji morali da proveravamo vrednost svake kontrole. U svakom slučaju, održavanje koda i skalabilnost bi bilo značajno urušeni.

Da bi se rešio ovaj problem, potrebno nam je da znamo listu stanja kroz koje automat stanja treba da prođe, pod uslovom da mu je na početku dat uslov koji mora da ispuni.



Slika 91: Prikaz stanja *Init* automata stanja Perionice sa redovima i *EVENT* strukturom - slučaj kada ne postoji greška

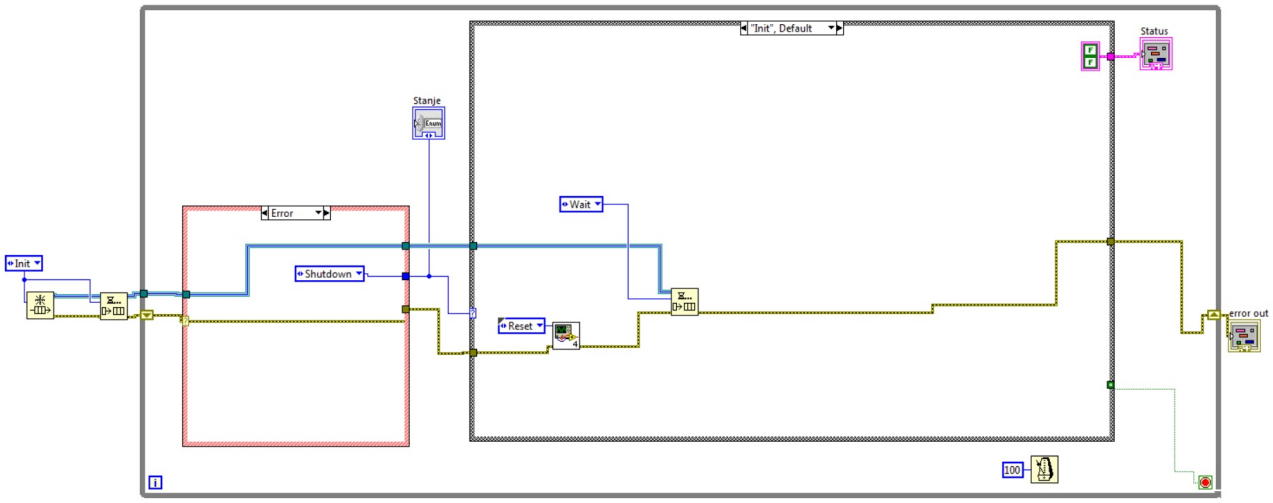
Ukoliko se, na primer, posmatra perionica auotmobila i u zavsinosti od izbora programa (koje je predstavljeno kontrolom na korisničkom programu) treba da se omogući prelaz u neke dve tačke grananja u neko od narednih stanja, jednostavno rešenje bi bilo da se na početku izbora programa sa ulazne kontrole generiše lista stanja kroz koji će ceo automat proći u zavisnosti od početnog izbora. U ovom slučaju se ne mora voditi računa u svakom stanju, u koje naredno stanje automat mora da dospe.



Slika 92: Izgled kontrole za stanja automata stanja Perionica

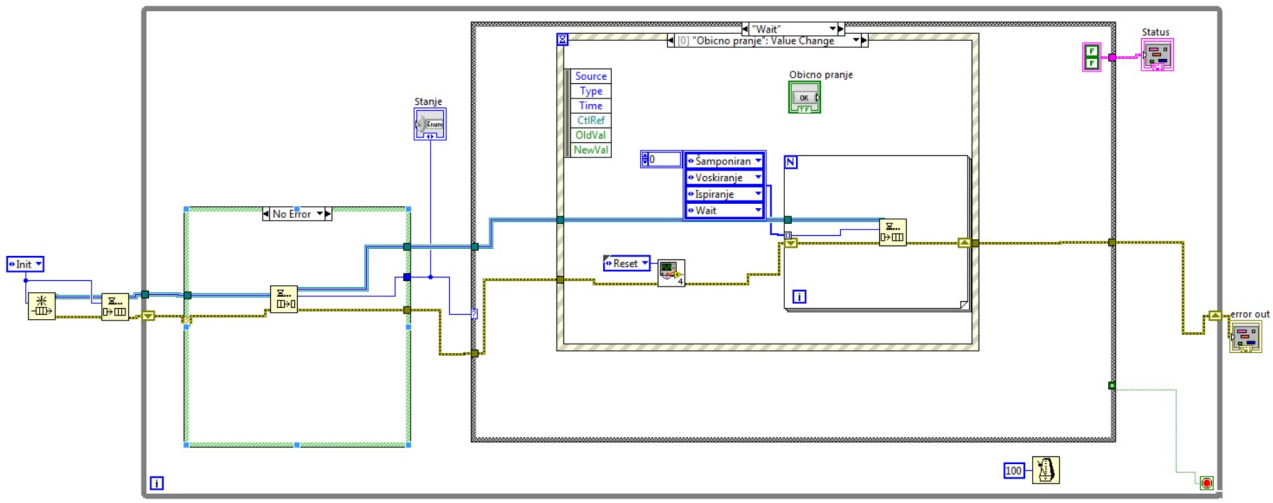
Automat stanja će na osnovu liste stanja znati kako da upravlja tokom celog izvršavanja programa. U tom slučaju neće biti potrebe da se u nekom stanju koje je potencijalno tačka granjanja vodi računa o tome u koje će se naredno stanje preći.

Da bi ilustrovali rad automata stanja sa redovima i *EVENT* strukturom razmotrićemo primer perionice automobila. Na Slici 90 dat je izgled projektnog „drveta" koje predstavlja aplikaciju koja simulira rad perionice. Osnova perionice je VI *Perionica_QueuedSM* koji sadrži sam automat stanja. Na Slici 89 je dat izgled *Front panel*-a ovog VI-a. Osnovna ideja perionice je da omogući korisniku da izabere između dve opcije pranja, *Obično pranje* ili *Specijalno pranje*. U zavisnosti od toga koje pranje je korisnik izabrao, automat stanja treba da prođe kroz



Slika 93: Prikaz stanja **Init** automata stanja Perionice sa redovima i *EVENT* strukturom - slučaj kada postoji greška

različitu sekvencu stanja. Na *Front panel*-u je moguće izabrati jedno od dve vrste pranja, te je moguće posmatrati signal greške, te stanja tajmer-a u svakom od stanja, kao i stanje u kom se sistem trenutno nalazi. Takođe, tu se nalazi i dugme za prekid programa.



Slika 94: Prikaz stanja ***WAIT*** sa izabranim Običnim pranjem automata stanja Perionice sa redovima i *EVENT* strukturom

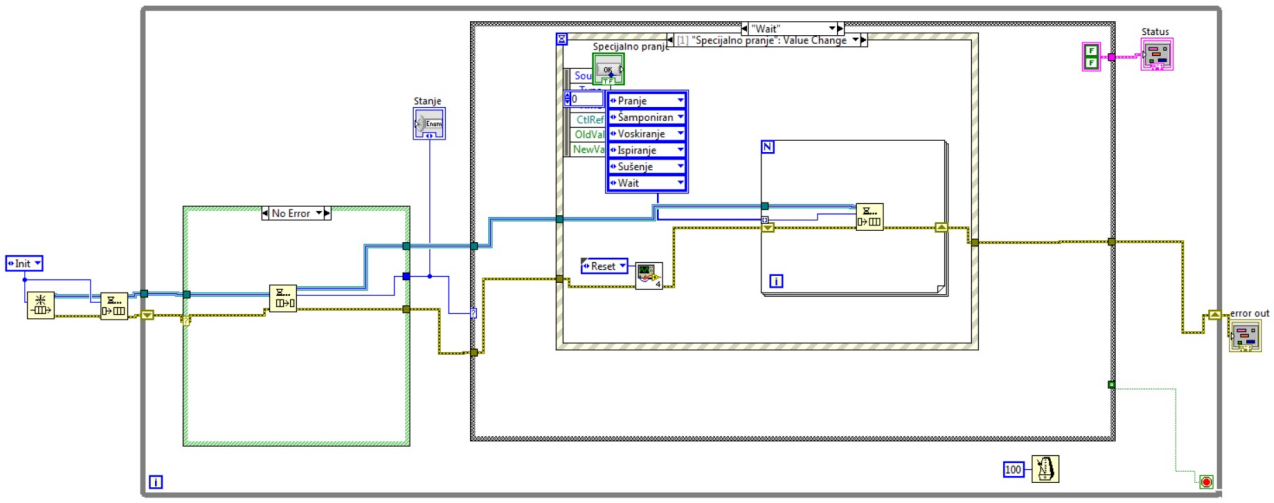
Automat stanja koji je razmatran ovde će imati stanja koja su prikazana na Slici 92. U zavisnosti koja vrsta programa pranja je izabrana, automat će prolaziti kroz neka od ovih stanja.

Dalje će biti analiziran kod koji predstavlja rešenje problema rada perionice.

Kao i svaki do sada razmatrani automat stanja u ovoj knjizi i u ovom

Redovi u LabView-u su podrazumevano FIFO tipa (*First In, First Out*.)

slučaju je potrebno da automat poseduje neko inicijalno stanje koje će postaviti sve vrednosti u programu na podrazumevane ili početne. Ovo stanje je prikazano na Slici 91. Na toj slici se vidi deo levo od *While* petlje koji omogućuje kreiranje reda sa fukcijom *Obtain queue*, te ubacuje prvi elemenat u red (blok *Enqueue element*), a to je *Init* stanje. Na Slici 91 je u *Error* petlji dat slučaj kada nema greške i tom slučaju se treba omogućiti čitanje prvog elementa iz reda (blok *Dequeue element*).



Slika 95: Prikaz stanja *WAIT* sa izabranim Specijalnim pranjem automata stanja Perionice sa redovima i *EVENT* strukturom

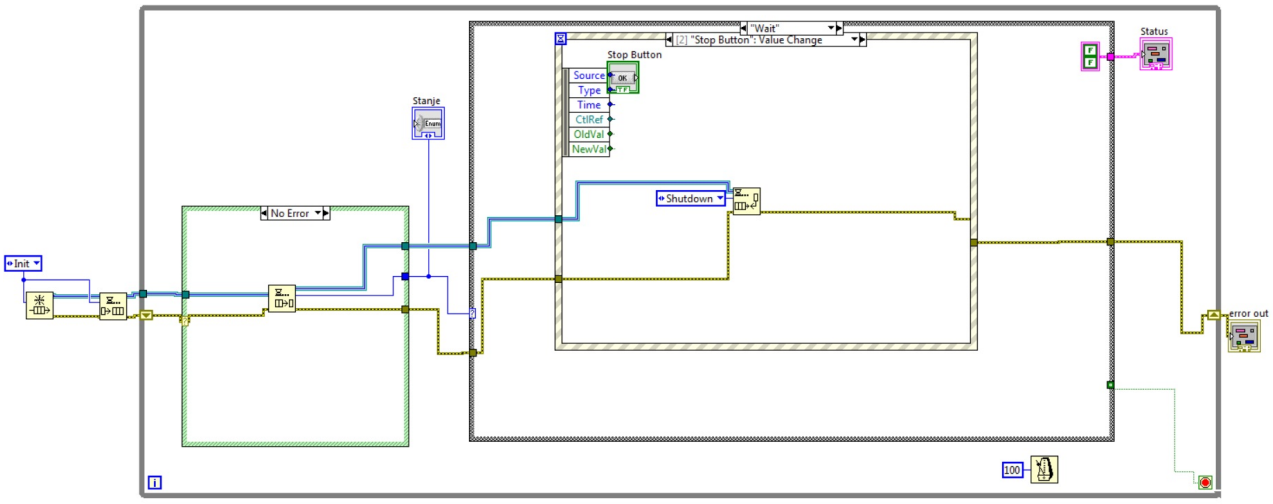
Slučaj kada se pojavi greška u programu i potrebno je detektovati može da se vidi na Slici 93. Tu se vidi u *Error* petlji da se u slučaju detektovanja greške na terminal *CASE* strukture za izbor stanja dovodi stanje *Shutdown* što omogućava da se program zaustavi u tom stanju.

Što se tiče samog stanja *Init*, ono kao i mnogo puta spomenuto do sada, postavalja vrednosti komponenti (indikatora, objekata) na početne (ili podrazumevane) vrednosti. U ovom slučaju, Tajmer koji će odbrojavati u nekom od stanja se resetuje pre prvog pokretanja, a vrednosti koje govore o stanju tajmera se postavljaju na logične inicijalne *false* (tajmer se nije ni pokrenuo, niti je istekao). Važno je napomenuti da između svih blokova koji se direktno odnose na operacije sa redovima mora postojati veza u smislu reference označena debelom plavom linijom.

Redovi u LabView-u se prosleđuju po referenci, a ne po vrednosti.

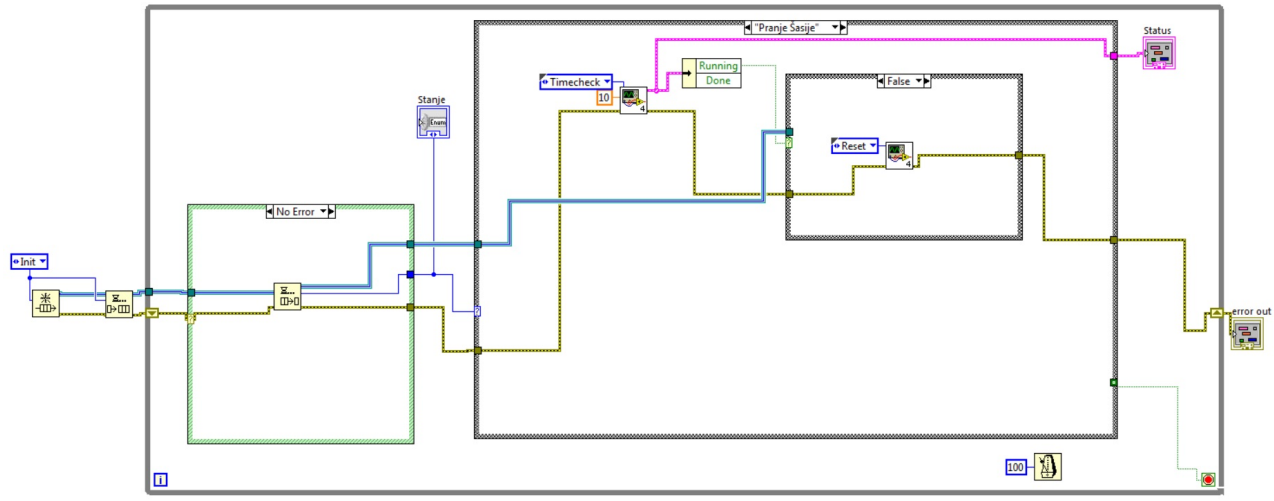
U samoj *Init* petlji (*Case*-u koji opisuje stanja) omogućen je prelazak u *Wait* stanje uz pomoć *Enqueue element* funkcije. *Wait* stanje je centralno za ovakav automat stanja, je u njemu mašina stanja "misli". Upravo se u *Wait* stanju pomoću *Event* strukture bira odlučuje koji događaj će da uzrokuje koji niz akcija automata stanja, odnosno kroz koju sekvencu stanja će automat da prođe da bi ispunio svoju svrhu.

Ukoliko se analizira *Event* slučaj *Obično pranje* (odnosno slučaj kada



Slika 96: Prikaz stanja *Wait* sa pritisnutim Stop dugmetom automata stanja Perionice sa redovima i *Event* strukturom

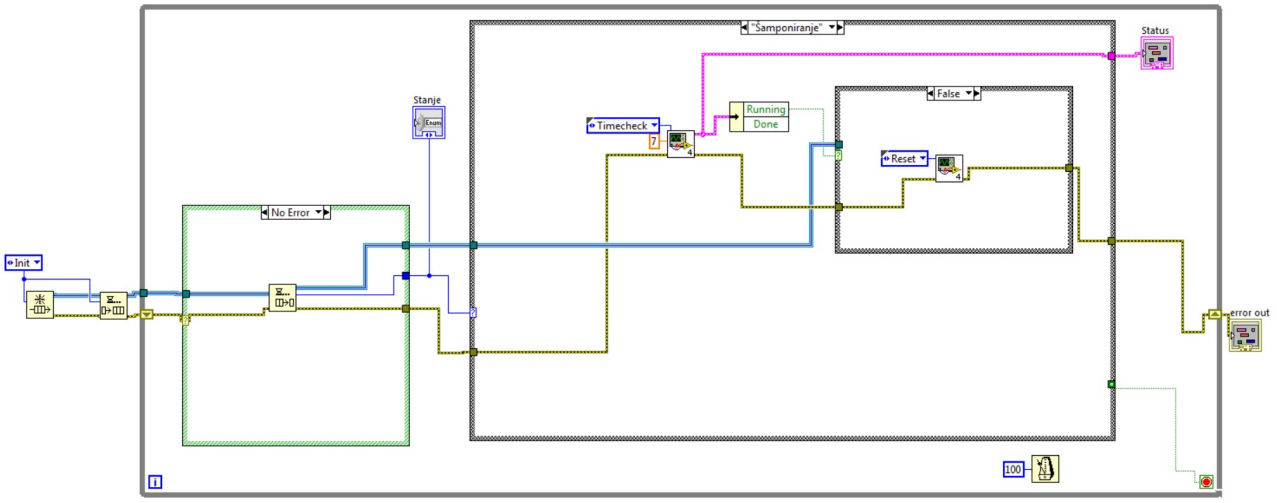
je pritisnuto dugme za izbor Običnog pranja), vidi se da se u njemu unosi niz stanja u red i to u sledećem redosledu *Šamponiranje, Voskiranje, Ispiranje, Wait*. Ova akcija govori, da će automat stanja, ukoliko bude radio bez greške ili nekog iznenadnog prekida, obaviti ovaj niz akcija. Izbor *Običnog pranja* je ilustrovan na Slici 94.



Slika 97: Prikaz stanja *Pranje Šasije* automata stanja Perionice sa redovima i *Event* strukturom

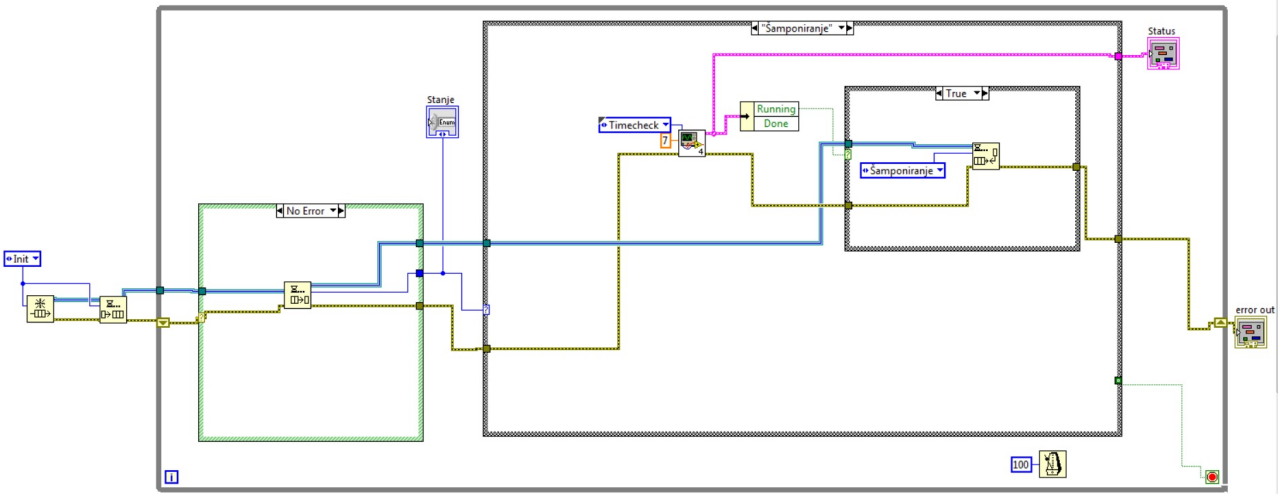
Slična je manipulacija stanjima obavljena i pri izboru *Specijalnog pranja*, samo što se sekvenca razlikuje. Sekvenca stanja u ovom slučaju izgleda ovako: *Pranje šasije, Šamponiranje, Voskiranje, Ispiranje, Sušenje i Wait*. U oba ova slučaja niz stanja se ubacuje u red pomoću *For* petlje koja će da broji onoliko puta koliko je stanja u nizu, te pomoću funkci-

je *Enqueue element* koja omogućava ubacivanje elementa po element u red. Izbor *Specijalnog pranja* je ilustrovan na Slici 95.



Slika 98: Prikaz stanja *Šamponiranje* automata stanja Perionice sa redovima i *Event* strukturom - slučaj kada je Tajmer obrojao

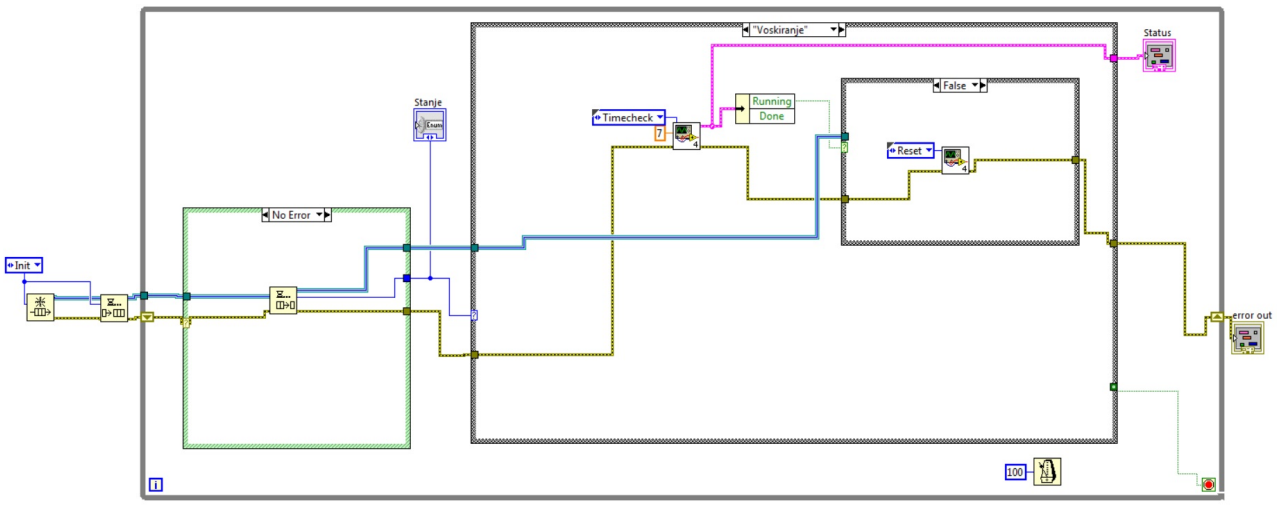
Poslednji događaj koji može da se desi u *Wait* stanju, a uzrokovaće neku promenu stanja je pritisak dugmeta *Stop*. U tom slučaju, potrebno je program prekinuti, odnosno odvesti ga u stanje iz koga će biti prekinut, a to je stanje *Shutdown*.



Slika 99: Prikaz stanja *Šamponiranje* automata stanja Perionice sa redovima i *Event* strukturom - slučaj kada je Tajmer završio brojanje

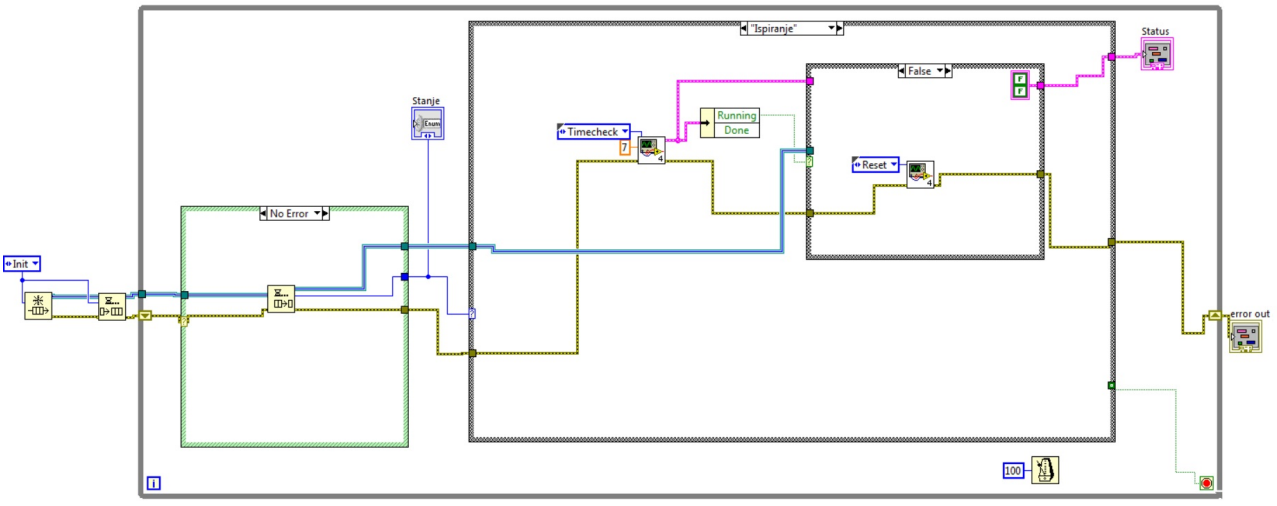
Važno je napomenuti da pritisak na dugme *Stop* mora da omogući momentalni prekid izvršavanja programa stoga se stanje *Shutdown* mora ubaciti na početak reda funkcijom *Enqueue element on opposite side*. O samom *Shutdown* stanju biće reči kasnije.

Ostatak stanja omogućava apliakciji da obavlja svoju fukciju u sva-



Slika 100: Prikaz stanja *Voskiranje* automata stanja Perionice sa redovima i *Event* strukturom

kom od namenskih stanja. Radi ilustracije rada ovakve mašine stanja, ovde u svakom od ovih stanja implementirana manipulacija sa tajmerom kako bi se ostalo u određenom stanju unapred zadatak količina vremena. Za više funkcionalnosti, potrebno je nadograditi ovaj primer, ali pošto ovde nije fokus na funkcionalnosti, već na šemi projektovanja, trudili smo se da uprostimo realizaciju.

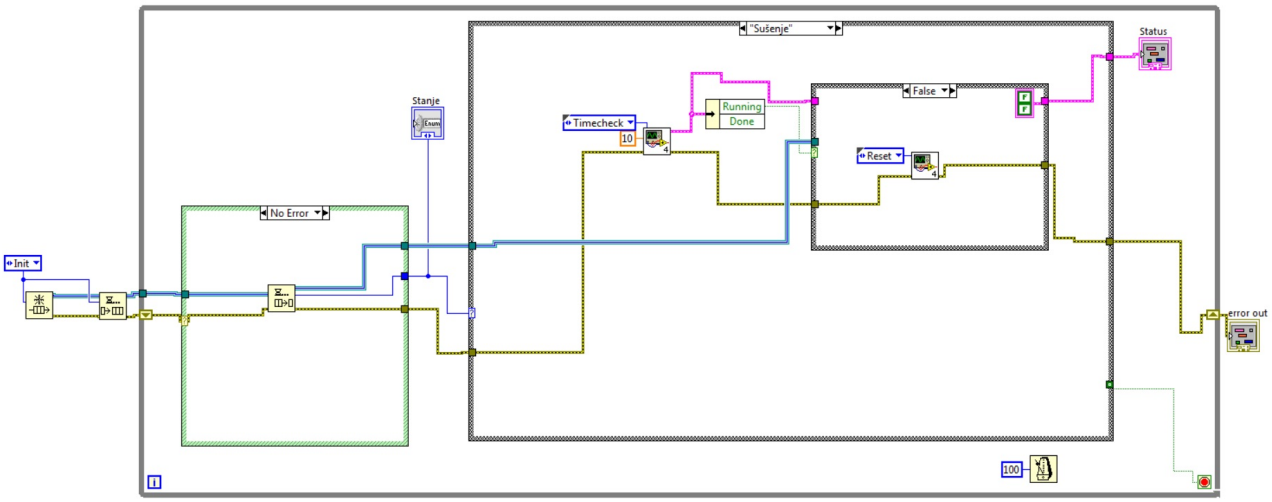


Slika 101: Prikaz stanja *Ispiranje* automata stanja Perionice sa redovima i *Event* strukturom

Biće objašnjeno detaljno stanje Šamponiranje, dok su druga stanja po prirodi relizovana na sličan način. Posmatrajući Sliku 98 može se uočiti pogon stanja Tajmera sa funkcijom *Timecheck* koja omogućava brojanje. Na ovaj način ja omogučeno da program dok god se nalazi u

ovom stanju broji vreme koje je proveo u njemu. Ukoliko vreme koje je provedeno u stanju ne zadovoljava vreme koje je potrebno da se provede u njemu, program automatski opet ubacuje stanje *Šamponiranja* na početak reda (funkcijom *Enqueue element on opposite side*) u *Case*-u *True* ilustrovanom na Slici 99. Ovo omogućava da se u narednoj iteraciji opet uđe u isto stanje, ako je tok izvršavanja programa ispravan. U drugom slučaju, ukoliko je provedeno dovoljno vremena u stanju *Šamponiranja*, potrebno je resetovati Tajmer funckijom *Reset* pogona akcija Tajmera.

U ovom primeru je korišćen Tajmer iz poglavlja .

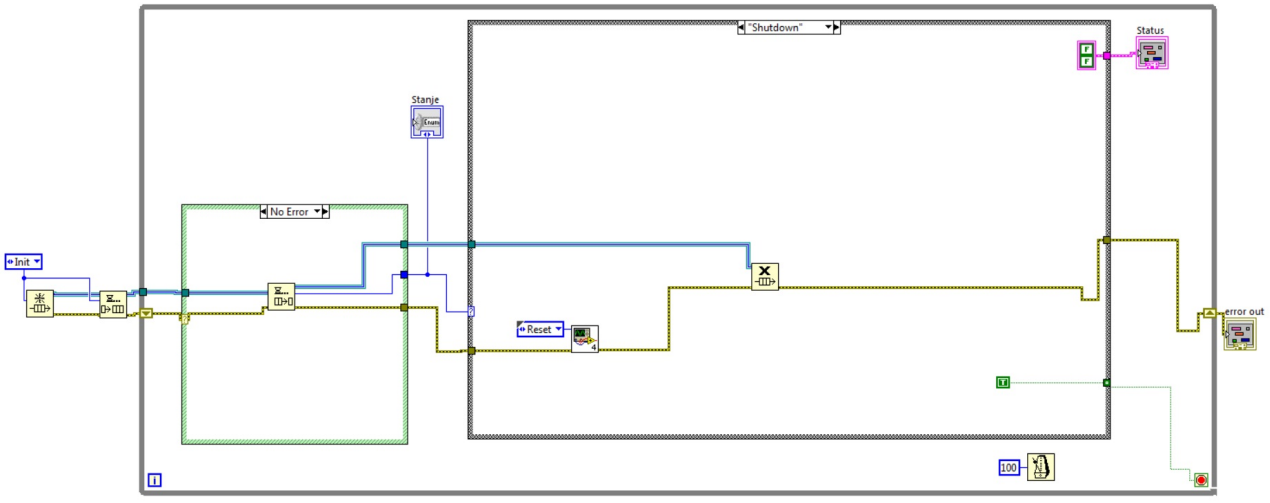


Slika 102: Prikaz stanja **Sušenje** automata stanja Perionice sa redovima i *EVENT* strukturom

Slična manipulacija sa tajemrima je obaljena i u ostalim stanjima vezanim za sam proces pranja, naime *Pranje šasije*, *Voskiranje*, *Ispiranje*, *Sušenje*. Ono što se razlikuje su samo vremena koliko je potrebno provesti u svakom od stanja.

Na kraju, postoji i već spomenuto stanje *Shutdown* u kome je potrebno prekinuti program i osloboditi sve resurse. Prekidanje programa se vrši veoma jednostavno slanjem *True* konstante na uslovni terminal *While* petlje (u svim ostalim stanjima omogučeno je slanje *False* konstante). *Shutdown* stanje ima još dve funkcije, a to je da resetuje tajmer i da oslobodi red uz pomoć funkcije *Release queue*.

Kao što se može videti, automat stanja sa redovima omogućava veliku fleksibilnost i mnogo manje razmišljanja o samoj manipulaciji stanjima, već se programer može fokusirati i na implementaciju funkcionalnosti u svakom od stanja.

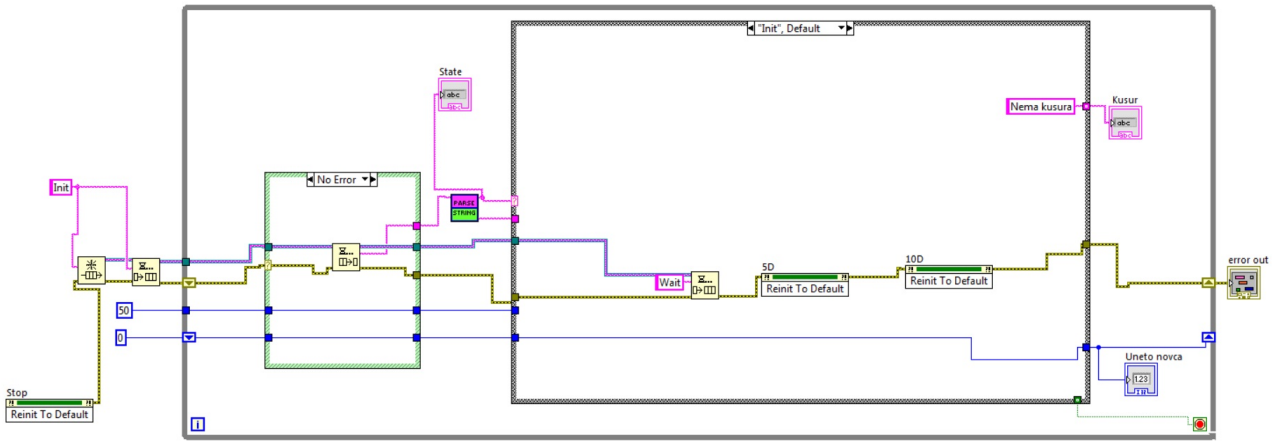


Slika 103: Prikaz stanja ***Shutdown*** automata stanja Perionice sa redovima i *EVENT* strukturom

Automati stanja sa EVENT strukturom i argumentima

Ukoliko su neka od stanja veoma slična, recimo ako očekuju parametre istog tipa na ulazu ili izlazu, moguće je uprostiti takav automat stanja njegovom realizacijom preko argumenata. Jedan primer stanja koja su po svojoj prirodi slična je primer stanja 5D i 10D iz primera automata za izbacivanje koka-kole. Ova dva stanja su po svojoj fukciji identična sem što jedno računa sumu na osnovu novčića od 5 dinara, a drugo na osnovu novčića od 10 dinara. Upravo iz tog razloga, je moguće od ova dva stanja formirati jedno, koje će da manipuliše sumom na različit način u zavisnosti od argumenta koji mu je prosleđen, a to je vrednost apoena novčića, tj. 5 dinara ili 10 dinara.

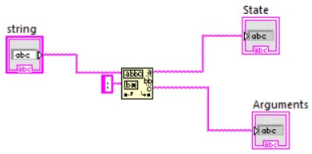
Pristup preko automata stanja sa *EVENT* strukturom i argumentima biće ilustorovan na primeru koji koji će biti objašnjen u nastavku. Primer ilusturuje rad mašine za kupovinu koka-kole, ali su stanja koja su po svojoj prirodi slična realizovana preko argumenata.



Slika 104: Prikaz stanja ***Init*** automata stanja sa argumentima

Na Slici 104 prikazano je stanje ***Init*** mašine stanja koja simulira rad mašine za kupovinu koka-kole. Kao šti je i očekivano, sa slike se može videti da ovo stanje služi da bi se vrednosti promenljivih i funkcija

postavile na podrazumevane. Ono što je iteresantno u ovo primeru je ikonica koja se nalazi levo od *Case* petlje koja omogućava manipulaciju stanjima. Naime, ova ikonica sa natpisom *Parse string* čini *sub-VI* koji omogućava parsiranje stringova. Realizacija ovog *sub-VI*-a je data na Slici 105.



Slika 105: Prikaz VI-a ***Parse Strings***

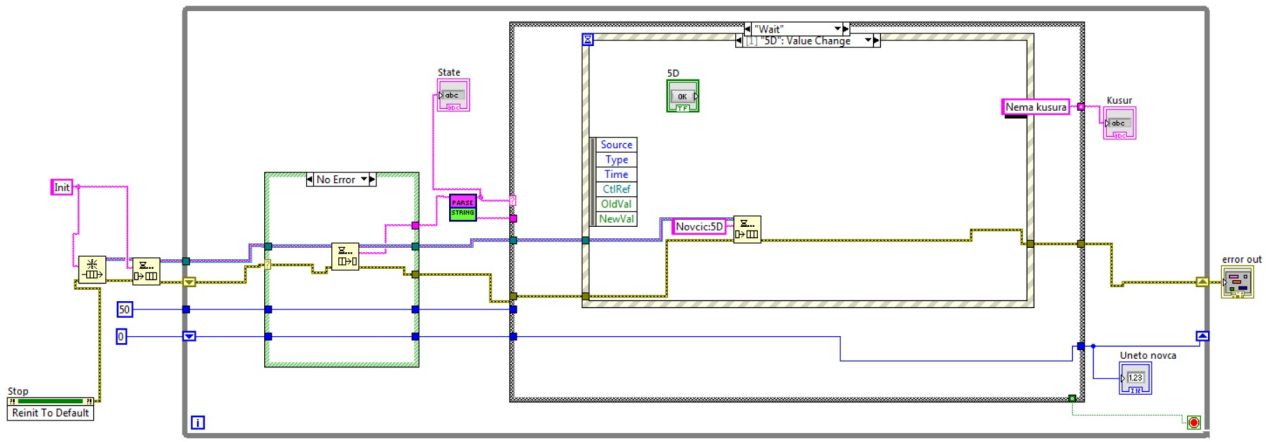
Kao što se vidi na poslednje pomenutoj slici, funkcija ima ulogu da razdvoji string koji je u formi "xxxxxxx:yyyyyyyyyyy"na delove koji odgovaraju tekstu pre delimitera i posle delimitera. Svakako, potrebno je unapred definisati strukturu kako je potrebno „pakovati" string koji treba da čine stanja i argumenti. U ovom konkretnom slučaju je iskorišćeno da stanje mora biti pre delimitera, a argument koji se prosleđuje zajedno sa stanjem iza delimitera.

U ovom primeru je za delimiter korišćeno ":", ali programer je slobodan da izabere koji god želi delimiter.

Ideja ovog pristupa, je da se realizuje samo jedno stanje koje odgovara vrednosti stanja pre delimitera, a u okviru njega u zavisnosti od argumenta se realizuje više funkcionalnosti.

Na Slici 106 se može videti izgled stanja *WAIT* u slučaju kada se ubaci novčić od 5 dinara (formalno to odgovara pritisku na dugme 5 dinara koje simulira obacivanje nočića). U tom stanju, ukoliko se desi adekvatan događaj (pritisak na dugme 5 dinara) se u red dodaje string "Novcic:5D".

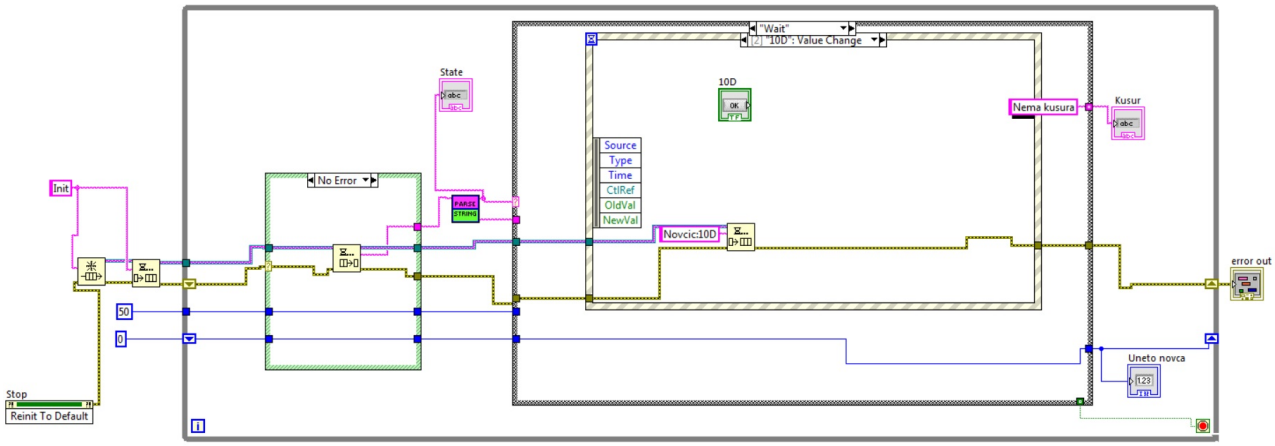
Važno je napomenuti da se stanja *Case* strukturi u ovom slučaju ne prosleđuju kao *custom* kontrola, nego u formi stringa.



Slika 106: Prikaz stanja ***WAIT*** automata stanja sa argumentima - slučaj ubacivanja 5 dinara

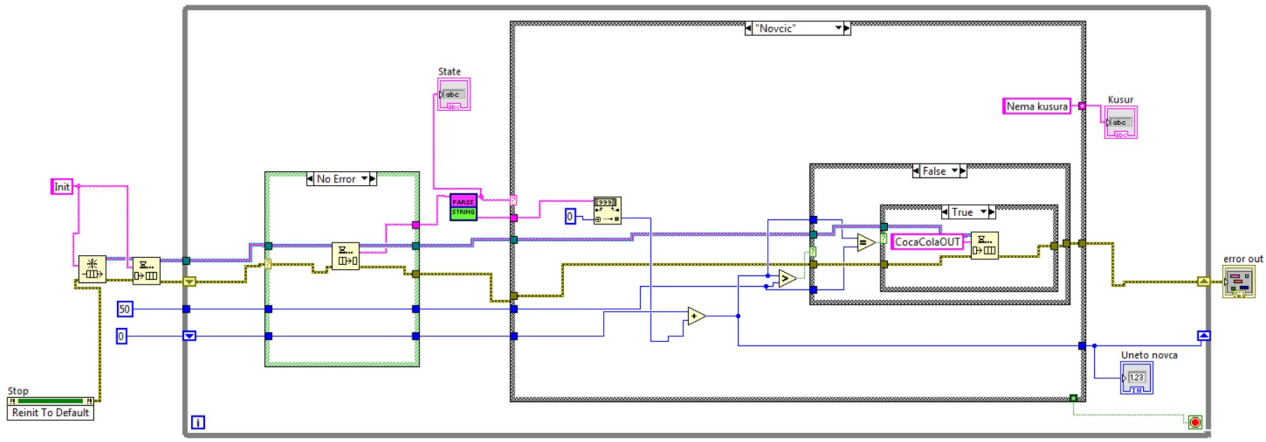
Ovo omogućava da se kada na red dođe ovo stanje da funckija koja je realizovana kao *sub-VI* „Parse String" pozove stanje **Novcic** i prosledi argument 5D, što odgovara slučaju 5 dinara. Naravno, slično je ponašanje i u slučaju datom na Slici 107, samo je tu omogućeno da se prosledi argument 10D stanju **Novcic**.

Na Slici 108 ilustrovan je slučaj kada je u automatu tačno novca. Ovaj slučaj je relalizovan u stanju **Novcic**. Iz stringa se parsiraju podaci o stanju i o količini novca koja je ubačena. Ta količina novca je



Slika 107: Prikaz stanja ***WAIT*** automata stanja sa argumentima - slučaj ubacivanja 10 dinara

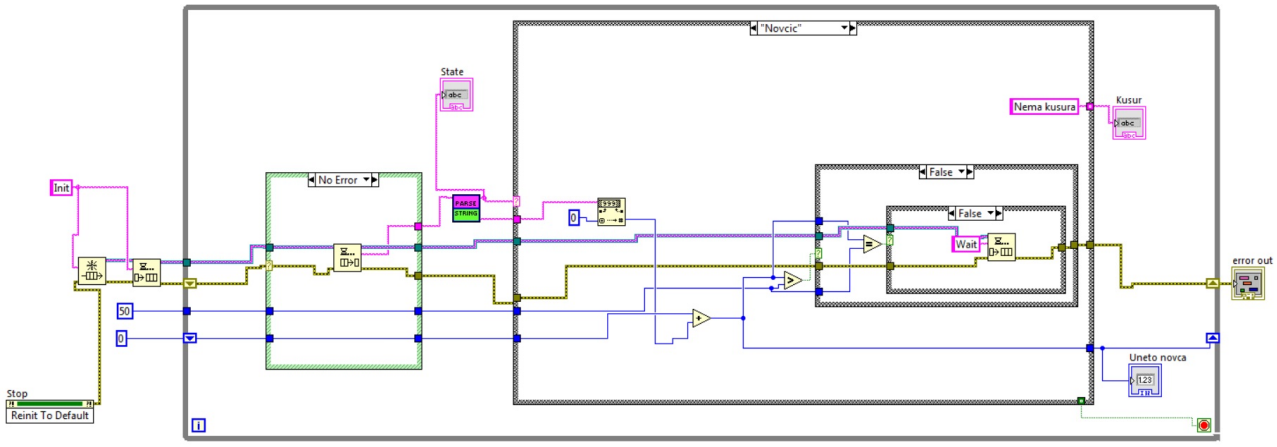
originalno u formi stringa, stoga se obavlja konverzija u celobrojni tip podataka da bi mogla informacija o ubačenom novćiću da se iskoristi za računanje ukupne sume novca u mašini. Suma je realizovana na jednostavan način preko *shift* registara. Ukoliko je trenutna suma novca koja je ubačena u aparat jednka ceni jedne koka kole (tj. kroz dve *Case* strukture je realizovana prvo provera da suma nije veća od zadate, a onda i da nije manja - što znači da je jednaka) treba ubaciti stanje *CocaColaOUT* u red na kraj reda. Ovakva realizacija omogućava da kada je u automatu tačna količina novca, da se izbaci jedna konzerva koka kole kada se uđe u stanje *CocaColaOUT*.



Slika 108: Prikaz stanja ***Novcic*** automata stanja sa argumentima - kada ima tačno novca u automatu

Slična procedura je u slučaju da u automatu nema dovoljno novca. Ta funkcionalnost je ilustorvana na Slici 109. Ono što je specifično je da

je i ova funkcionalnost realizovana u stanju **Novcic**. U ovom slučaju se kroz dve *Case* strukture prvo proverava da suma nije veća od zadate, a onda i da nije jednaka - što znači da je manja). Tada treba ubaciti stanje ***Wait*** u red na kraj reda. Ovakva realizacija omogućava da kada je u automatu manja količina novca od potrebne za kupovinu koka kole, da se sistem vrati u stanje *Wait*, gde opet čeka nastavak ubacivanja novca.



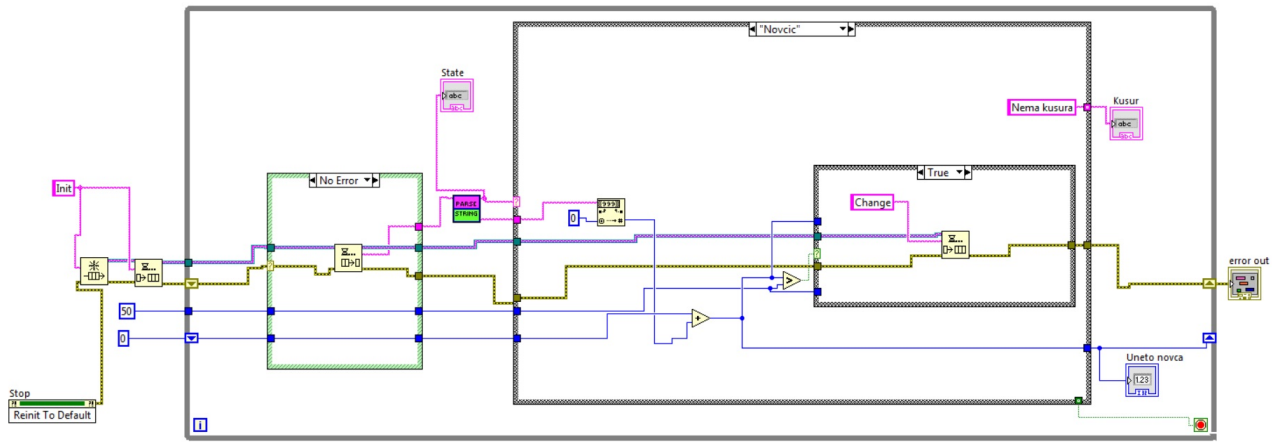
Slika 109: Prikaz stanja ***Novcic*** automata stanja sa argumentima - kada nema dovoljno novca u automatu

Treća varijanta je u slučaju da u automatu ima previše novca. Ova funkcionalnost je ilustrovana na Slici 110. Takođe je i ova funkcionalnost realizovana u stanju **Novcic**. U ovom slučaju se kroz jednu *Case* strukturu proverava da li je suma veća od zadate (sada druga *Case* struktura nije aktuelna pošto se ulazi u *True* slučaj prve *Case* strukture). U slučaju da je u automatu veća količna novca od potrebne količine za kupovinu koka kole, potrebno je da sistem pređe u stanje *Change*, gde je relizovana funkcija koja simulira vraćanje kusura.

Ovo stanje (*Change*) je ilustrovano na Slici 111. Postoje dve osnovne funkcije koje ovo stanje treba da obavi. Prva je da izračuna razliku između trenutne količine novca u automatu i cene koliko košta jedna koka kola i da prikaže tu vrednost u vidu stringa na indikatoru **Kusur**. Ovaj postupak simulira vraćanje kusura. Druga funkcija koja se realizuje u ovom stanju je da se na kraj reda u koji se smeštaju stanja doda stanje ***CocaColaOUT***, da bi se posle vraćanja kusura omogućilo izbacivanje koka kole iz aparata.

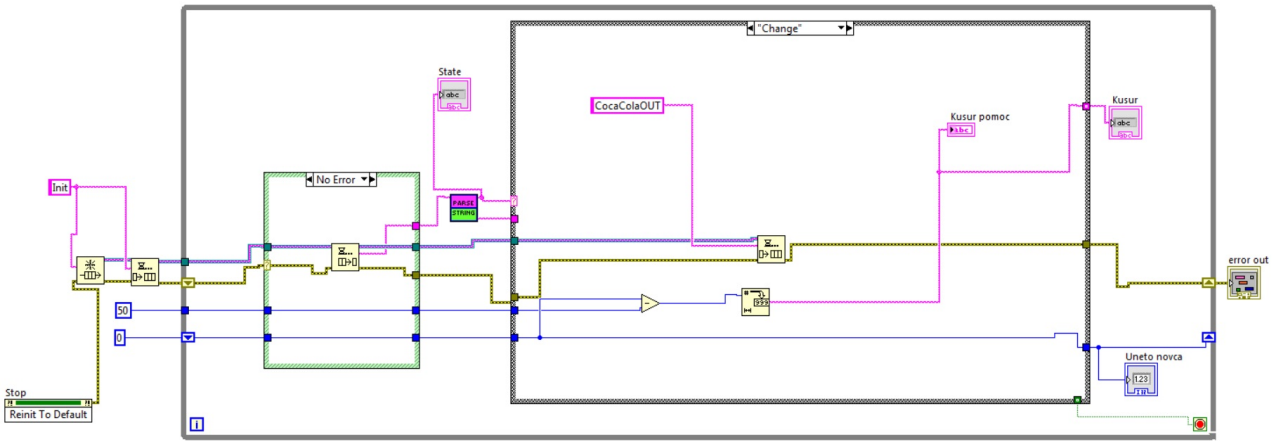
Na Slici 112 dat je prikaz stanja *CocColaOUT*. Ovo stanje u principu samo omogućava da se na kraj reda u koji se smeštaju stanja doda stanje *SHUTDOWN*. Nikakva specijalna realizacija izbacivanja koka kole nije implementirana.

Na kraju u stanju *Shutdown* koje je dato na Slici 113 su oslobođeni



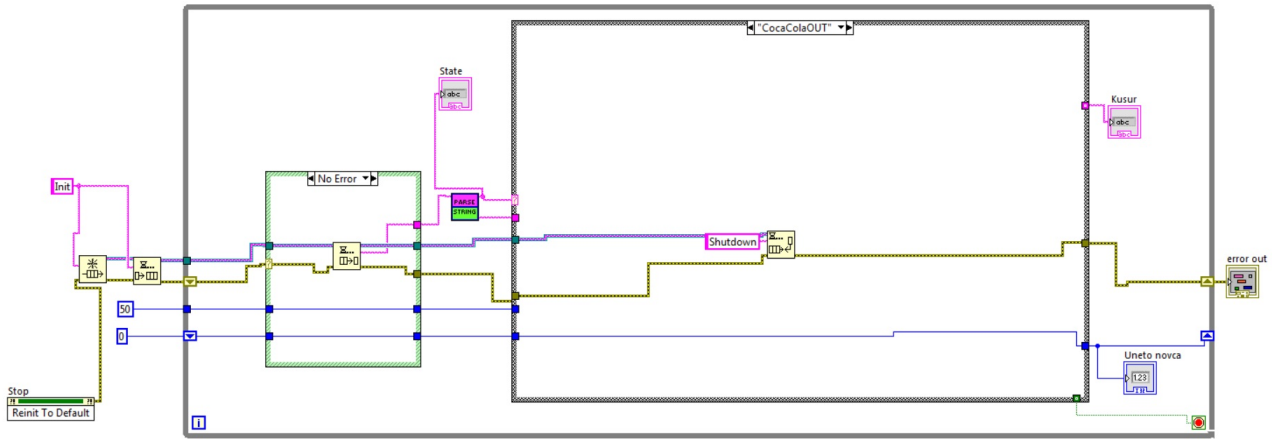
Slika 110: Prikaz stanja ***Novcic*** automata stanja sa argumentima - kada ima kusura u automatu

resursi koji se tiču reda (funkcija *Release queue*), te su postavljene vrednosti dugmadi koji simuliraju unos novčića na podrazumevano *false* stanje. I naravno, omogućeno je da se program završi prosleđivanjem *True* konstante na uslovni terminal *While* petlje.

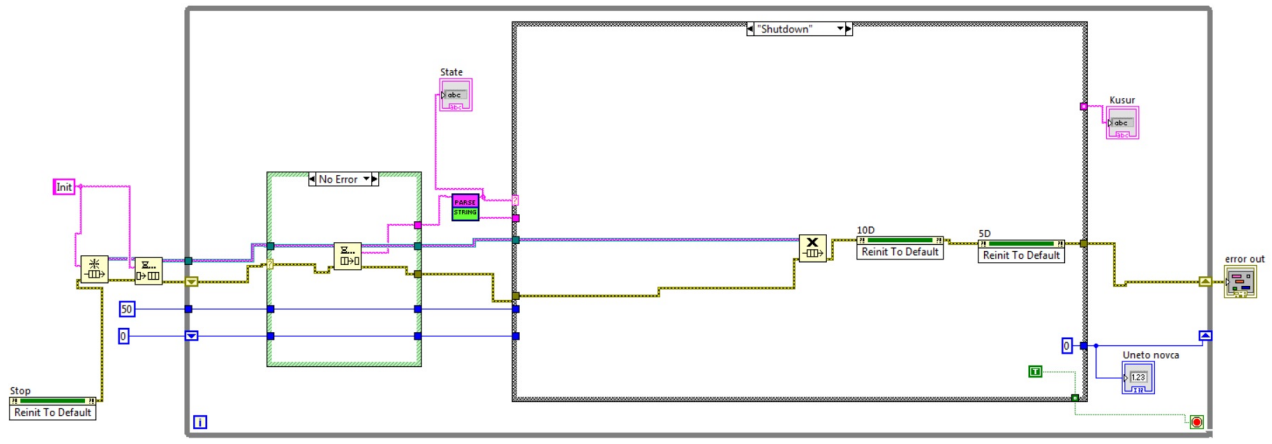


Slika 111: Prikaz stanja ***Kusur*** automata stanja sa argumentima

Na ovaj način je relizovan ceo automat stanja koji simulira rad automata za kupovinu koka kole.



Slika 112: Prikaz stanja ***Coca Cola Out*** automata stanja sa argumentima



Slika 113: Prikaz stanja ***Shutdown*** automata stanja sa argumentima

Simulacija digitalnih regulatora u LabView-u

U ovom poglavlju definisan je pojam PID regulacije, kao i određene modifikacije školskog PID regulatora. Da bi se kontinualni PID regulator mogao implementirati na digitalnom uređaju, potrebno je izvršiti njegovu diskretizaciju. Prikazane su neke od najčešće korištenih metoda za diskretizaciju (Ojler 1, Ojler 2, Tustin). Na kraju, prikazana je implementacija digitalnog PID-a u okviru *LabView* okruženja.

PID regulator

PID regulator predstavlja najčešće korišten regulator. Uz pomoć njega se upravlja sa preko 90% upravljačkih petlji u procesnoj industriji. Razlog njegove popularnosti je njegova jednostavna struktura, kao i mali broj parametara koje je potrebno podesiti [6]. Takođe, nije nužno precizno poznavati model procesa da bi se PID projektovao. PID regulator se sastoji od tri člana, proporcionalnog, integralnog i diferencijalnog. Samim tim, još jedna njegova prednost se ogleda u tome da poseduje fiziku interpretaciju, te je zbog toga jednostavan za razumevanje. Naime, svako od dejstava nam govori o ponašanju sistema ili u datom trenutku ili u prošlosti ili u budućnosti. Ova fizička interpretacija samog regulatora, odnosno svakog od dejstava ponaosob će biti jasnije objašnjena u nastavku kroz matematičku interpretaciju svakog od dejstava.

[6] Karl J. Åström; Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. SA: The Instrumentation, Systems, and Automation Society, 2 sub edition edition, 1995

Proporcionalno dejstvo predstavlja upravljanje u odnosu na trenutnu vrednost greške. Matematička definicija proporcionalnog dejstva data je u jednačini (1). Parametar k_p predstavlja proporcionalno pojačanje.

$$p_d(t) = k_p e(t) \tag{1}$$

Da bi se eliminisala greška u ustanjom stanju potrebno je u regulator dodati integralno dejstvo. Matematička definicija integralnog dejstva data je u jednačini (2). Parametar k_i predstavlja pojačanje integralnog

dejstva.

$$i_d(t) = k_i \int_0^t e(t)dt \tag{2}$$

Diferencijalno dejstvo predstavlja upravljanje u odnosu na izvod greške u vremenu. Uz pomoć diferencijalnog dejstva može da „ulepša" odziv sistema jer diferencijalno dejstvo „predviđa" ponašanje sistema u budućnosti. Matematička definicija diferencijalnog dejstva data je u jednačini (3). Parametar k_d predstavlja diferencijalno pojačanje.

$$d_d(t) = k_d \frac{de(t)}{dt} \tag{3}$$

Sabiranjem proporcionalnog, integralnog i diferencijalnog dejstva, dobija se jednačina školskog PID regulatora

$$u(t) = k_p e(t) + k_i \int_0^t e(t)dt + k_d \frac{de(t)}{dt} \tag{4}$$

Funkcija prenosa školskog PID regulatora dobija se primenom Laplasove transformacije na izraz (4)

$$G_c(s) = k_p + \frac{k_i}{s} + k_d s \tag{5}$$

Modifikacije PID regulatora

Osnovna verzija PID regulatora doživila je nekoliko modifikacija, ovde su prikazane neke od njih. Ako pogledamo strukturu školskog PID regulatora, očigledno je da će postojati problem pri visokofrekventnom šumu zbog postojanja diferencijalnog dejstva. U trenutcima postojanja šuma regulator će imati veliko pojačanje, što nije poželjno. Iz tog razloga u regulator se dodaje niskopropusni filter. Tada „školski" PID postaje

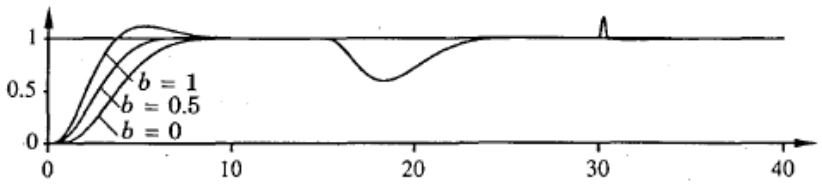
$$G_c(s) = k_p + \frac{k_i}{s} + \frac{k_d s}{T_f s + 1} \tag{6}$$

Navedenom modifikacijom se postiže da pojačanje na visokim frekvencijama bude ograničeno.

Modifikacija P dejstva ogleda se u tome da se željena vrednost i izmerena vrednost tretiraju zasebno, čime se dobija dosta fleksibilnija struktura. To se postiže uvođenjem parametra b čija je vrednost neki broj iz opsega [0,1], a on se u praksi najčešće nalazi u opsegu od [0.7, 1].

$$p(t) = k_p(br(t) - y(t)) \tag{7}$$

Dakle, odziv na promenu reference, zavisiće od parametra b, čija je uloga da smanji preskok. Kada je $b = 1$ odziv će biti brz, a kada je $b = 0$ odziv će biti spor, što je prikazano na Slici (114).



Slika 114: Odziv u zavisnosti od parametra b.

Diskretizacija

Analogne regulatore nije moguće direktno implementirati na digitalnom uređaju. Da bi se regulator mogao implementirati potrebno je naći njegovu digitalnu reprezentaciju. U ovom poglavlju opisana su tri često korištena načina diskretizacije kontinualnih procesa. Sva tri prikazuju transformaciju kontinualne funkcije prenosa u diskretnu funkciju prenosa. Naravno, na digitalnom uređaju nije moguće direktno implementirati ni diskretnu funkciju prenosa, odnosno potrebno je definisati ponašanje sistema u vremenskom domenu. Diferencnu jednačinu ponašanja sistema u vremenskom domenu moguće je dobiti primenom inverzne z-transofrmacije. Primjena inverzne z-transformacije data je kroz primer diskretizacije regulatora.

Diskretizacija ili odabiranje je princip konverzije analognog u digitalni signal. Međutim, diskretizacija sama po sebi ne predstavlja analogno-digitalnu konverziju (A/D konverziju), već samo jedan deo iste. Diskretizacija se može vršiti po:

1. Vremenu
2. Amplitudi

Signali u prirodi su obično kontinualni i po vremenu i amlitudi. Takve signale nazivamo analognim signalima. Da bi se ovi signali mogli obrađivati na digiralnom uređaju potrebno je izvršiti A/D (analogno-digitalnu) konverziju. Prvo se vrši diskretizacija po vremenu, odnosno meri se vrednost signala u određenim vremenskim trenutcima. Obično su ti trenuci ekvidistantni, a vreme između dva trenutka naziva se vreme odabiranja i najčešće se označava sa T ili T_s. Teorijski gledano, vreme odabiranja se bira tako da frekvencija odabiranja bude bar dva puta veća od najveće frekvencije koja se pojavljuje u samom signalu. Pored diskretizacije po vremenu, potrebno je izvršiti i diskretizaciju po amplitudi. Naime, svaki A/D konvertor ima određeni broj bita za reprezentaciju vrednosti signala, te stoga jačina odbirka signala diskretizovanog po vremenu uzima vrednost koja pripada konačnom skupu vrednosti.

Prema diskretizaciji, signali se mogu podeliti u četiri grupe:

1. Vremenski kontinualni,amplitudski kontinualni- Analogni signali
2. Vremenski diskretni, amplitudski kontinualni - Impulsni signali
3. Vremenski kontinualni, amplitudski diskretni - Relejni signali
4. Vremenski diskretni, amplitudski diskretni - Digitalni signali

Ojler 1 transformacija (Euler 1)

Ukoliko znamo kontinualnu funkciju prenosa $G(s)$, diskretna funkcija prenosa $G(z)$ dobija se primjenom smene

$$s = \frac{z-1}{T} \tag{8}$$

gde T predstavlja vreme odabiranja. Očigledno je da je

$$z = sT + 1 \tag{9}$$

Potrebno je razmotriti preslikavanje kompleksne leve s-poluravni upotrebom ove transformacije. Iz izraza (9) jasno je da će se leva s-poluravan biti pomerena udesno za jedan u z domenu, kao što je prikazano na Slici 115a. Ukoliko znamo da je u z domenu oblast stabilnosti jedinična kružnica sa centrom u nuli, jasno je da može da se desi da stabilan proces u kontinualnom domenu postane nestabilan u diskretnom domenu ukoliko se diskretizacija izvrši Ojler 1 metodom. Sa druge strane, prednost ovog metoda je jednostavno računanje.

Ojler 2 transformacija (Euler 2)

Ukoliko znamo kontinualnu funkciju prenosa $G(s)$, diskretna funkcija prenosa $G(z)$ dobija se primjenom smene

$$s = \frac{z-1}{zT} \tag{10}$$

gde je T vreme odabiranja. Očigledno je da je

$$z = \frac{1}{1-sT} \tag{11}$$

Može se pokazati da se upotrebom ove transformacije leva s-poluravan presliava u kružnicu poluprečnika $\frac{1}{2}$ sa centrom u $(\frac{1}{2}, 0)$, što je prikazano na Slici 115b. Dobra osobina ove disretizacije je to što očuvava stabilnost, mada ne preslikava levu s-poluravan u jediničnu kružnicu u z-domenu.

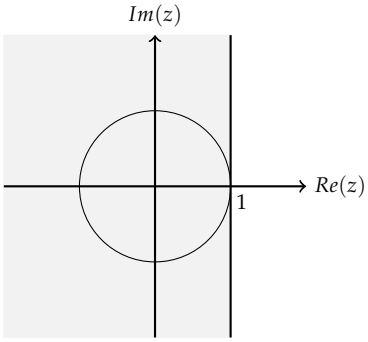
Tustinova (bilinearna) transformacija

Ukoliko znamo kontinualnu funkciju prenosa $G(s)$, diskretna funkcija prenosa $G(z)$ dobija se primjenom smene

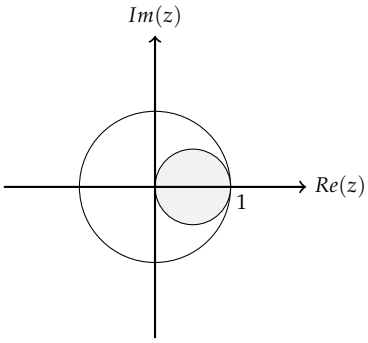
$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{12}$$

gde je T vreme odabiranja.

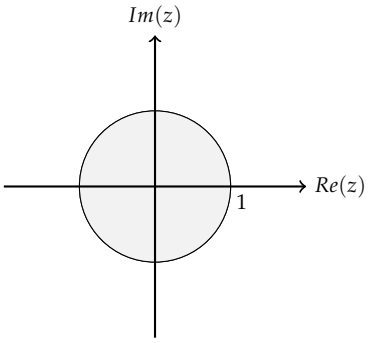
Primenom bilinearne transformacije leva s-poluravan preslikava se u jediničnu kružnicu sa centrom u koordinatnom početku, što je prikazano na Slici 115c. Dakle, bilinearnom transformacijom oblast stabilnosti u kontinualnom domenu preslikava se u oblast stabilnosti u diskretnom domenu.



(a) Ojler 1 transformacija



(b) Ojler 2 transformacija



(c) Tustinova transformacija

Slika 115: Oblast u koju se preslika leva s-poluravan upotrebom karakterističnih transformacija

Diskretizacija PID regulatora

Kao što je već rečeno ranije, da bi se PID implementirao na digitalnom uređaju potrebno ga je diskretizovati. Bez gubljenja na opštosti prikazaćemo diskretizaciju školskog PID-a. Postupak za diskretizaciju PID-a sa određenim modifikacijama se konceptualno ne menja, samo je račun drugačiji. Naravno, nakon dobijanja diskretne funkcije prenosa regulatora, potrebno je odrediti diferencnu jednačinu regulatora da bi se ona mogla implementirati na digitalnom uređaju.

Funkcija prenosa školskog PID regulatora je

$$G_c(s) = k_p + \frac{k_i}{s} + k_d s = \frac{k_p s + ki + kds^2}{s}$$

.

Da bi smo dobili diskretnu funkciju prenosa regulatora potrebno je da primenimo neku od transformacija koje su ranije opisane. U ovom primeru iskoristićemo Tustinovu aproksimaciju koja koristi smenu

Osobina inverzne z-transformacije koja definiše vremenko pomeranje $Z^{-1}\left\{z^{-n}X(z)\right\} = x(kT - nT)$

$$s = \frac{2}{T}\frac{z-1}{z+1}.$$

Kada izvšimo navedenu smenu nad kontinualnom funkcijom prenosa regulatora $G_c(s)$ dobijamo diskretnu funkciju prenosa PID regulatora

$$G_c(z) = \frac{k_p\frac{2}{T}\frac{z-1}{z+1} + k_i + k_d(\frac{2}{T}\ fracz - 1z + 1)^2}{\frac{2}{T}\frac{z-1}{z+1}}.$$

Sređivanjem ovog izraza dobija se

$$G_c(z) = \frac{k_p(z^2-1) + \frac{k_iT}{2}(z^2+2z+1) + \frac{2k_d}{T}(z^2-2z+1)}{z^2-1}.$$

Konačno potrebno je grupisati članove po stepenima promenljive z. Dobija se

$$G_c(z) = \frac{z^2(k_p + \frac{k_iT}{2} + \frac{2k_d}{T}) + z(k_iT - \frac{4k_d}{T}) - kp + \frac{k_iT}{2} + \frac{2k_d}{T}}{z^2-1}.$$

Sada je potrebno odrediti diferencnu jednačinu regulatora. Znamo da je $G_c(z) = \frac{U(z)}{E(z)}$, gde je $U(z)$ signal upravljanja, a $E(z)$ signal greške iz čega sledi

$$(z^2-1)U(z) = (z^2(k_p + \frac{k_iT}{2} + \frac{2k_d}{T}) + z(k_iT - \frac{4k_d}{T}) - kp + \frac{k_iT}{2} + \frac{2k_d}{T})E(z).$$

Primenom inverzne z-transformacije na navedeni izraz dobija se

$$u[kT+2T] - u[kT] = k_1e[kT+2T] + k_2e[kT+T] + k_3e[kT],$$

gde su

$$k_1 = k_p + \frac{k_iT}{2} + \frac{2k_d}{T} \tag{13}$$

$$k_2 = k_iT - \frac{4k_d}{T} \tag{14}$$

$$k_3 = \frac{k_iT}{2} + \frac{2k_d}{T} - k_p. \tag{15}$$

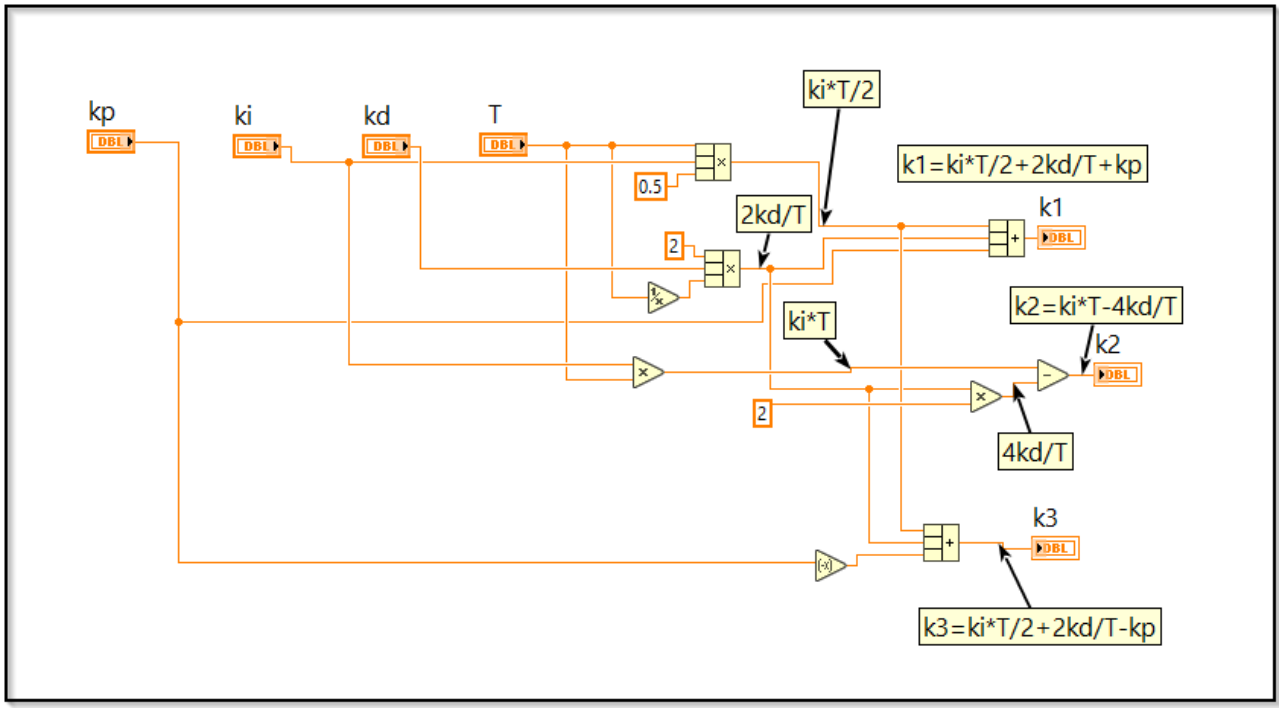
Da bismo ovaj izraz prilagodili tako da trenutno upravljanje računamo na osnovu prethodnih vrednosti greške i upravljanja, potrebno je da čitav izvraz vremenski pomerimo za dva trenutka unazad. Matematički to bi značilo da uvodimo smenu $nT = kT + 2T$. Tada prethodni izraz postaje

$$u[kT] = u[kT-2T] + k_1e[kT] + k_2e[kT-T] + k_3e[kT-2T].$$

Dakle, dobija se diferencna jednačina koja prikazuje kako izgleda upravljanje u tekućem jednačinu. Navedenu jednačinu moguće je implementirati na digitalnim uređajima.

Implementacija PID regulatora

Nakon što se odredi jednačina upravljanja regulatora, moguće je pristupiti njegovoj implementaciju u nekom oruženju, u ovom slučaju u *LabView* okruženju. Ideja je da se implementira PID kao *subVI* koji se kasnije može iskoristiti za upraljanje nekim procesom. Naravno, parametri PID regulatora k_p, k_i i k_d, kao i vreme odabiranja T će biti ulazni parametri ovog *subVI*-a da bi on bio podesiv.



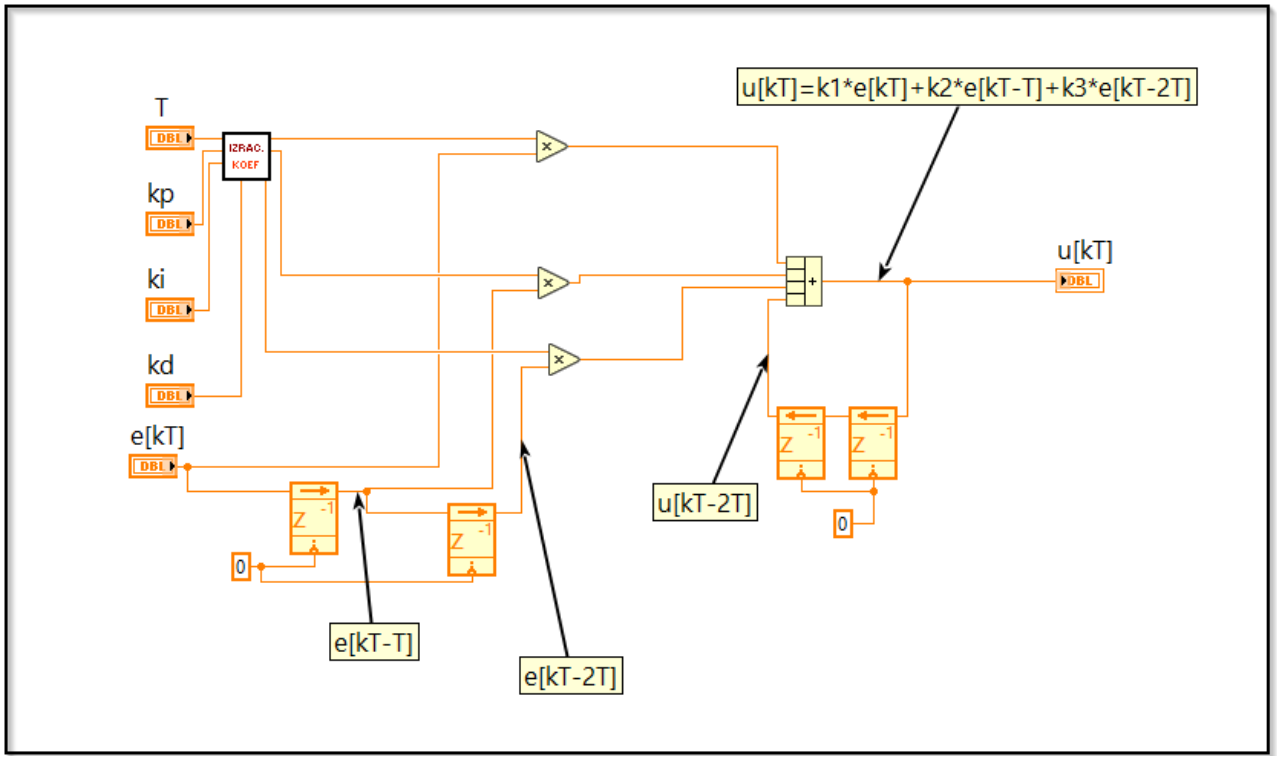
Slika 116: Prikaz izračunavanja pomoćnih koeficijenata za implementaciju školskog PID regulatora.

Zgodno je da se računanje koeficijenata k_1, k_2 i k_3, navedenih u jednačinama (13-15) izvdvoji u poseban *subVI* da bi kod izgledao kompaknije. Ulazi u ovaj *subVI* treba da budu sve konstante koje su nam potrebne za izračunavanje konstanti k_1, k_2 i k_3, a to su k_p, k_i, k_d i T. Naravno, izlazi su koeficijenti k_1, k_2 i k_3. Implementacija je prikazana na Slici 116. Nema potrebe da se detaljno ulazi u analizu implementacije pošto se sve svodi na algebarsko računanje izlaznih koeficijenata.

Implementacija samog PID regulatara prikazana je na Slici 117. Ulazi u ovaj *subVI* su k_p, k_i, k_d, T, kao i trenutna vrednost greške e. Izlaz iz PID-a je trenutna vrednost upravljanja koja se izračunava kao

$$u[kT] = u[kT - 2T] + k_1 e[kT] + k_2 e[kT - T] + k_3 e[kT - 2T].$$

Da bismo implemetirali ovu jednačinu, potrebno je odrediti vrednosti greške u prethodna dva trenutka izvršavanja ($e[kT - T]$ i $e[kT - 2T]$),



Slika 117: Implementacija školskog PID regulatora.

kao i vrednost upravljanja u trenutnu $kT - 2T$. Ove vrednosti moguće je dobiti upotrebom *Feedback Node*-a, čija je osnovna funkcionalnost prosleđivanje vrednosti promenljive u narednu iteraciju. Na taj način možemo dobiti prethodnu vrednost. Naravno ulančavanjem ovih struktura možemo dobiti čitav niz prethodnih vrednosti. *Feedback Node* poseduje terminal za inicijalizaciju. Prikaz *Feedback Node*-a se može promeniti, tj. može se izabrati prikaz *z-Transform Delay Node*. Naravno, izbor prikaza ne utiče na funkcionalst. Pored određivanja prethodnih vrednosti greške i upravljanja, potrebno je izračunati koeficiente k_1, k_2 i k_3, što je izvšeno upotrebom funkcionalnosti za izračunavanje koeficijenata koja je opisana ranije. Ostatak implementacije su jednostavne matematičke operacije da bi se došlo do konačnog izraza za vrednost trenutnog upravljanja, te nema potrebe da se detaljno objašnjavaju.

Prikaz funkcionisanja impementiranog PID regulatora biće prikazan kroz primer koji sledi. Simuliraće se upravljanje procesom drugog reda upotrebom školskog PID regulatora.

Primer simulacije

U ovom delu prikazana simulacija rada digitalnog regulatora. Da bismo testirali regulator potreban je proces kojim će se upravljati. Kao

primer uzećemo proces čija funkcija prenosa ima oblik

$$G(s) = \frac{b}{s(s+a)}. \tag{16}$$

Kao što je bilo potrebno diskretizovati regulator da bi se izvršavao na digitalnom uređaju, isto tako potrebno je diskretizovati proces da bi mogli simulirati njegovo ponašanje na digitalnom uređaju. U ovom primeru diskretizacija je izvršena Tustinovom transformacijom. Primenom Tustinove transformacije na izraz (16) dobija se diskretna funkcija prenosa procesa

$$G(z) = \frac{b}{\frac{2}{T}\frac{z-1}{z+1}(\frac{2}{T}\frac{z-1}{z+1}+a)}.$$

Sređivanjem ovog izraza dobija se

$$G(z) = \frac{\frac{bT}{2(a+\frac{2}{T})}z^2 + \frac{bT}{a+\frac{2}{T}}z + \frac{bT}{2(a+\frac{2}{T})}}{z^2 - \frac{4}{a+\frac{2}{T}}z + \frac{\frac{2}{T}-a}{\frac{2}{T}+a}},$$

što se može zapisati kao

$$G(z) = \frac{b_2z^2 + b_1z + b_0}{z^2 + a_1z + a_0},$$

gde su koeficijenti b_2, a_1, a_0:

$$b_2 = \frac{bT}{2(a+\frac{2}{T})},$$
$$b_1 = \frac{bT}{a+\frac{2}{T}},$$
$$b_0 = \frac{bT}{2(a+\frac{2}{T})},$$
$$a_1 = -\frac{4}{a+\frac{2}{T}},$$
$$a_0 = \frac{\frac{2}{T}-a}{\frac{2}{T}+a}.$$

Iz činjenice da je $G(z) = \frac{Y(z)}{U(z)}$ dobija se

$$(z^2 + a_1z + a_0)Y(z) = (b_2z^2 + b_1z + b_0)U(z).$$

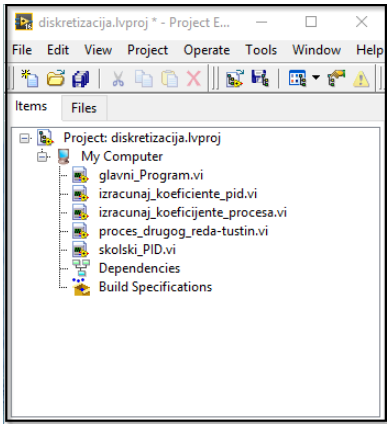
Primenom inverzne z-transformacije na prethodni izraz dobija se

$$y[kT+2T] + a_1y[kT+T] + a0y[kT] = b_2u[kT+2T] + b_1u[kT+T] + b_0u[kT].$$

Da bismo ovaj izraz prilagodili tako da trenutni izlaz računamo na osnovu prethodnih vrednosti izlaza i upravljanja, potrebno je da čitav izvraz vremenski pomerimo za dva trenuta unazad. Matematički to

Napomena: Tustinova transformacija je jedan od načina diskretizovanja kontinualne funkcije prenosa u diskretnu funkciju prenosa. Vrši se smena:

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

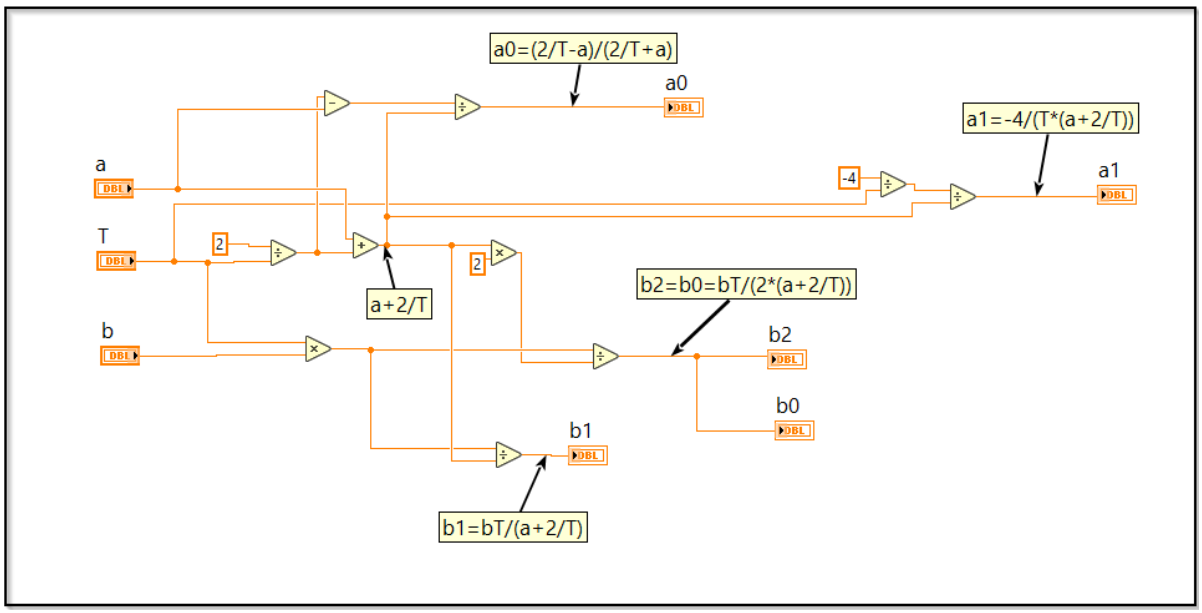


Slika 118: Prikaz stabla projekta. Sadrži sve pomoćne funkcionalnosti i glavni program.

bi značilo da uvodimo smenu $nT = kT + 2T$. Tada prethodni izraz postaje

$$y[kT] = -a_1 y[kT-T] - a_0 y[kT-2T] + b_2 u[kT] + b_1 u[kT-T] + b_0 u[kT-2T].$$

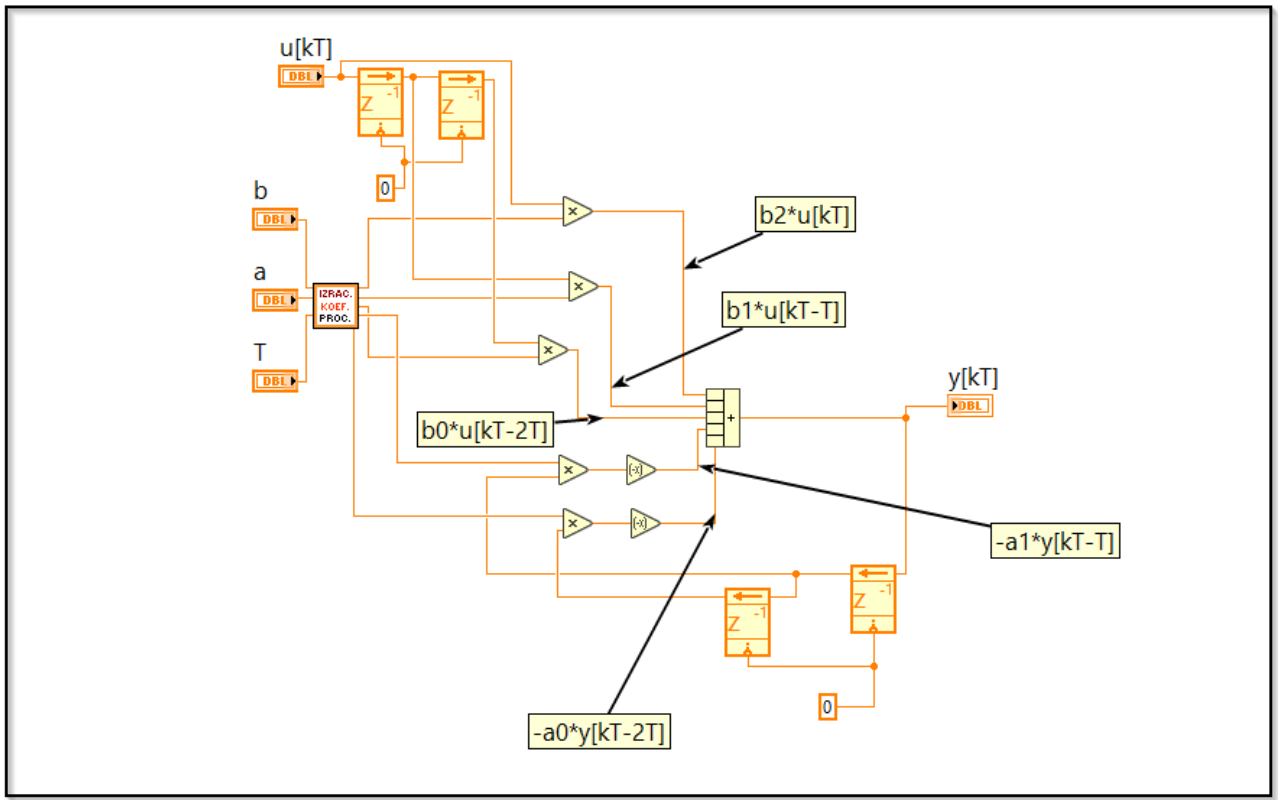
Dobijeni izraz je pogodan za implementaciju na digitalnom uređaju. Implementacija *subVI*-a koji simulira proces prikazana je na Slici 120. Logika koja se koristi pri implementaciji identična je logici koja je opisana pri implementaciji PID regulatora, stoga ovde neće biti detaljno opisana. Slično kao i kod implementacije PID-a korisno je napraviti pomoćni *subVI* koji izračunava koeficijente a_0, a_1, b_0, b_1, b_2. Njegova implementacija prikazana je na Slici 119 Ulazi u *subVI* koji opisuje proces su signal upravljanja i vreme odabiranja, ali i parametri a i b uz pomoć kojih možemo izabrati određeni proces iz klase procesa sa funkcijom prenosa definisanom izrazom (16). Izlaz je vrednost izlaza procesa koji simuliramo.



Slika 119: Prikaz izračunavanja pomoćnih koeficijenata za implementaciju diskretizovanog modela procesa.

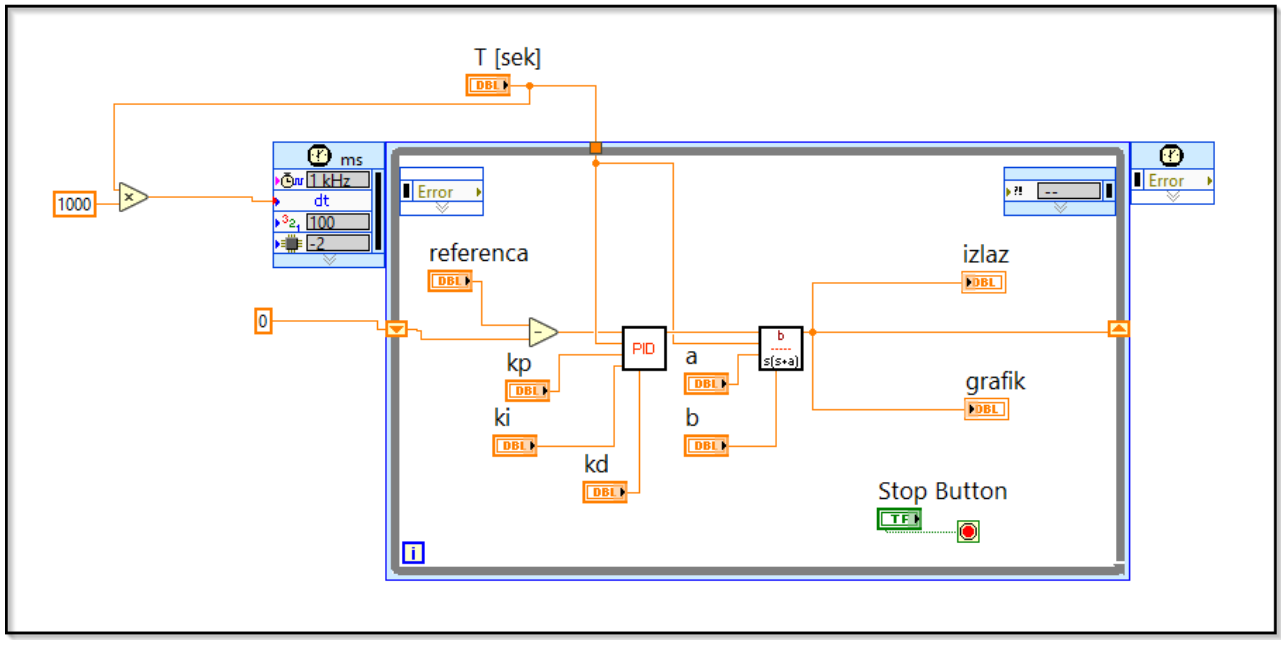
Na kraju, potrebno je implementirati glavnu simulacionu petlju koja će simulirati čitav sistem automatskog upravljanja. U ovom primeru uzećemo da su vremena odabiranja procesa i PID regulatora identična. Ukoliko želimo precizno da definišemo vreme izvršavanja, pogodno je korisiti *Timed Loop* strukturu. Na njen ulaz koji definiše interval izvršavanja potrebno je povezati vreme odabiranja T koje se može implementirati kao kontrola, tako da se sa *Front Panel*-a može promeniti vrednost. Potrebno je paziti na jedinice! Unutar same pelje potrebno je dodati s*subVI* koji implementira PID regulator, kao i *subVI* koji služi za simulaciju procesa. Naravno signal upravljanja koji je izlaz regulatora

povezuje se na ulaz procesa. Takođe potrebno je povezati sve konstante. Za PID su to k_p, k_i, k_d, kao i vreme odabiranja T. Za proces su to b, a i T. Sve navedene konstante se mogu implementirati kao kontrole, radi jednostavne promene sistema koji simuliramo. Konačno potrebno je definisati signal greške, koji je u principu razlika željene i ostvarene vrednosti. Željenu vrednost implementiramo kao kontrolu dok je ostvarena vrednost izlaz iz procesa koju uz pomoć *Shift* registra prebacujemo u narednu iteraciju. Izlaz procesa prikazan je na indikatoru kao i na grafiku koji je moguće dodati sa *Front Panel*-a tako što se doda *Waveform Chart* iz palete *Graph*. Implementacija glavne petlje data je na Slici 121, dok je izgled *Front Panel*-a, kao i primer jedog pokretanja simulacije prikazan na Slici 122.

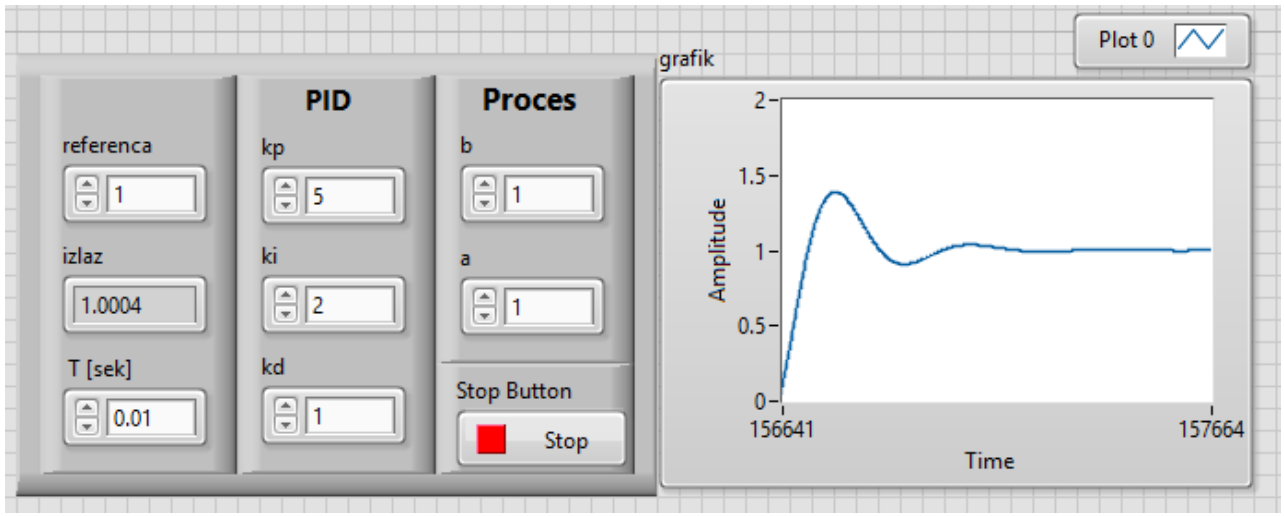


Slika 120: Implementacija diskretizovanog modela procesa upotrebom Tustinove aproksimacije.

Na Slici su prikazani parametri sistema koji se simulira. Ukoliko se uporedi odziv koji je prikazan na grafiku sa odzivom identičnog kontinualnog sistema videće se da razlika skora da i ne postoji.



Slika 121: Prikaz implementacije glavne petlje za simulaciju.



Slika 122: Prikaz *Front Panel*-a simulacije.

Dvoslojna aplikacija za akviziciju i upravljanje - LabVI-EW i cRIO

U ovom poglavlju prikazan je primer razvoja dvoslojne aplikacije za nadzor i upravljanje maketom *FeedBack Pressure Process Rig 38-714*. Dvoslojna aplikacija podrazumeva da postoji deo koji se izvršava na računaru (PC-u) i *National Instruments cRIO* kontroleru. Deo aplikacije koji se izvršava na PC-u predstavlja nadzorno-upravljački interfejs ka korisniku. Ukratko je opisan korišćeni hardver, maketa i *cRIO* kontroler, dodavanje kontrolera u projekat, dok je implementacija same aplikacije detaljnije opisana.

FeedBack Pressure Process Rig 38-714

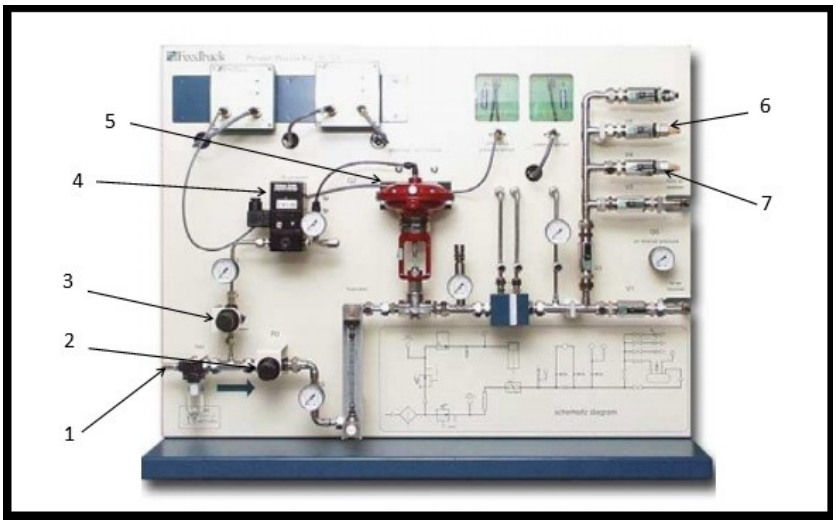
Izgled korišćene makete, *FeedBack Pressure Process Rig 38-714*, prikazan je na slici 123. Vazduh se u sistem dovodi iz kompresora dovodnim vodom. Deo vazduha odlazi u upravljački vod, ka IP pretvaraču, a ostatak odlazi u radni vod, koji vodi vazduh do ventila 6 i 7. Pritisak vazduha u upravljačkom i u radnom vodu može se podešavati ručnim regulatorima pritiska 2 i 3. IP pretvarač na osnovu zadatog upravljanja, koje se iz upravljačkog uređaja dovodi u vidu standardnog strujnog signala u opsegu od 4 do 20 mA propušta određenu količinu vazduha ka pneumatskom servo ventilu 5, odnosno vrši pretvaranje strujnog u pneumatski signal. U ventilu 5 se nalazi membrana koja se pod uticajem dovedenog vazduha pomera, delujući na polugu igličastog ventila i time menja vrednost pritiska u radnom vodu.

National Instruments cRIO 9024

Kontroler koji je korišćen u ovom primeru je *National Instruments cRIO 9024*. *cRIO* predstavlja ugrađen (*embedded real-time*) kontroler namenjen aplikacijama za upravljanje i nadzor sistema. Kontroler poseduje procesor brzine 800 MHz, 512 MB dinamičke memorije, kao i 4GB memorije za trajno skladištenje informacija. Raspolaže različitim vrstama portova, kao što su dva *Ethernet* ulaza, jedan *USB* i jedan serijski

Delovi makete sa slike 123:

1. dovodni vod,
2. ručni regulator pritiska,
3. ručni regulator pritiska,
4. IP pretvarač,
5. pneumatski servo ventil i,
6. ventili.

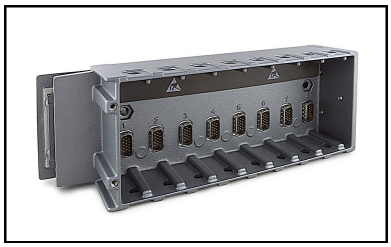


Slika 123: Izgled makete FeedBack Pressure Process Rig 38-714.

ulaz. *cRIO 9024* ne dolazi sa integrisanim *I/O* (*Input/Output*) modulima, tako da je moguće kombinovati ovaj kontroler sa kompatibilnim *cRIO* šasijama.

Izgled korišćenog kontrolera prikazan je na slici , dok je korišćena šasija pokazana na slici . Na pomenutu šasiju postavljeni su moduli *NI9203*, koji sadrži samo analogne ulaze, i *NI9265*, koji poseduje samo analogne izlaze.

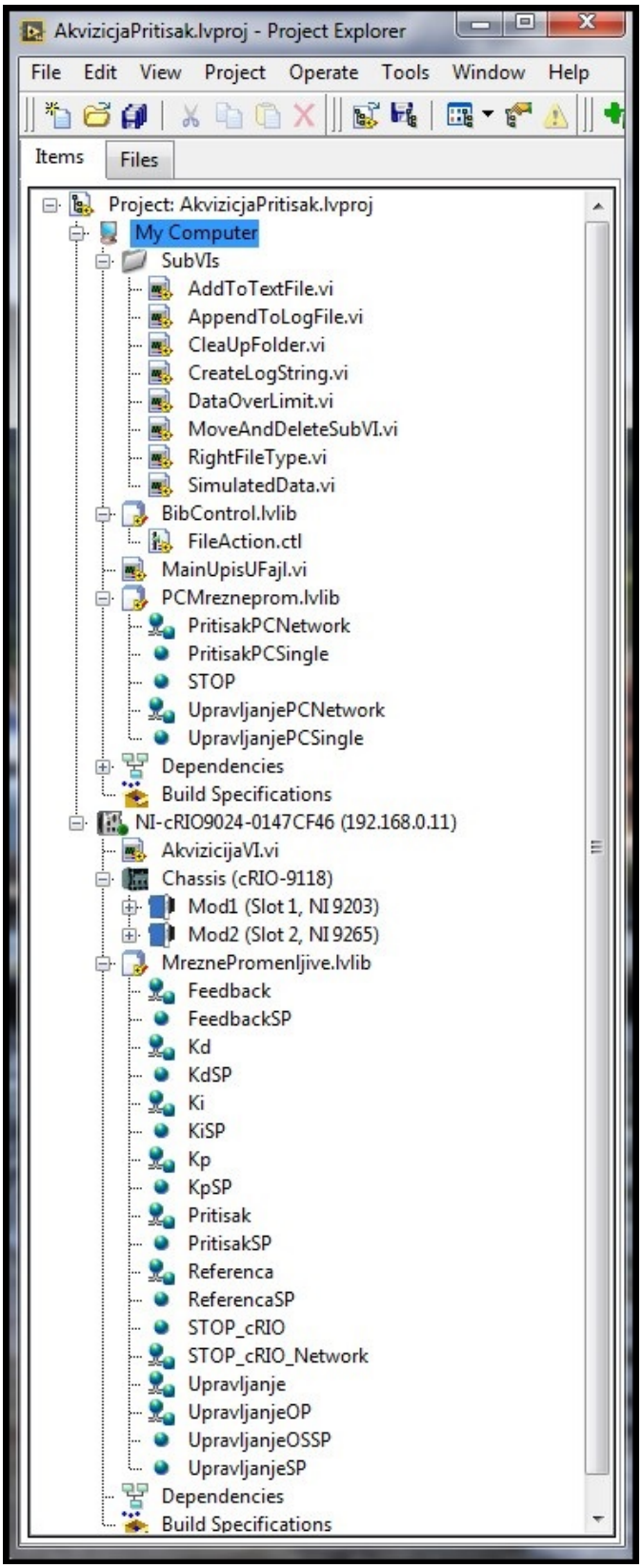
Slika 124: Izgled *National Instruments cRIO 9024* kontrolera.



Slika 125: Izgled šasije *NI9118*.

Konfiguracija cRIO kontrolera

Pre same implementacije aplikacije potrebno je proveriti da li je kontroler uspešno povezan sa računarom preko *Ethernet*-a, na istoj mreži. Nakon što smo obezbedili napajanje i povezali kontroler sa računarom, potrebno je pokrenuti program *NI MAX*. Ukoliko je uređaj uspešno povezan, on će se pojaviti u okviru sekcije *Remote Systems*. Uređaj je moguće dodati i ručno ukoliko je poznata IP adresa ili ime *Host*-a. Kada je uređaj uspešno povezan može se započeti sa kreiranjem projekta. Potrebno je kreirati novi projekat i potom desnim klikom na naziv projekta dodati *cRIO* kontroler u projekat (New → Targets and Devices). Uređaj je moguće dodati i ručno ukoliko je poznata njegova *IP* adresa i nalazi se u istoj mreži kao i računar. Izgled projektnog stabla moguće je videti na slici 126.Virtuelni instrumenti koji treba da se izvršavaju na PC delu nalaze se ispod *My Computer*, dok VI koji je potrebno da se izvršava na kontroleru se nalazi ispod naziva uređaja. Ispod naziva kontrolera se takođe vide i analogni moduli koji su povezani na šasiju.



Slika 126: Izgled projektnog stabla.

Implementacija dvoslojne aplikacije za akviziciju i upravljanje

Deo aplikacije na cRIO kontroleru

Na slikama 128 i 129 prikazan je blok dijagram virtuelnog instrumenta koji se izvršava na samom kontroleru. Ovde se vrši upravljanje pritiskom, i to je moguće upravljanje u otvorenoj sprezi ili upravljanje u zatvorenoj povratnoj sprezi pomoću PID regulatora (slika 128) i komunikacija sa PC delom aplikacije (slika 129). Izmerena vrednost pritiska se očitava sa analognog ulaza *AIo*, dok se upravljački signal šalje na analogni izlaz *AOo*. Da bi komunikacija između dva sloja bila moguća potrebno je koristiti deljene promenljive (*Shared Variables*), i to dva tipa varijabli - *Single Process* i *Network Published* deljene promenljive. *Single Process* deljenje promenljive se koriste obično u okviru istog VI-a kada je potrebno transportovati podatke, a ne mogu se direktno povezati žicama, kao npr. što je slučaj u ovom zadatku, kada imamo dve paralelne petlje. *Network Published* deljene variable se koriste za čitanje i pisanje podatka u okviru čitave mreže. Dakle, ideja za podatke koji dolaze sa kontrolera i treba da budu dostupni na *PC* delu je sledeća: da se izmerena vrednost pritiska upiše u *Signle Process* deljenu promeljivu (nazavnu *PritisakSP*) u gornjoj petlji, a da se u donjoj petlji vrednost iz *PritisakSP* prepiše u *Network Shared* promenljivu nazavnu *Pritisak*, kako bi se ta vrednost bila dostupna za korišćenje na *PC* delu aplikacije. Što se tiče podataka koji treba da dođu kao informacija sa korisničkog interfejsa, referentna vrednost pritiska u mA, parametri regulatora (*Kp* - proporcionalno dejstvo, *Ki* - integralno dejstvo i *Kd* - difencijalno dejstvo regulatora), da li je upravljanje u otvorenoj ili zatvorenoj sprezi (ukoliko je otvorena sprega u pitanju potrebno je proslediti i konkretnu vrednost upravljanja), sve vrednosti se direktno upisuju u *Network Shared* promenljive i onda se iz njih prepisuje u *Single Process* promenljive na *real-time* nivou aplikacije i tek onda se koriste u programu.

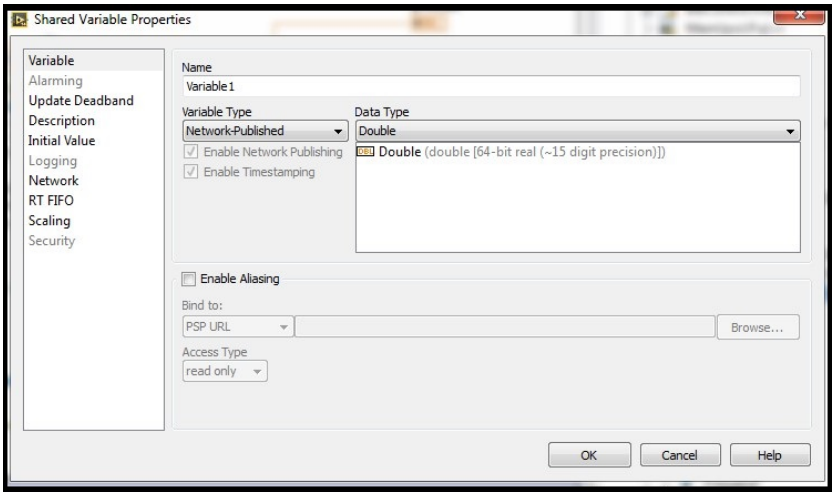
Da bi se kreirala deljena promenljiva, prvo je potrebno kreirati novu biblioteku u okviru *PC* dela i *real-time* dela aplikacije desnim klikom miša i izborom *New* → *Library*. Potom se u biblioteci kreira željena promenljiva desnim klikom miša na biblioteku i izborom *New* → *Variable*. Time se dobija prozor prikazan na slici 127, gde je moguće podesitit parametre promenljive.

Za upravljanje u zatvorenoj sprezi koristi se PID regulator koji je ovde implementiran kao *Matlab* skripta, a pseudo kod koji opisuje rad regulatora je sledeći:

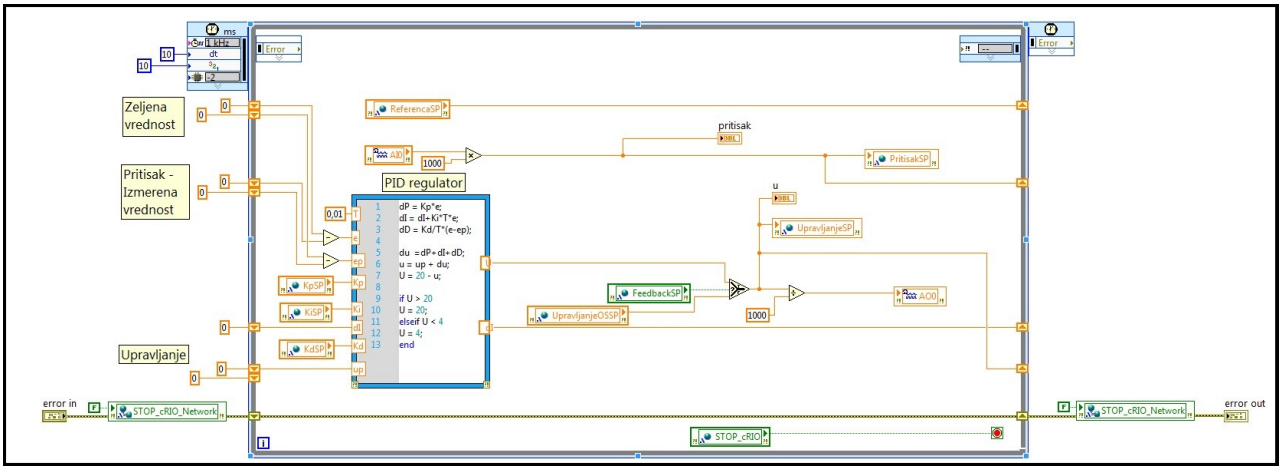
```
dP = Kp*e;
dI = dI+Ki*T*e;
dD = Kd/T*(e-ep);
```

```
du =dP+dI+dD;
u = up + du;
U = 20 - u;
if U > 20
   U = 20;
elseif U < 4
   U = 4;
```

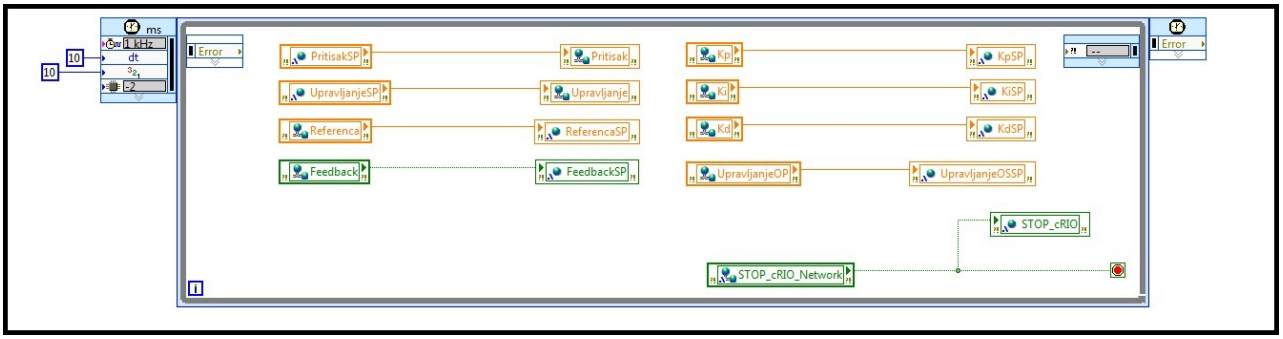
Prilikom korišćenja *Matlab* skripte potrebno je dodati ulaze koji se koriste u skripti i to je moguće dodati desnim klikom miša na skriptu, odabirom opcije *Add Input* i upisati ime promenljive. Slično važi i za izlaze. Da bi omogućili rad PID regulatora za određene promenljive moraju se koristiti *shift* registri, jer je potrebno zapamtiti i prethodne vrednosti promenljive. U ovom slučaju su bitne vrednosti reference, izmerene vrednosti pritiska, upravljanja u zatvorenoj sprezi i integralnog dejstva.



Slika 127: Kreiranje deljene promenljive.



Slika 128: Blok dijagram VI-a koji se izvršava na *cRIO* kontroleru.



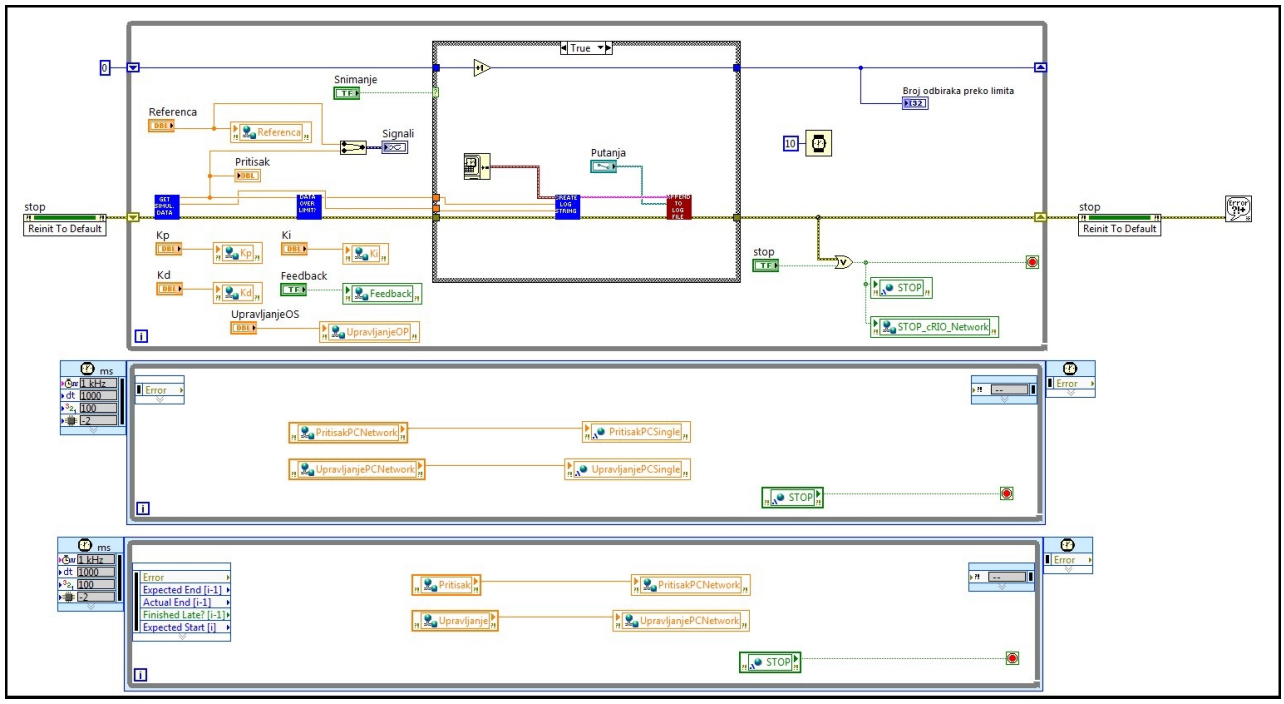
Slika 129: Blok dijagram VI-a koji se izvršava na *cRIO* kontroleru.

PC deo aplikacije - korisnički interfejs

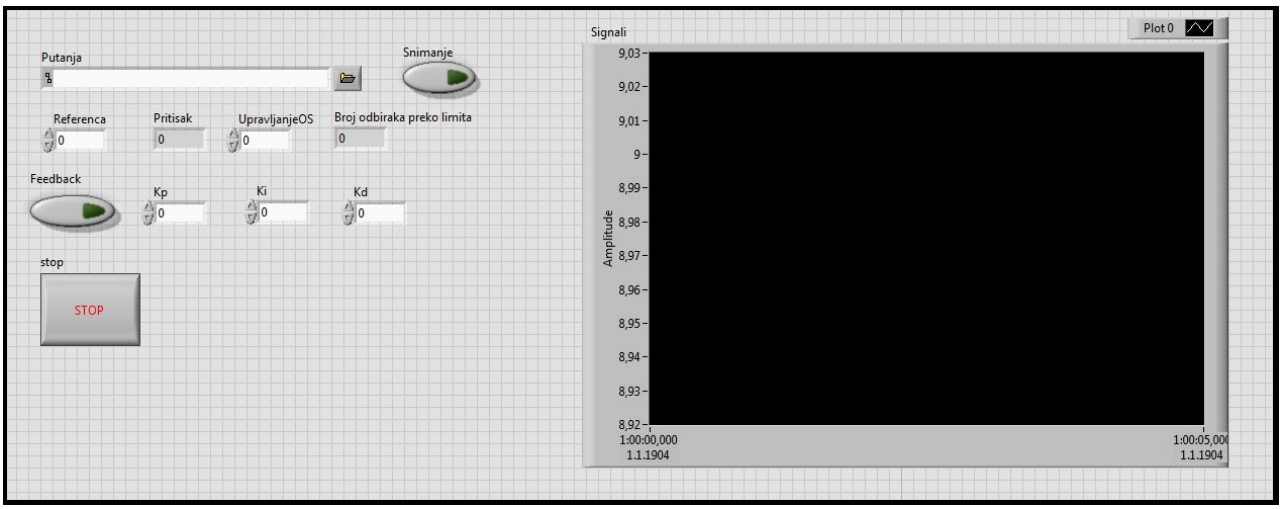
Osim već navedenih promenljivih koje postoje u programu i koje se mogu uneti i očitati sa korisničkog interfejsa, ideja za ovu aplikaciju je bila da se omogući i logovanje podataka u *.txt* fajl u formatu *timestamp*, izmerena vrednost pritiska i upravljanje (slika). Da bi se omogućilo logovanje podataka potrebno je uraditi niz operacija, koje su već opisane u drugom poglavlju, te ovde nisu ponovo objašnjene, već se kodovi mogu videti na slikama od 133 do 135.

Korisnik može da upravlja sistemom i bez opcije logovanja podataka, ali ukoliko želi da snimi podatke potrebno je da prvo odabere putanju gde želi da snimi podatke i pritisne dugme *Snimanje*. Takođe, na prednjem panelu korisnik ima i grafički prikaz oba signala na jednom grafiku - referenta i izmerena vrednost pritiska.

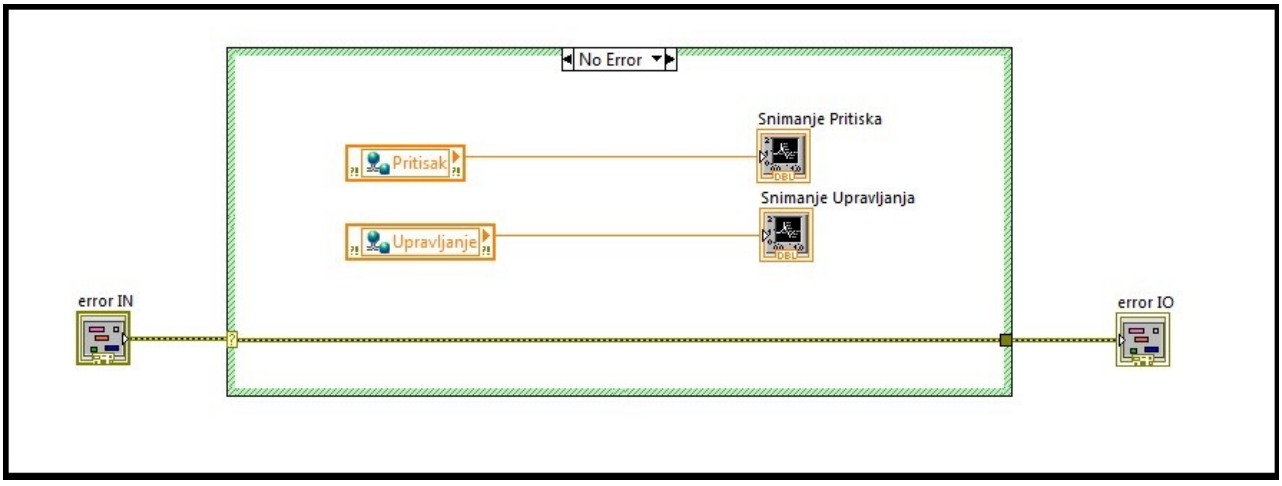
U bilo kom trenutku korisnik može zaustaviti aplikaciju pritiskom na dugme *STOP*. Bitno je napomenuti da kada korisnik zaustavi aplikaciju iz ovog prozora da se zaustavi i deo aplikacije koji se izvršava na kontroleru. Zbog toga su iskorišćene deljive promenljive. Kada korisnik pritisne dugme *STOP* ili se desi greška u programu, ta vrednost se upisuje u *Network Published* promenljivu, koja se na *real-time* nivou prepisuje u *Single Process* deljenu promenljivu i onda se ona koristi da potpuno zaustavi aplikaciju na kontroleru.



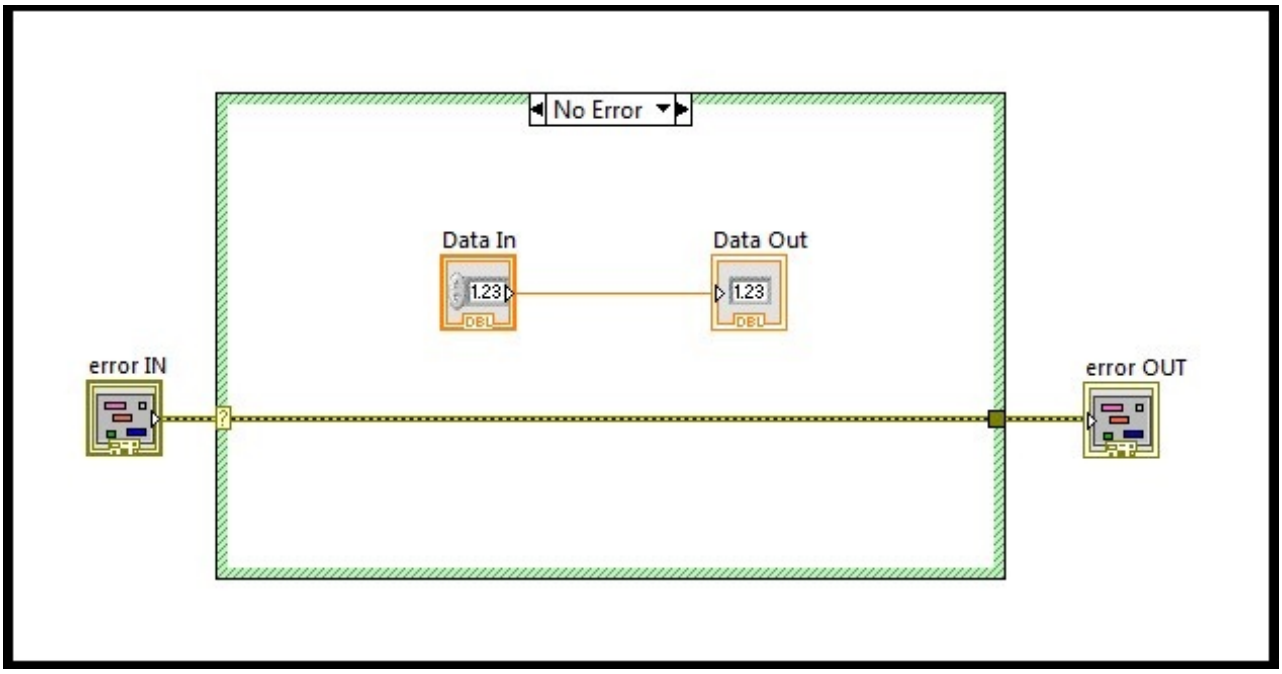
Slika 130: Blok dijagram korisničkog interfejsa.



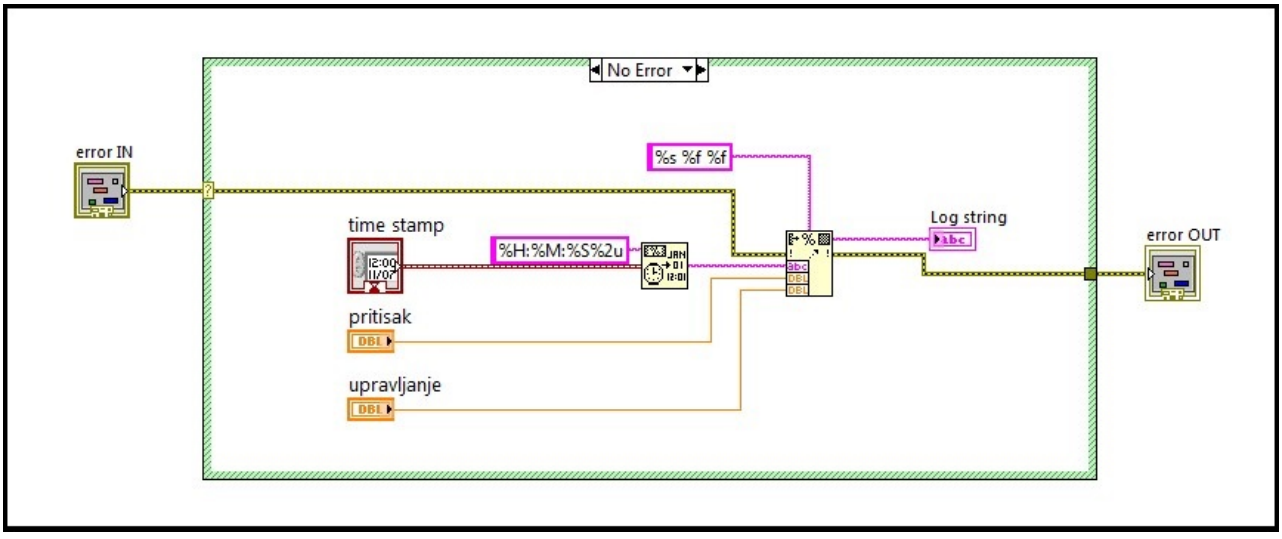
Slika 131: Izgled korisničkog interfejsa.



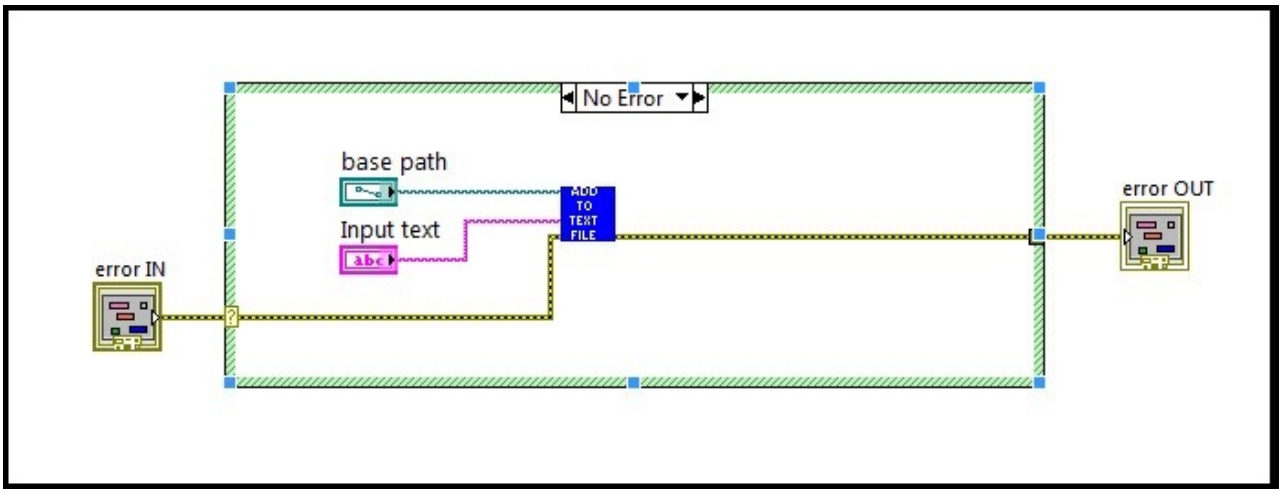
Slika 132: Blok dijagrama SimulatedData.vi



Slika 133: Blok dijagrama DataOverLimit.vi



Slika 134: Blok dijagrama CreateLogString.vi



Slika 135: Blok dijagrama AppendTo-LogFile.vi

```
10:47:03,34 21,401903 5,000000
10:47:03,35 21,401903 5,000000
10:47:03,36 21,401903 5,000000
10:47:03,38 21,401903 5,000000
10:47:03,39 21,401903 5,000000
10:47:03,39 21,401903 5,000000
10:47:03,41 21,401903 5,000000
10:47:03,42 21,401903 5,000000
10:47:03,43 21,401903 5,000000
10:47:03,44 21,401903 5,000000
10:47:03,45 21,401903 5,000000
10:47:03,46 21,401903 5,000000
10:47:03,47 21,401903 9,000000
10:47:03,48 21,401903 9,000000
10:47:03,49 21,401903 9,000000
10:47:03,50 21,401903 9,000000
10:47:03,51 21,401903 9,000000
10:47:03,52 21,401903 9,000000
10:47:03,53 21,401903 9,000000
10:47:03,54 21,401903 9,000000
10:47:03,55 21,401903 9,000000
10:47:03,56 21,401903 9,000000
10:47:03,57 21,401903 9,000000
10:47:03,58 21,401903 9,000000
10:47:03,59 21,401903 9,000000
10:47:03,60 21,401903 9,000000
10:47:03,61 21,401903 9,000000
10:47:03,62 21,401903 9,000000
10:47:03,63 21,401903 9,000000
10:47:03,64 21,401903 9,000000
10:47:03,65 21,401903 9,000000
10:47:03,66 21,401903 9,000000
10:47:03,67 21,401903 9,000000
10:47:03,68 21,401903 9,000000
```

Slika 136: Izgled log fajla.

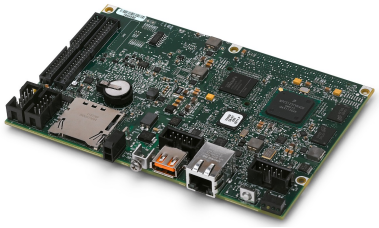
Troslojna aplikacija za akviziciju i upravljanje-LabView, sbRio

U ovom poglavlju prikazan je primer razvoja troslojne aplikacije koja podrazumeva deo koji se izvšavana na PC-u kao i delove koji se izvšavju na *National Instruments sbRio* kontroleru, u nastavku *sbRio*. Svakako, ovakva aplikacija može da se realizuje i na drugim platformama koje imaju mikroprocesorski deo, kao i FPGA čip [7]. Deo aplikacije koji se izvršava na PC-u predstavlja nadzorno-upravljački intrfejs ka korisniku. Sa druge strane, *sbRio* se sastoji od *Real-Time* procesora i *FPGA* pločice, pa na njemu postoje još dva dela aplikacije. Na *FPGA* ploči se izvršava akvizicija signala, dok se na *Real-Time* procesoru izvšava logički deo aplikacije. Ukratko je opisan *sbRio* kontroler i date su neke njegove osnovne karakteristike. Postupno je opisano kofigurisanje samog kontrolera, kao i konfiguracija projekta, odnosno dodavanje *sbRio* kontrolera u projekat. Konačno, detaljno je opisana implementacija aplikacije za nadzor i upravljanje.

[7] National Instruments. *CompactRIO Developers Guide*. National Instruments, May 2009

National Instruments sbRio

Kontroler koji se koristi u ovom projektu je *National Instruments sbRio 9636*. Ovaj kontroler pripada familiji *Compact Rio* kontrolera, ali su mu *Real-Time* procesor i *FPGA* na istom čipu, pa zbog toga ima oznaku *Single-Board*. Izled *sbRio* kontrolera prikazan je na Slici 137.

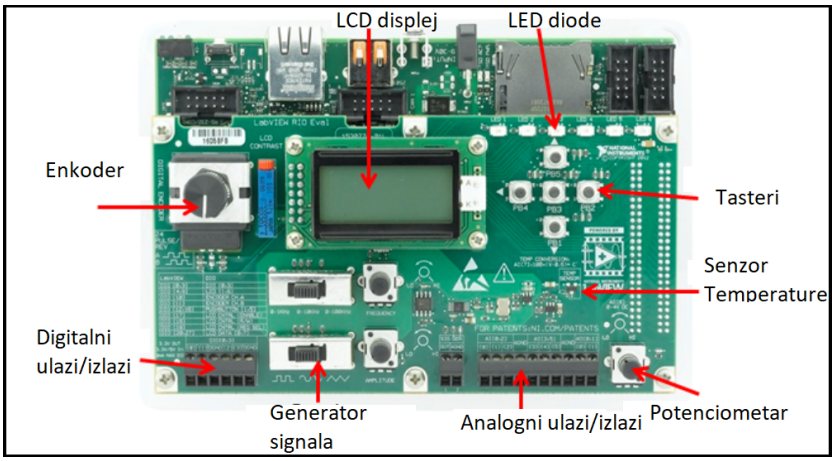


Slika 137: Prikaz *sbRio* 9636 kontrolera.

Kontroler je dizajneran tako da bude fleksibilan, pouzdan i da ima dobre performanse. Stoga, *sbRio 9636* poseduje procesor brzine 400 MHz, 256MB dinamičke RAM memorije, kao i 512MB memorije za trajno skladištenje informacija. Kao što je već rečeno *sbRio* poseduje *FPGA*, u konkretnom slučaju to je *Spartan-6 LX45*. Kontroler podržava povezivanje putem *USB, CAN* i *Ethernet* protokola. Takođe kontroler ima i niz analognih i digitalnih ulaznih i izlaznih pinova, o čemu će biti reči u nastavku. Kompletnu specifikaciju moguće je pronaći na oficijalnom sajtu *National Instruments* kompanije.

U ovom primeru zajedno sa *sbRio* kontrolerom koristi se *National Instruments LabView Rio Eval Kit*, koji pedstavlja spoj perifernih kompo-

nenti. Na Slici 138 prikazan je izgled ovog pomoćnog alata. Na njemu se nalazi 6 LED dioda, 5 tastera, LCD displej, enkoder, senzor temperature, potenciometar i signal generator. Pored navedenih komponenti, na pločici postoje konektori koji se mogu iskoristi za povezivanje dodatnih analognih ili digitalnih uređaja.



Slika 138: Prikaz *National Instruments LabView Rio Eval Kit* pločice.

Da bi se moglo rukovati komponentama koje se nalaze na proširenju potrebno je znati na koje pinove su povezani sami uređaji. U Tabeli 1 prikazan je način povezanosti komponenti sa *Eval Kit*-a na *sbRio* kontroler.

sbRio pin	Komponenta na *Eval Kit-u*
DIO [0:3]	DIO [0:3]
DIO [4:9]	LED [1:6]
DIO [10]	Enkoder, kanal A.
DIO [11]	Enkoder, kanal B.
DIO [12:16]	Taster [1:5].
DIO [17]	LCD CNTRL (ENABLE)
DIO [18]	LCD CNTRL (R/W)
DIO [19]	LCD CNTRL (REG SEL)
DIO [20:27]	LCD DATA [0:7]
AI [0:5]	AI [0:5]
AO [0:1]	AO [0:1]
AI 6	Potenciometar
AI 7	Senzor temperature

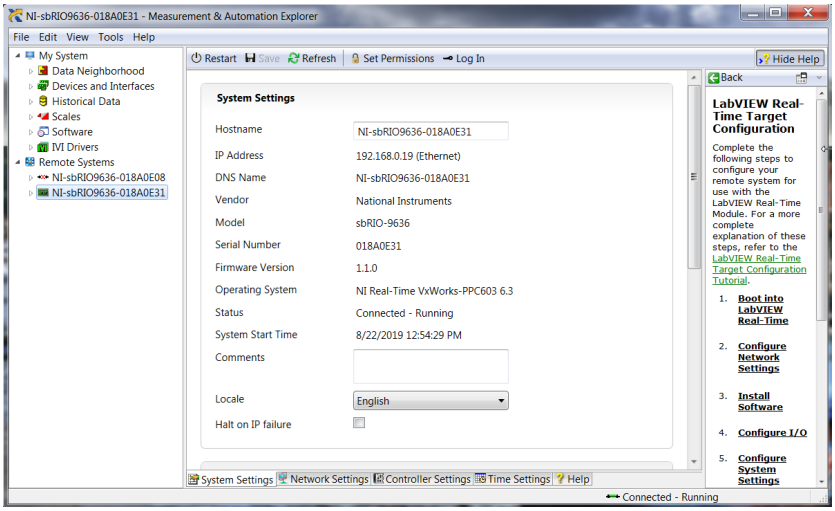
Tabela 1: Prikaz međusobne povezanosti komponenti *Eval Kit*-a i pinova *sbRio* kontrolera.

Tabela će biti od velikog značaja prilikom implementacije *vi*-a čiji će zadatak biti prikupljanje informacija sa kontrolera, kao i slanje informacija sa kontrolera na eksterne komponente. Potrebno je naglasiti da *DIO [0:3]* na *Eval Kit*-u predstavlja konektor na koji je moguće povezati eksterne uređaje koji nisu deo samog *Eval Kit*-a. Isto važi i za *AI [0:5]* i *AO [0:1]*. Takođe, bitno je naglasiti da se temperatura dobija kao $100(V - 0.5)$, gde V predstavlja izmerenu vrednost koju sentor daje. Na taj način dobija se temperatura u stepenima Celzijusa. Još je bitno naglasiti da signal generator nije direktno povezan na *sbRio*, pa ga je potrebno povezati na neki od analognih ulaza koji su navedeni ranije.

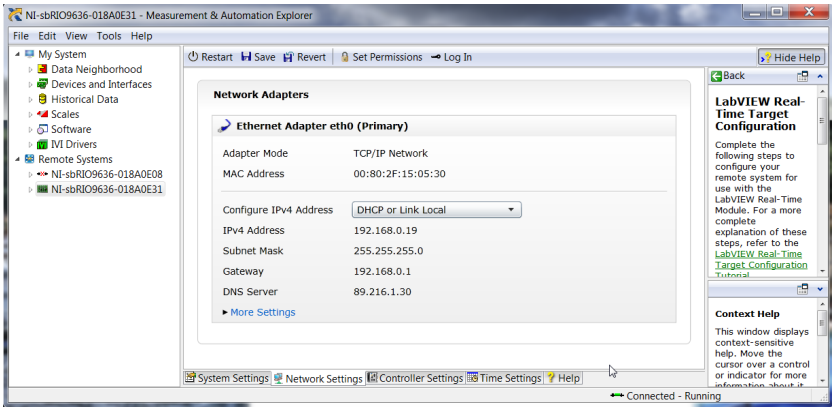
Konfiguracija sbRio kontrolera

Kao i kod bilo kojeg drugog *National Instruments* hardvera, pre početka implementacije softvera potrebno je proveriti da li je uređaj uspešno povezan. U opisu samog uređaja navedeno je da se *sbRio* može povezati putem *Ethernet*-a. Dakle potrebno je povezati na istu mrežu računar i *sbRio* kontroler. Pored mrežnog povezivanja, potrebno je obezbediti napajanje za *sbRio*. Kada je uređaj povezan, potrebno je proveriti podešavanja u okviru *NI MAX*-a. Svi kontroleri koji su na istoj

mreži trebali bi se pojaviti u okviru sekcije *Remote Systems*, kao što se vidi na Slici 139. Pored uređaja koji su trenutno na mreži, pojaće se i uređaji koji koji su ranije korišteni ali trenutno nisu dostupni. Ukoliko se uređaj ne pojavi, moguće je dodati ga ručno, uz uslov da je poznata njegova *IP* adresa ili ime *Host*-a. Ukoliko je uređaj uspešno povezan, trebalo bi da se dobije sličan prozor kao na Slici 139. Ukoliko u polju *Status* piše da je uređaj povezan i pokrenut, sve je u redu. U ovom prozoru se mogu videti neke opcije koje karakterišu uređaj. Može se videti ime *Host*-a koje je moguće promeniti. Takođe vidljiva je *IP* adresa, verzija firmvera i tako dalje.



Slika 139: Prikaz izgleda *NI MAX* okruženja prilikom rukovanja *NI sbRio* uređajem.

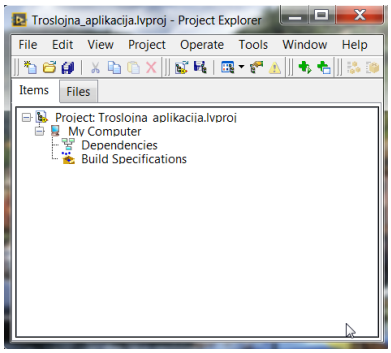


Slika 140: Prikaz izgleda *NI MAX* okruženja prilikom rukovanja mrežnim podešavanjima *NI sbRio* uređaja.

Ukoliko se prebaci na karticu *Network settings* otvaraju se mrežna podešavanja uređaja. Kao što se vidi na Slici 140, ovde je moguće videti *MAC* adresu uređaja i podešavanja vezana za *IP* adresu, gde se može izabrati način na koji je definisana *IP* adresa, Pored *DHCP* na-

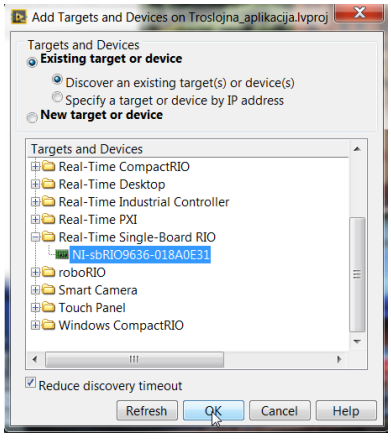
čina određivanja adrese, moguće je podesiti statičku *IP* adresu, čime se dobija mogućnost definisanja *IP* adrese po sopstvenoj želji. Jasno je da se računar i kontroler moraju nalaziti na istoj podmreži. Na slici se može uočiti da je moguće resetovati kontroler iz *NI MAX*-a. Pored toga u kartici *Controller settings* moguće je izvršiti reset *IP* adrese na podrazumevanu vrednost, prebaciti kontroler u *Safe* mod rada i isključiti neke procese koji se pokreću prilikom uključenja uređaja.

Kada se kontroler korektno podesi u okviru *NI MAX*-a, može se početi sa konfiguracijom projekta koji sadrži deo koji treba da se izvršava na *sbRio* kontroleru. Kada se kreira novi projekat u okviru *LabView* okruženja, stablo projekta ima izgled kao na Slici 141. Na slici se vidi da postoji samo jedan uređaj na koji je moguće dodavati funkcionalnosti, a to je samo računar na kome smo upravo kreirali projekat.



Slika 141: Prikaz stabla projekta neposredno nakon kreiranja.

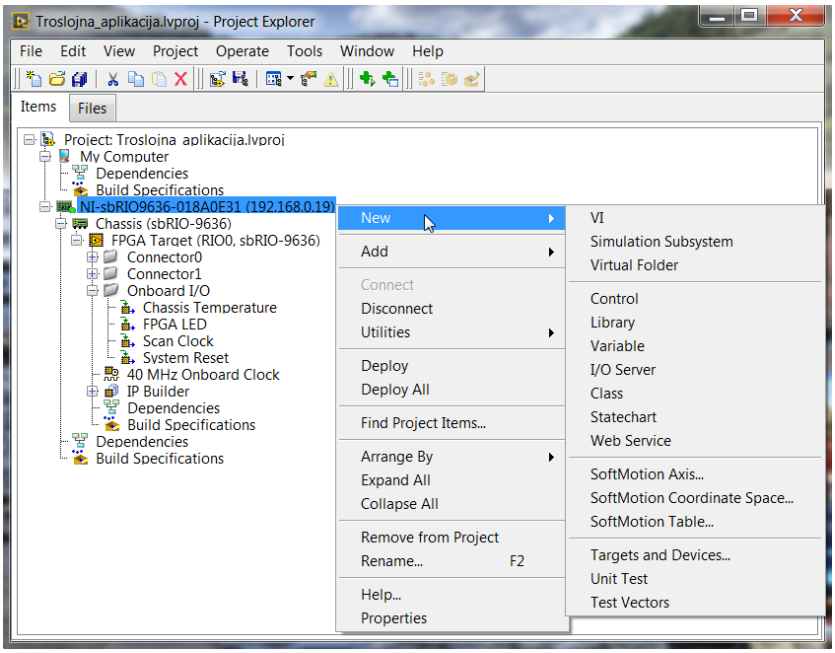
Jasno je da *sbRio* predstavlja drugi uređaj, pa ga je potrebno dodati. To je moguće izvšiti tako što se u okviru stabla projekta desnim klikom na samo ime projekta otvori set opcija. Tu je potrebno izabrati opciju *new Targets and Devices*. Kada se izabere ova opcija, otvara se novi prozor koji je prikazan na Slici 142. Sada je potrebno odabrati koji uređaj se dodaje u projekat. Najjednostavniji način je izbor automatske dekcije uređaja koji su povezani, kao što je prikazano na slici. Tada je potrebno samo odabrati odgovarajući folder koji predstavlja tip uređaja, što je u ovom slučaju *Real-Time Single-Board Rio*. Unutar foldera trebalo bi da se pojave svi uređaji koji pripadaju ovoj porodici, a uspešno su povezani na mrežu (što podrazumeva da je sve provereno u *NI MAX*-u). Tada jednostavnim izborom odgovarajućeg uređaja i potvrdom dodaje se uređaj u projekat. Ukoliko nije moguće automatski pronaći uređaj, uređaj je moguće dodati direktnim unosom njegove *IP* adrese, koju je opet moguće proveriti u *NI MAX*-u. Naravno, jasno je da računar mora biti u istoj podmreži kao i *sbRio*, inače dodavanje nije moguće. Kada se dodavanje uspešno izvši, stablo projekta dobija izgled kao na Slici 143. Paralelno sa *PC* uređajem pojavljuje se odabrani *sbRio*. Uočljivo je da unutar šasije postoji takozvani *FPGA Target* koji zapravo predstavlja *FPGA* pločicu. Može se primetiti da u ovom slučaju *FPGA* poseduje dva konektora koji sadrže digitalni i analogne ulaze i izlaze.



Slika 142: Prikaz menija čija je svrha dodavanje novog uređaja u projekat.

Ukoliko se otvore opcije nad samim imenom *sbRio* uređaja, može se dobiti niz opcija kao na Slici 143. Kao što se vidi na slici, na ovaj način moguće se povezati sa uređajem. Pri konekciji, svi fajlovi koji su vezani za *sbRio* prebacuju se na sam kontroler. Fajlove je moguće prebaciti i uz pomoć komandi *Deploy* i *Deploy all*. Takođe, iz menija je moguće dodati nove fajlove i foldere u projekat, kao i kreirati nove. Bitno je naglasiti da fajlovi koji se dodaju iz menija kao što je prikazano na slici svoju funkcionalnost izvršavaju na *Real-Time* procesoru *sbRio* kontrolera. Sa druge strane, iz sličnog menija u okviru *FPGA* sekcije moguće je

manipulisati fajlovima čija se funkcionalnost ispunjava na *FPGA* čipu. U nastavku poglavlja prikazana je implementacija određenih funkcionalnosti čija je destinacija *Real-Time* procesor, kao i funkcionalnosti čija je destinacija *FPGA* čip.



Slika 143: Prikaz stabla projekta nakon dodavanja *sbRio* kotrolera.

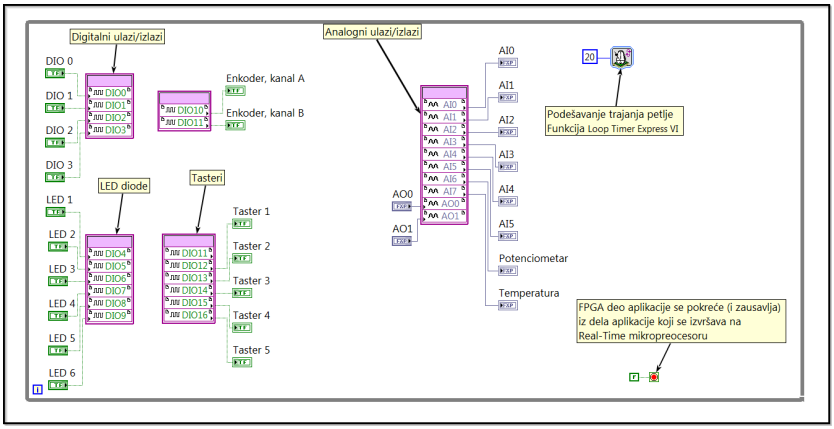
Implementacija troslojne aplikacije za akviziciju i upravljanje

Troslojna aplikacija za akviziciju i upravljanje sastoji se iz dela koji se izvršava na *FPGA*, dela koji se izvršava na *Real-Time* procesoru *sbRio* kontrolera i *PC* dela aplikacije koji predstavlja korisnički interfejs. *FPGA* deo aplikacije je zadužen za komunikaciju sa *Eval Kit*-om, tj. *Eval Kit* je direktno povezan na pinove *sbRio*-a koji se kontrolišu preko *FPGA* čipa. Cilj aplikacije koja je opisana u nastavku je da prikupi informacije o sa *Eval Kit*-a i prikaže ih na korisničkom interfejsu koji se izvršava na računaru. Takođe potrebno je da prosledi komande sa korisničkog interfejsa na *Eval Kit*. Komande uglavnom podrazumevaju uključenje i isključenje LED dioda.

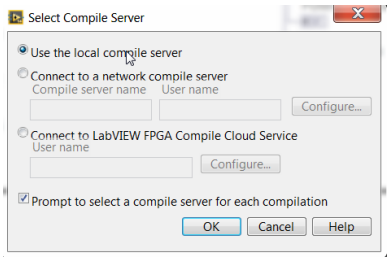
FPGA deo aplikacije

Prilikom razvoja aplikacija ovog tipa, najbolje je krenuti od najnižeg nivoa, odnosno od *FPGA* dela aplikacije. U ovom primeru *FPGA* ima jednostavan zadatak, treba samo da pročita informacije o stanju komponenti na *Eval Kit*-u, ali i da definiše vrednosti određenih kom-

ponenti, kao što su na primer LED diode. Implementacija navedene funkcionalnosti data je na Slici 144. Da bi se dodao *VI* koji je vezan za *FPGA* deo aplikacije potrebno ga je dodati u *FPGA Target* u stablu projekta.



Slika 144: Prikaz implementacije *FPGA* dela aplikacije.



Slika 145: Prikaz prozora za izbor servera za kompajliranje *FPGA* dela aplikacije.

Jasno da čitava funkcionalnost mora biti unutar *while* petlje da bi se ponavljala tokom vremena. Potrebno je kreirati kontrole i indikatore koji će predstavljati određene komponente. Ideja je da se izvan ovog *VI*-a jednostavno prepoznaje kojom komponentom se manipuliše.

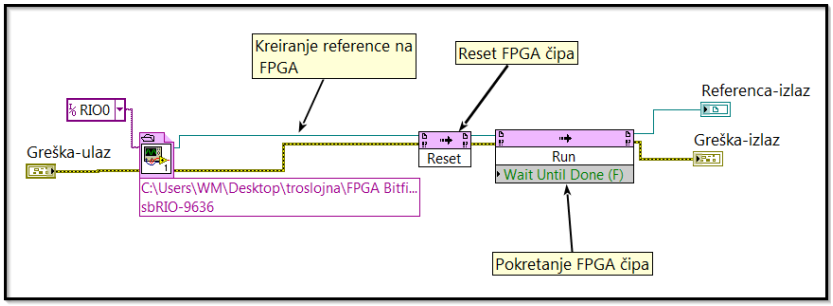
Jasno je da je sa stanovišta *FPGA* čipa stanje jedne od LED dioda ulaz, jer se ovo stanje treba menjati iz nekih spoljašnjih funkcionalnosti. Zbog toga se diode predstavljaju kontrolama. Sa druge strane, *FPGA* treba da prikupi informacije u stanju tastera na *Eval Kit*-u i prosledi ih ka nekim spoljnim funkcionalnostima, pa su zbog toga Tasteri predstavljeni indikatorima. Isto važi i za enkoder, temperaturu i potenciometar. Čitanje i pisanje se vrši pomoću funkcije *FPGA I/O Node*. U ovom trenutku potrebno se osvrnuti na Tabelu **??** u kojoj je definisan raspored povezivanja pinova *sbRio*-a i *Eval Kit*-a. Potrebno je samo ispratiti tabelu i povezati odgovarajuće kontrole/indikatore na odgovarajuće fizičke pinove. Pinovi koji su vezani za upravljanje LCD displejem nisu iskorišteni pošto rukovanje displejem je malo kompleksnije od čistog prosleđivanja u svakoj iteraciji, a to ne predstavlja deo ovog primera. Razdvojenost čvorova u implementaciji nema nikakav funkcionalni značaj, već samo estetski. Na *FPGA* nivou se samo prosleđuju sirovi podaci, dok je obrada podataka (kao što je na primer izračunavanje temperature u zavisnosti od naponskog nivoa) implementirana u okviru dela aplikacije koji se izvršava na *Real-Time* procesoru. Naravno, na ovaj način nije ispunjen puni potencijal koji poseduje *FPGA*, međutim ideja ovog primera je samo da pokaže način rukovanja *FPGA* čipom bez mnogo komplikacija. Bitno je definisati vreme izvršavanja jednog ciklusa petlje, što je izvršeno uz pomoć funkcije *Loop Timer Express VI*

sa parametrom 20 otkucaja. Petlja je konfigurisana kao beskonačna iz razloga što će se *FPGA* deo aplikacije isključivo pokretati (i zaustavljati) iz dela koji se izvršava na *Real-Time* procesoru. Po završetku implementacije, potrebno je kompajlirati implementiranu funkcionalnost. Najjednostavniji način je klikom na *Run* dugme, odnosno pokušajem pokretanja aplikacije. U tom slučaju otvoriće se prozor koji je prikazan na Slici 145. Potrebno je izabrati server na kome će se kompajlirati fajl. Postoji ocija za kompajliranje na lokalnom računaru na kome je implementiran fajl, kao i mogućnost kompajliranja negde na mreži ili pak *LabView Cloud* servisu. Po završetku kompajliranja, generisaće se *.lvbitx* fajl koji zapravo definiše konfiguraciju *FPGA* čipa. Ovaj fajl je smešten u folder *FPGA Bitfiles* i moguće je koristiti ga na drugim *FPGA* čipovima. Naravno, po završetku kompajliranja pokrenuće se kompajlirani fajl jer je kompajliranje pokrenuto klikom na dugme *Run* pa je moguće izvšiti neko inicijalno testiranje funkcionisanja. Kompajliranje je takođe moguće izvšiti definisanjem *Build* specifikacije u stablu projekta u sekciji vezanoj za *FPGA*.

Real-Time deo aplikacije

Nakon implementacije *FPGA* dela aplikacije, potrebno je razviti i *Real-Time* deo. Ovaj deo aplikacije obuhvata manipulisanje *FPGA* čipom, prilagođavanje podataka i razmenu podataka sa *PC* delom aplikacije. Ovaj deo aplikacije podeljen je u nekoliko *subVI*-eva radi bolje preglednosti.

Prvi u nizu je *subVI* koji vrši pokretanje *FPGA* dela aplikacije. Implementacija ove funkcionalnosti prikazana je na Slici 146. Ulaz u navedenu funkcionalnost je samo signal greške, dok su izlazi signal greške i referenca na *FPGA* deo aplikacije koji je pokrenut.



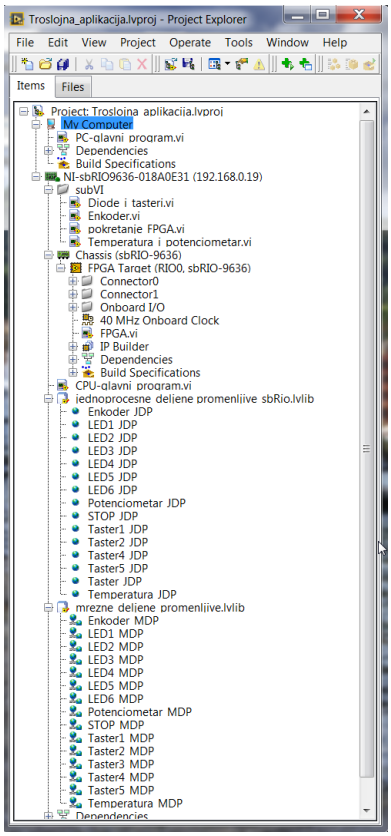
Slika 146: Prikaz implementacije funkcionalnosti koja pokreće *FPGA* deo aplikacije.

Kreiranje reference se vrši uz pomoć funkcije *Open FPGA VI Reference*. Na ulaz *resource name* ove funkcije treba dovesti konstantu koja predstavlja *sbRio* na kome je potrebno pokrenuti *FPGA*. Prilikom dodavanja ove funckije potrebno je odabrati resurs na osnovu kojeg će se

pokrenuti *FPGA*.

Najjednostavniji i najbrži način je izbor *.lvbitx* fajla koji je već generisan, te je taj način i iskorišten. Takođe, moguće je da se odabere da se referenca dobije na osnovu *VI*-a koji opisuje *FPGA* deo aplikacije, kao i na osnovu *Build* specifikacije. Nakon otvaranja reference, dobra je praksa da se resetuje *FPGA* čip, da bi se vratio u inicijalno stanje. Najjednostavniji način je upotreba *Invoke Method*-a sa izborom operacije *Reset*. Posle reseta moguće je pokrenuti *FPGA* čip, što se opet radi upotrebom *Invoke Method*-a, samo sa izbrorom operacije *Run*. Kada se *FPGA* čip pokrene dobija se referenca koju je moguće iskorisiti za manipulisanje *FPGA* delom aplikacije i ona predstavlja izlaz funkcionalnosti koja je zadužena za pokretanje *FPGA* čipa, a koja je upravo opisana.

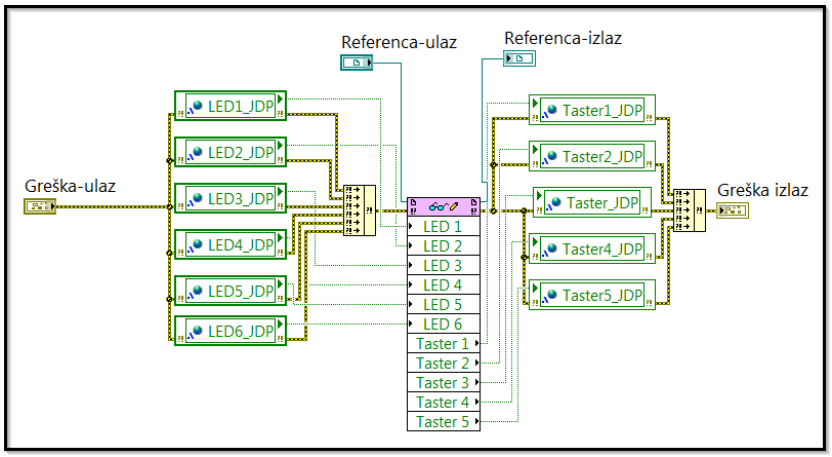
Kao što je već rečeno, potrebno je da ovaj sloj aplikacije manipuliše *FPGA* čipom, odnosno da čita podatke sa čipa ili da upisuje podatke na čip. *Real-Time* sloj aplikacije može se sastojati od više paralelnih petlji, kao i od više *VI*-eva koji se paralelno izvšavaju. Zbog toga je dobra praksa da se kreiraju deljene promenljive za podatke koji se razmenjuju između petlji. U ovom primeru neophodno je kreirati promenljive za LED diode, tastere, potenciometar, temperaturu i enkoder, stim da se pod promenljivom koja je zadužena za enkoder podrazumeva celobrojna promenljiva koja predstavlja poziciju enkodera, a ne sirove podatke koji dolaze sa *FPGA* čipa. Pored navedenih promenljivih, vrlo je bitno kreirati promenljivu koja je zadužena za propagaciju informacije o zaustavljanju programa. Ta promenljiva nazvana je *STOP_JDP*. Deljene promenljive koje pripadaju samo *Real-Time* delu aplikacije nalaze se u biblioteci *jednoprocesne_deljene_promenljive_sbRio*, a kompledan sadžaj ove biblioteke može se videti unutar stabla projekta koje je prikazano na Slici 147.



Slika 147: Prikaz stabla projekta nakon implementacije svih funkcionalnosti.

Nakon kreiranja svih deljenih promenljivih, može se krenuti u implementaciju *subVI*-eva koji će manipulisati sa *FPGA* delom aplikacije u smislu čitanja sa čipa i pisanja na *FPGA* čip. Radi bolje preglednosti, funkcionalnost čitanja i pisanja podeljena je u tri manje funkcionalnosti. Prva u nizu je funkcionalnost koja čita stanje tastera sa *FPGA* čipa, što naravno podrazumeva da će se dobiti informacije o stanju tastera na *Eval Kit*-u. Pored čitanja podataka o tasterima, ovaj *subVI* sadrži upisivanje željenog stanja LED dioda na *FPGA* čip, što naravno dovodi do prenosa tih stanja do samih dioda na *Eval Kit*-u. Implementacija navedene funkcionalnosti pikazana je na Slici 148. Ulazi u ovu funkcionalnost su referenca na *FPGA* i signal greške. Identični signali predstavljaju i izlaze, pošto se podaci razmenjuju preko deljenih promenljivih kao što je najavljeno ranije.

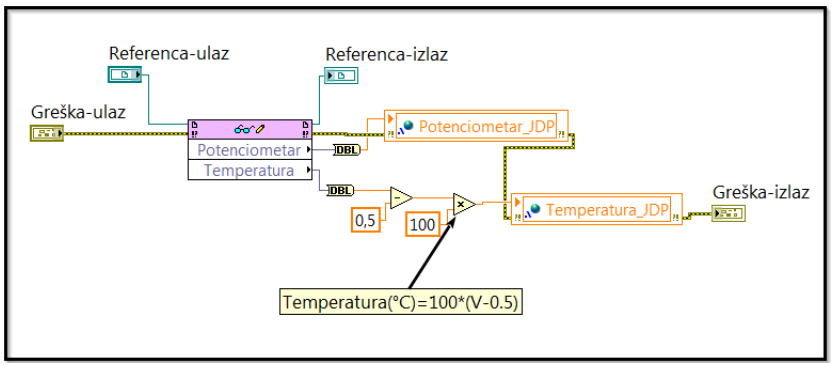
Implementacija je prilično jednostavna, čitanje i pisanje se obavlja upotrebom funckije *Read/Write Control Function* iz *FPGA Interface* pale-



Slika 148: Prikaz implementacije funkcionalnosti koja upisuje željeno stanje na LED diode i čita stanje tastera.

te funkcija. Potrebno je na ulaz *FPGA VI Reference In* dovesti referencu na *FPGA* funkcionalnost, koja je zapravo ulaz u kompletan *subVI*. Nakon toga pogrebno je dodati deljene promenljive koje predstavljaju diode i tastere. Na kraju potrebno je povezati deljene promenljive sa odgovarajućim elementima na *FPGA* interfejsu.

Naredni *subVI* koji prikuplja informacije sa *FPGA* čipa je *Temperatura_i_potenciometar*. I ovaj *subVI* ima identične ulaze i izlaze kao i prethodno opisani *subVI Diode_i_tasteri*, a njegova implementacija prikazana je na Slici 149.

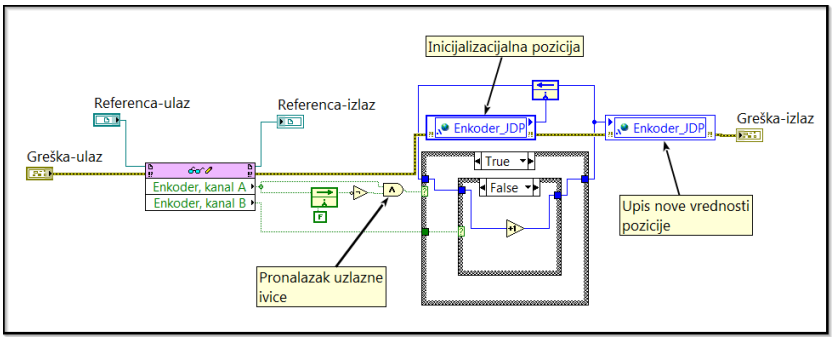


Slika 149: Prikaz implementacije funkcionalnosti koja čita vrednosti sa potenciometra i senzora temperature.

Kao što se vidi na slici, princip je identičan kao i kod prethodno opisanog *subVI*-a, stim da je dobro izvršiti konverziju iz vrednosti sa nepoomičnim zarezom (*FPGA* podržava samo format podataka sa nepomičnim zarezom) u vrednost sa pomičnim zarezom, odnosno u *double* vrednost. Pored toga, ovde je pogodno konvertovati naponski nivo koji se dobija sa senzora temperature u temperaturu u stepenima Celzijusa. Prilikom opisa *Eval Kit*-a rečeno je se temperatura u stepenima Celzijusa dobija kao $100(V - 0.5)$, gde V predstavlja naponski nivo koji

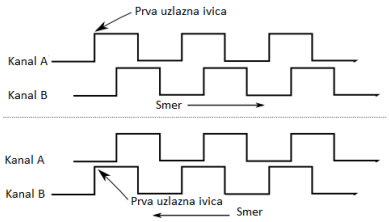
senzor daje.

Poslednja od pomoćnih funkcionalnosti je obrađivanje informacija koje se dobijaju sa enkodera. Impementacija navedene funkcionalnosti prikazana je na Slici 150. Ulazi i izlazi ove funkcionalnosti su identični kao i kod prethodno opisanog *subVI*-a. Takođe, na identičan način se čitaju informacije sa *FPGA* čipa, međutim u ovom slučaju se neće prosleđivati sirove informacije nego će se izvršiti brojanje inkremenata na samom enkoderu. Pre nego što se objasni implementacija, potrebno se osvrnuti na princip rada enkodera, odnosno potrebno je prodiskutovati šta znače kanali A i B koji se čitaju sa *FPGA* čipa.



Slika 150: Prikaz implementacije funkcionalnosti određuje poziciju enkodera.

Prilikom okretanja enkodera generiše se signal koji sadrži četvrtke. Prolazak kroz svaki mali inkrement enkodera generiše jednu četvrtku. Jasno je da je pronalaskom uzlaznih ivica u signalu i njihovim brojanjem moguće odrediti poziciju enkodera. Međutim, problem se javlja ukoliko se enkoder može okretati u suprotnom smeru. Tada je nemoguće tačno odrediti poziciju, pošto okretanje na bilo koju stranu generiše identične četvrtke. Zbog toga se uvodi još jedan kanal, koji je pomeren u odnosu na prvobitni kanal.



Slika 151: Slikovit prikaz signala koji se generišu na kanalima A i B prilikom okretanja enkodera.

Na Slici 151 je prikazan primer generisanja signala na enkoderu. Upotrebom oba kanala moguće je jednoznačno odrediti smer kretanja. Naime, ukoliko se prvo pojavi uzlazna ivica na kanalu A, a potom na kanalu B, enkoder se kreće u smeru koji možemo proglasiti pozitivnim smerom. Ako se ipak prvo pojavi uzlazna ivica na kanalu B, to znači da se enkoder kreće u smeru koji smo proglasili negativnim. Jasno je da se ugao enkodera može odrediti ukoliko znamo koliko inkrementa poseduje enkoder u jednom krugu.

Najjednostavniji način za implementaciju prethodno opisanog načina funkcionisanja enkodera je da se pronalaze uzlazne ivice na jednom od kanala, recimo A, a da se pritom gleda kakvo je stanje kanala B u tom trenutku. Ukoliko se pojavi uzlazna ivica na kanalu A, a na kanalu B je nizak naponski nivo znači da se enkoder kreće u smeru koji smo proglasili za pozitivan smer kretanja pošto se na kanalu B još uvek

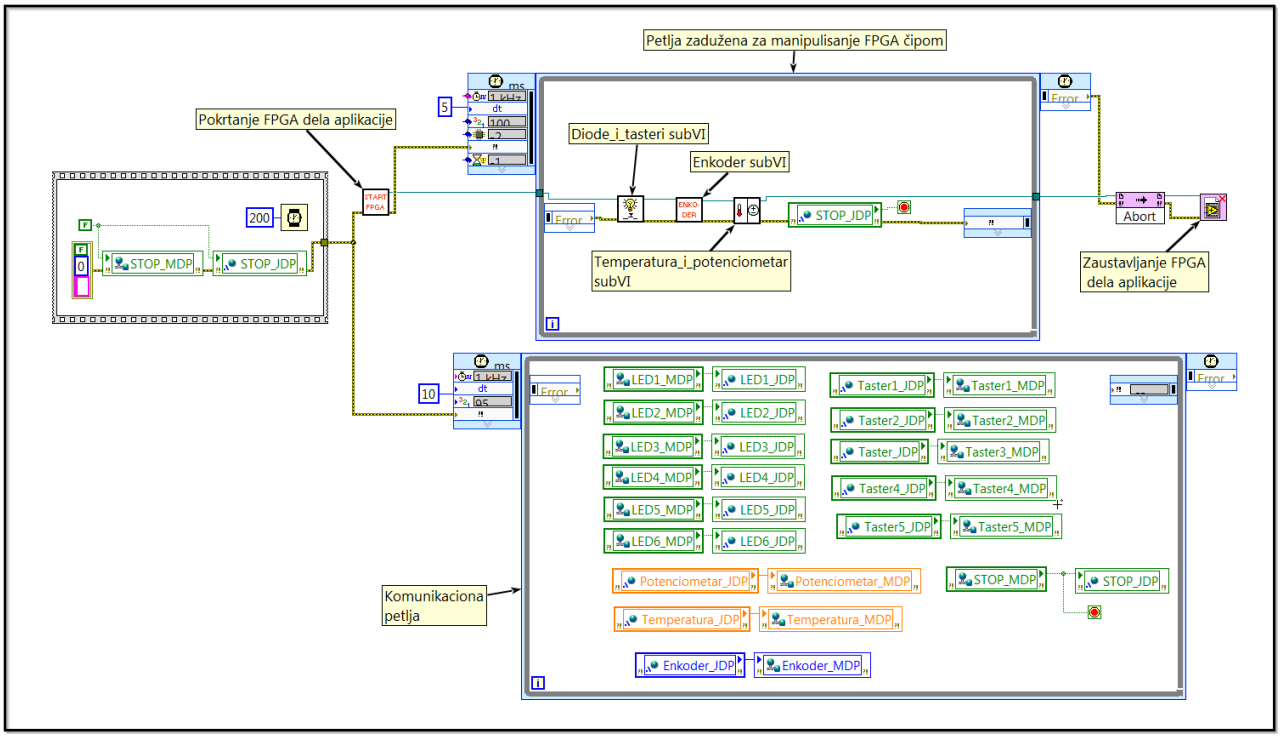
nije pojavila uzlazna ivica. Međutim, ukoliko se u trenutku pojavljivanja uzlazne ivice na kanalu A zatekne visok naponski nivo na kanalu B, to znači da se već desila uzlazna ivica na kanalu B, što znači da se enkoder kreće u negativnom smeru. Dakle, pri kretnji enkodera u pozitivnom smeru inkrementira se trenutna pozicija, dok pri kretanju u negativnom smeru pozicija se dekrementira. Na ovaj način pozicija enkodera je jednoznačno određena u svakom trenutku.

Implementacija opisanog algoritma nije preterano komplikovana, prvo se odredi uzlazna ivica na kanalu A, što znači da prethodno stanje treba da bude *false*, a trenutno *true*. Prethodno stanje se može dobiti upotrebom *Feedback Node*-a. Ukoliko se desi uzlazna ivica ulazi se u *True* slučaj spoljašnje *Case* strukture. Tada je samo potrebno proveriti na koju stranu se okrenuo enkoder, što znači samo proveriti stanje na kanalu B. Ukoliko je na B kanalu *False* konstanta enkoder se kreće u negativnom smeru, pa je potrebno dekrementirati poziciju, dok se u suprotnom slučaju kreće u pozitivnom smeru pa je potrebno inkrementirati poziciju. Na Slici 150 prikazan je samo slučaj kada je potrebno inkrementirati, pošto je suprotan slučaj sličan, samo se pozicija ne povećava za jedan inkrement, nego smanjuje. *False* slučaj spoljašnje *Case* strukture predstavlja situaciju kada se nije desila uzlazna ivica, i u tom slučaju potrebno je samo proslediti staru vrednost pozicije, bez izmena, odnosno samo povezati terminale, što nema potrebe prikazivati na posebnoj slici. Vrednost pozicije se prenosi iz iteracije u iteraciju upotrebom *FeedBack Node*-a, a nova vrednost pozicije se upisuje u deljenu promenljivu *Enkoder_JDP*. Takođe, iz ove iste promenljive se inicijalizuje *Feedback Node* da bi se pri novom pokretanju aplikacije nastavilo od prethodno ostvarene pozicije. Takođe, moguće je postaviti vrednost početne pozicije na bilo koju vrednost ako se pre početka izvršavanja ove funkcionalnosti upiše neka vrednost u deljenu promenljivu *Enkoder_JDP*.

Sve prethodno opisane funkcionalnosti su smeštene u folder *sub-VI* radi bolje organizacije. Kada su one uspešno implementirane, potrebno ih je iskombinovati u glavnoj funkcionalnosti koja se izvršava na *Real-Time* procesoru. Pre prikaza implementacije, potrebno je kreirati mrežne deljene promenljive da bi se mogle razmenjivati informacije između *PC* i *Real-Time* delova aplikacije. Potrebno je kreirati identične promenljive kao u biblioteci deljenih promenljivih koje pripadaju *sbRio* kontroleru. Naravno, za svaku promenljivu je potrebno definisati da je mrežna. Sve promenljive se nalaze u bibilioteci *mrezne_deljene_promenljive*.

Nakon kreiranja mrežnih mrežnih deljenih promenljivih, moguće je implemenirati glavni program *Real-Time* dela aplikacije. Implementacija ove funkcionalnosti prikazana je na Slici 152.

Na početku izvšavanja potrebno je postaviti *false* vrednosti u pro-



Slika 152: Prikaz implementacije glavnog programa *Real-Time* dela aplikacije.

menljive vezane na zaustavljanje programa, da bi se sprečilo eventualno neželjeno zaustavljanje nakon izvšenja prve iteracije petlje. Ovaj deo koda odvojen je u poseban frejm da bi se obezbedila mogućnost da se sačeka određeno vreme, da bi se vrednosti sigurno upisale pre nego što petlje krenu sa čitanjem. Ostatak koda moguće je staviti u drugi frejm, međutim estetski je lepše da se uz pomoć signala greške uspostavi redosled izvršavanja u kodu.

Po završetku inicijalizacije promenljivih, potrebno je pokrenuti *FPGA* čip, odnosno *FPGA* deo aplikacije. Već je opisan *subVI* koji ispunjava datu funkcionalnost i nazvan je *pokretanje_FPGA*. Ulaz u ovaj *subVI* je samo signal greške, dok se na izlazu pored signala greške dobija i referenca na pokrenuti *FPGA* deo aplikacije. Na Slici 152 se vidi da postoje dve petlje koje se paralelno izvršavaju. Gornja petlja zadužena je za razmenu podataka sa *FPGA*, odnosno zadatak joj da prikupi informacije o stanju tastera, upiše vrednosti za željeno stanje LED dioda, pročita vrednosti sa potenciometra i senzora temperature, kao i da odredi poziciju enkodera. Pošto govorimo o *Real-Time* delu aplikacije, logično je da se koristi *Timed Loop* struktura. U ovom slučaju izabran je period pozivanja od 5 milisekundi, sa prioritetom petlje 100. Unutar petlje potrebno je samo pozvati funkcionalnosti koje su ranije implementirane, a ispunjavaju sve zahteve koje ova petlja treba

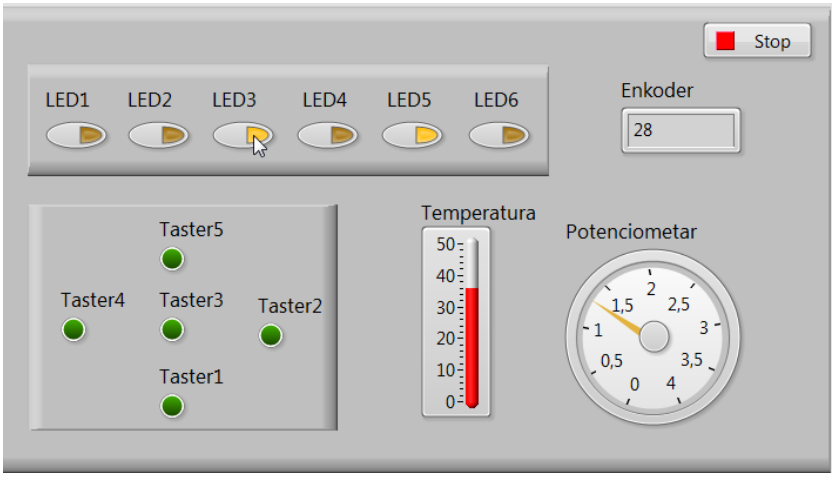
da ispuni. Dakle, moguće je bilo kojim redosledom dodati *subVI*-eve *Temperatura_i_potenciometar*, *Diode_i_tasteri* i *Enkoder*. Kada se ove funkcionalnosti izvrše i ažuriraju se podaci, potrebno je samo proveriti da li je potrebno zaustaviti petlju, što se jednostavno čita iz promenljive *STOP_JDP*. Petlja će se takođe zaustaviti ukoliko se pojavi greška bilo gde tokom izvršavanja, međutim to je u nadležnosti same *Timed Loop* strukture, te nema potrebe to eksplicitno implementirati. Naravno, kada se petlja završi, potrebno je zaustaviti *FPGA* i osloboditi referencu, što se radi upotrebom *Invoke Method*-a sa parametrom *Abort* i funkcije *Close FPGA VI Reference*.

Zadatak druge petlje je komunikacija sa *PC* delom aplikacije, što podrazumeva upisivanje odgovarajućih vrednosti u mrežne deljene promenljive, kao i čitanje vrednosti iz mrežnih deljenih promenljivih. Jasno je da računar ne može da vidi deljene promenljive koje su vezane za *sbRio*, te je stoga potrebno vršiti razmenu informacija uz pomoć mrežnih deljenih promenljivih. Kao što se vidi na Slici 152 stanje LED dioda se čita iz mrežnih deljenih promenljivih u deljene promenljive koje su vezane za *sbRio* procese. Time se obezbeđuje da se sa korisničnog interfejsa, koji se izvršava na računaru, može manipulisati stanjem LED dioda. Takođe, čita se vrednost iz mrežne deljene promenljive *STOP_MDP* i upisuje u deljenu promenljivu *STOP_JDP* koja je vezana za *sbRio* procese, čime se omogućuje zaustavljanje kompletne aplikacije sa korisničkog interfejsa. Vrednost svih ostalih promenljivih potrebno je prikazivati na korisničkom interfejsu ili izvršavati određene funkcionalnosti u zavisnosti od stanja ovih promenljivih. Zbog toga se vrši pisanje u mrežne deljene promenljive iz deljenih promenljivih *sbRio* procesa. Pošto komunikacija sa interfejsom treba da ima manji prioritet od same funkcionalnosti *Real-Time* aplikacije, prioritet ove petlje je postavljen na 95, dok je interval pozivanja postavljen na 10 milisekundi.

Očigledno je da u ovom primeru *Real-Time* deo aplikacije nema kompleksne funkcionalnosti, već samo prikuplja i razmenjuje informacije između ostalih komponenti. Razlog za ovako jednostavan *Real-Time* deo aplikacije je to što je ideja ovog primera da se pokaže način implementacije troslojne aplikacije, sa što manje komplikovanja da bi se jasno uvidele osnovne komponente svake troslojne aplikacije. Čitaocu se ostavlja da implementira značajno kompleknije zadatke na *Real-Time* delu aplikacije i iskoristi puni potencijal *sbRio* kontrolera.

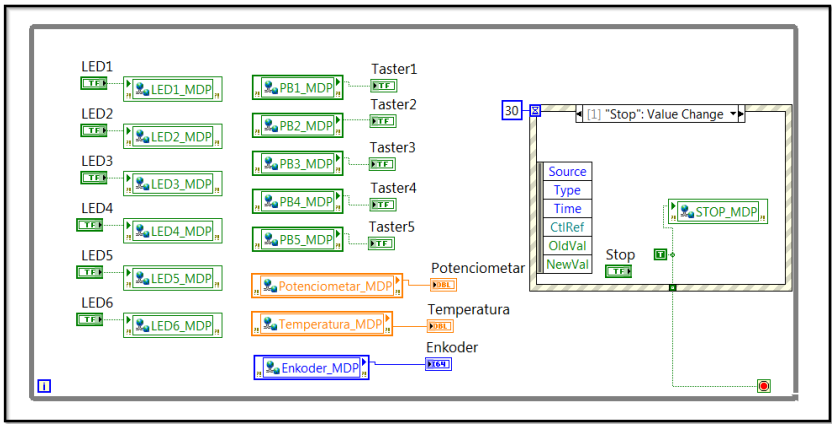
PC deo aplikacije- korisnički interfejs

PC deo aplikacije treba da predstvlja upotrebljiv korisnički interfejs *Front Panel* treba na što jednostavniji način prikaže sve potrebne zadatke. Odnosno, potrebno je dodati kontrole za LED diode. Prome-



Slika 153: Prikaz izgleda *Front Panel*-a *PC* dela aplikacije.

nom stanja ovih kontrola, menja se stanje LED dioda na *Eval Kit*-u koji je povezan na *sbRio* kontroler. Takođe, potrebno je dodati indikatore koji će prikazivati stanje tastera na *Eval Kit*-u. Pored toga, potrebno je dodati indikatore za prikaz temperature, vrednosti naponskog nivoa na potenciometru i trenutne pozicije enkodera. Dobra praksa je da se izaberu indikatori na osnovu čijeg izgleda se odmah može naslutiti o kojoj promenljivoj je reč. Recimo za temperaturu je dobro izabrati *Vertical Fill Bar* koji svojim izgledom podseća na termometar. Naravno, potrebno je dodati Stop taster pomoću koga je moguće zaustaviti kompletnu aplikaciju. Na Slici 153 prikazan je izgled *Front Panel*-a *PC* dela aplikacije.



Slika 154: Prikaz implementacije *PC* dela aplikacije.

Implementacija funkcionalnosti *PC* dela aplikacije prikazana je na Slici 154. Potrebno je upisati stanje kontrola koje reprezentuju LED diode u odgovaranuće mrežne deljene promenljive. Time će se željeno

stanje za LED diode proslediti na *sbRio* i dalje na kontroler. Takođe potrebno je pročitati stanje tastera iz mrežnih deljenih promenljivih i prikazati ga na odgovarajuće indikatore na *Front Panel*-u. Takođe, potrebno je slično uraditi i sa temperaturom, potenciometrom i pozicijom enkodera. Još je samo potrebno detektovati aktivaciju tastera *Stop*. Najjefikasniji način je upotrebom *Event* strukture koja će hvatati događaj promene vrednosti na tasteru *Stop*. Kada se desi aktivacija, šalje se *true* konstanta na terminal za zaustavljanje. Takođe, *true* konstanta se upisuje i u mrežnu deljenu promenljivu *STOP_MDP* čime se omogućuje zaustavljanje *sbRio* dela aplikacije. U suprotnom slučaju, kada taster *Stop* nije aktiviran, *false* konstanta se šalje na terminal za zaustavljanje petlje, ali nema porebe za upisom u *STOP_MDP* jer se nije desila promena u odnosu na prethodno stanje.

Da bi se aplikacija testirala, potrebno je pokrenuti sva tri dela aplikacije, mada se dva dela nalaze na *sbRio* kontroleru. Dakle, prvo je potrebno pokrenuti aplikaciju na *sbRio* kontroleru, a zatim i deo aplikacije koji se izvršava na računaru. Naravno, ukoliko je potrebno, moguće je trajno smestiti *sbRio* delove aplikacije na sam kontroler, kao i podesiti da se pri uključenju kontrolera pokreće željena aplikacija.

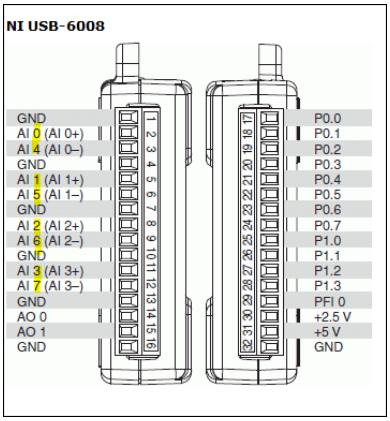
Aplikacija za simulaciju rada tastature sa NI DAQ 6008

U ovom poglavlju prikazano je povezivanje matrične tastature i pisanje softvera za prikupljanje informacija o stanju tastera na matričnoj tastaturi. Ukratko su opisane osobine multifunkcijskog uređaja *NI DAQ 6008*. Takođe, prikazan je način njegovog povezivanja i provere funkcionalnosti ovog uređaja. Nakon toga, opisana je matrična tastatura koja je korištena u ovom primeru. Prikazan je način povezivanja tastature, kao i potencijalni problemi koji postoje prilikom rukovanja ovakvim tipom tastature. Na kraju, nakon hardverkog dela posla, prikazan je razvoj softvera za ciklično prikupljanje stanja tastera na matričnoj tastaturi.

NI DAQ 6008

NI DAQ 6008 predstavlja jedan od mnogobrojnih uređaja kompanije *National Instruments* koji se koristi u jednostavnije svrhe kao što su jednostavno logovanje podataka, merenja u sistemima koji nisu sigurnosno kritični, kao i za laboratorijske ekspetimente. Ovaj uređaj predstavlja multifunkcijski ulazno/izlazni uređaj. Poseduje 8 analognih ulaza koji imaju 12-Bitne A/D konvertore i ima mogućnost prikupljanja 10kS/s. Takođe, tu su i dva analogna izlaza. Pored analognih ulaza i i izlaza, *DAQ 6008* poseduje i 12 digitalnih pinova koji mogu biti konfigurisani kao ulazi ili izlazi. Jedna od osnovnih prednosti *NI DAQ 6008* uređaja je jednostavno povezivanje sa računarom, preko USB-a. Takođe, *DAQ 6008* spada u red lakih uređaja, te je stoga prilično portabilan. Na Slici 155 prikazan je izgled *DAQ 6008* uređaja, dok je na Slici 156 prikazan raspored ulaza/izlaza ovog uređaja. Kompletnu dokumentaciju ovog, kao i svih ostalih *National Instruments* uređaja moguće je pronaći na oficijalnom sajtu ove kompanije. Određene specifičnosti koje su neophodne za realizaciju ovog projekta prodiskutovane su u sekciji koja se bavi razvojem softvera za prikupljanje informacija o stanju tastera na matričnoj tastaturi.

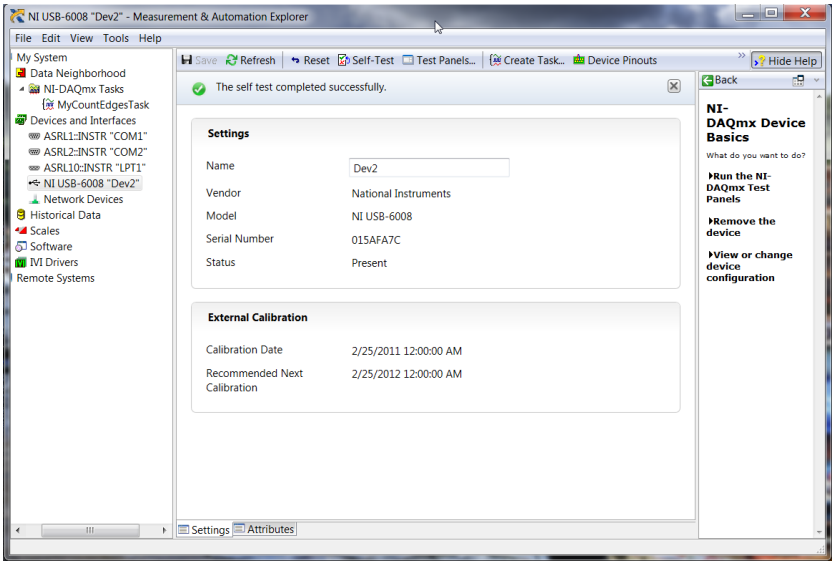
Slika 155: Izgled *NI DAQ 6008* uređaja.



Slika 156: Raspored pinova *NI DAQ 6008* uređaja.

Kao što je već rečeno, povezivanje *NI DAQ 6008* uređaja prilično je jednostavno, vrši se preko USB-a. Da bi se utvrdilo da sve funkcioniše, odnosno da je uređaj uspešno povezan na računar, neophodno je otvo-

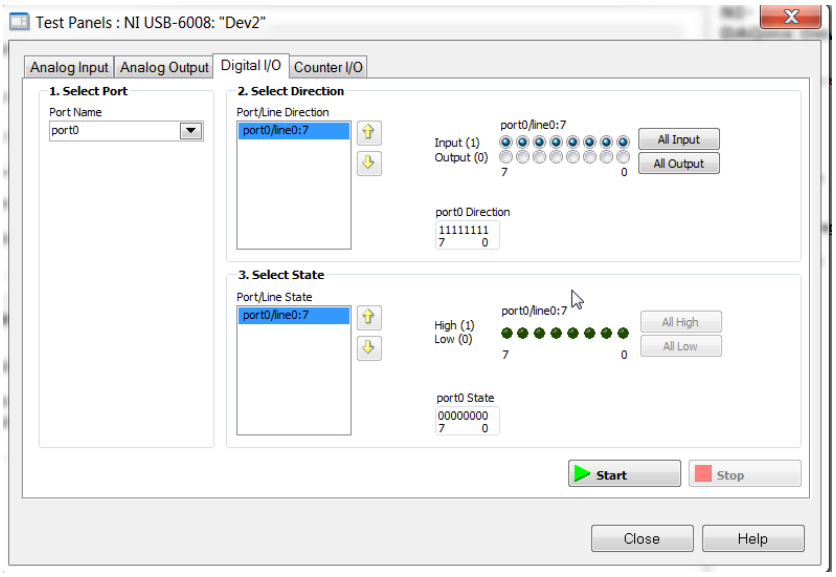
riti *NI MAX (National Instruments Measurement & Automation Explorer)*. Na Slici 157 je prikazan izgled *NI MAX* okruženja kada je povezan *NI DAQ 6008*. Ukoliko je uređaj uspešno povezan, trebalo bi da se pojavi u okviru sekcije *Devices and Interfaces*, pod imenom koje sadrži vrstu uređaja, kao i ime koje je moguće promeniti u okviru polja *Name*. Ukoliko nisu instalirani drajveri za *NI DAQ 6008*, jasno će biti naznačeno kada se selektuje uređaj. Pored toga, biće ponuđena opcija za automatsko instaliranje drajvera. Naravno, drajvere i uputstvo za instalaciju moguće je pronaći na sajtu *National Instruments* kompanije, te stoga ova knjiga ne sadrži uputstvo za to.



Slika 157: Prikaz izgleda *NI MAX* okruženja prilikom rukovanja *NI DAQ 6008* uređajem.

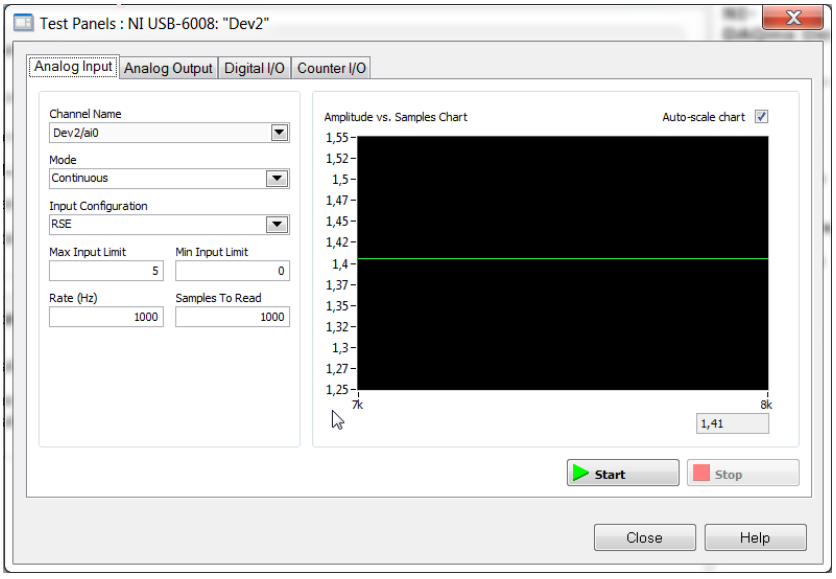
Najjednostavniji mogući test ovog uređaja vrši klikom na polje *Self-Test* i ukoliko je sve u redu trebalo bi da se dobije poruka kao na Slici 157. Specifični testovi mogu se vršiti klikom na *Test Panels*, gde se može vršiti testiranje svakog ulaza/izlaza posebno. Na Slici 158 prikazan je izgled panela za testiranje digitalnih ulaza/izlaza. Kao što se vidi na slici, potrebno je izabrati port (ovaj uređaj poseduje dva porta), nakon toga potrebno je odrediti da li je pin ulaz ili izlaz, i konačno moguće je setovati određene vrednosti na pinove ukoliko su definisani kao izlazi, odnosno čitati stanje na pinovima ukoliko su konfigurisani kao ulazi. Da bi se efikasno testirao rad pinova potrebno je povezati neke uređaje na same pinove, na primer diode za izlaze, tastere za ulaze. Druga varijanta bi bila da se uz pomoć uređaja za merenje napona proverava stanje pina.

Kao još jedan primer, navodimo princip testiranja analognog ulaza, što je prikazano na Slici 159. Prvo je potrebno izabrati koji analogni ulaz se testira, a nakon toga potrebno je odrediti parametre koji su



Slika 158: Prikaz izgleda *Test Panel*-a prilikom testiranja digitalnih ulaza/izlaza *NI DAQ 6008* uređaja.

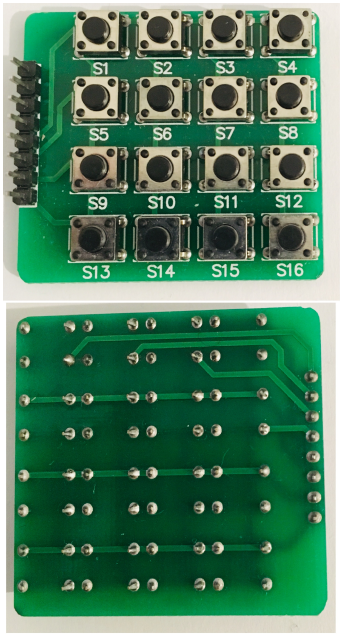
specifični za mereni signal, mod rada, konfiguracija ulaza, kao i ograničenja. Nije potrebno navoditi sve moguće opcije, jer izbor zavisi od uređaja koji je povezan na sam analogni izlaz, te se ostavlja čitaocu da sam prouči moguće opcije kada mu bude neophodno. Po pokretanju testa, očitavanja sa ulaza prikazuju se na grafiku. Nakon uspešno završenog testiranja, moguće je stupiti u razvoj softvera za *NI DAQ 6008*.



Slika 159: Prikaz izgleda *Test Panel*-a prilikom testiranja analognih ulaza *NI DAQ 6008* uređaja.

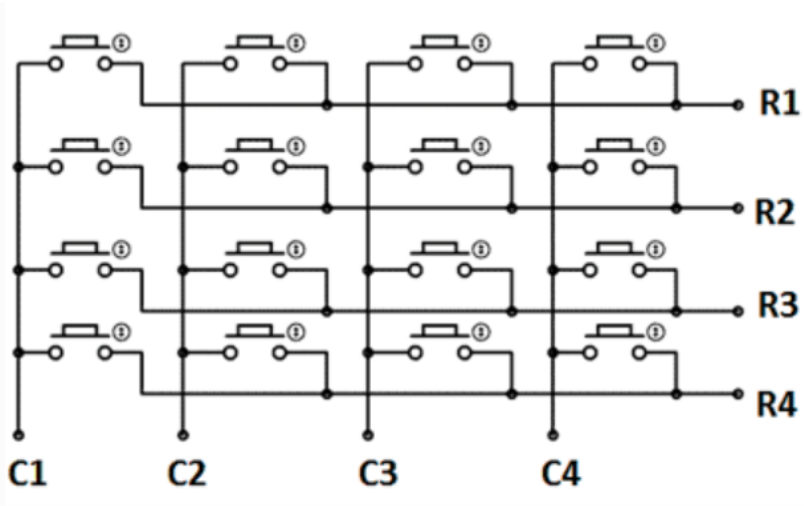
Matrična tastatura

Pre pisanja softvera za prikupljanje informacija o stanju tastera na matričnoj tastaturi neophodne je opisati princip rada same tastature. Prvo je potrebno odgovoriti na pitanje zbog čega se koristi matrična tastatura. Za primer možemo uzeti tastaturu koja je korištena u ovom projektu. Tastatura poseduje 16 tastera i kada bi željeli svaki pojedinačno povezati na neki drugi uređaj morali bi iskoristiti 16 pinova. Kod matrične tastature to nije slučaj, potrebno je svega 8 pinova. Naziv matrična tastatura potiče od toga što se tasteri rasporede i povežu u matričnu strukturu, pa se tasterima pristupa preko vrste i kolone. Zbog toga, za tastaturu od 16 tastera, potrebno je 4 pina za vrste i 4 pina za kolone. Na Slici 160 prikazan je izled konkretne matrične tastature koja je korištena u ovom projektu, dok je na Slici 161 prikazana šema matričnih tastatura ovog tipa. Kao što je već rečeno, osnovna prednost ovog vida tastatura je „jeftinije" povezivanje. Naravno, ovakav vid tastature ima i nedostatke o kojima će biti reči kasnije.



Slika 160: Izled matrične tastature.

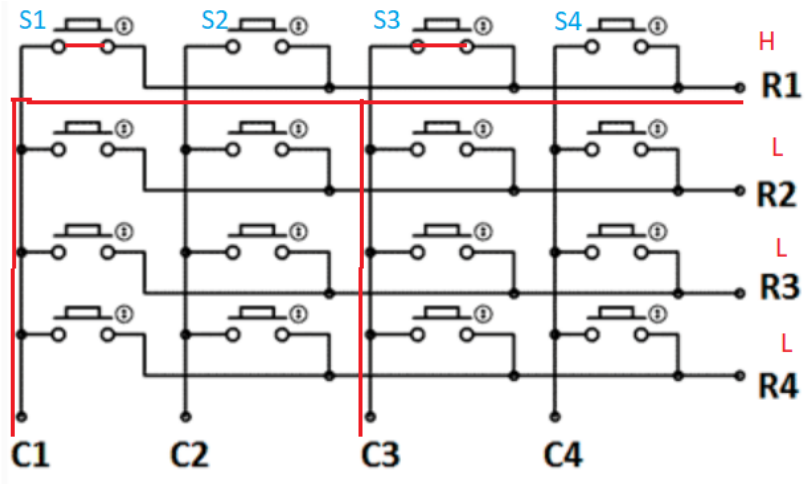
Analiziraćemo šemu tastature da bismo shvatili algoritam po kome treba skenirati stanje tastera na samoj tastaturi. Očigledno je da ne postoji mogućnost da se svi tasteri očitaju u isto vreme, te to predstavlja prvi nedostatak ove tastature, međutim to je jasno iz samog koncepta da se smanji broj potrebnih pinova za povezivanje. Ideja je da se skenira po vrstama ili kolonoma. Objasnićemo samo skeniranje po vrstama, pošto je princip za skeniranje po kolonama identičan.



Slika 161: Prikaz šeme matrične tastature. Sa R_i su označeni indeksi vrsta, dok su sa C_i označeni indeksi kolona.

Kada se radi o skeniranju po vrstama, tada se može postaviti visok naponski nivo na pin koji je povezan na vrstu koju je potrebno skenirati. Na sve ostale vrste se postavi niski naponski nivo. U tom slučaju, ukoliko je pritisnut neki taster u datoj vrsti, biće detektovan na odo-

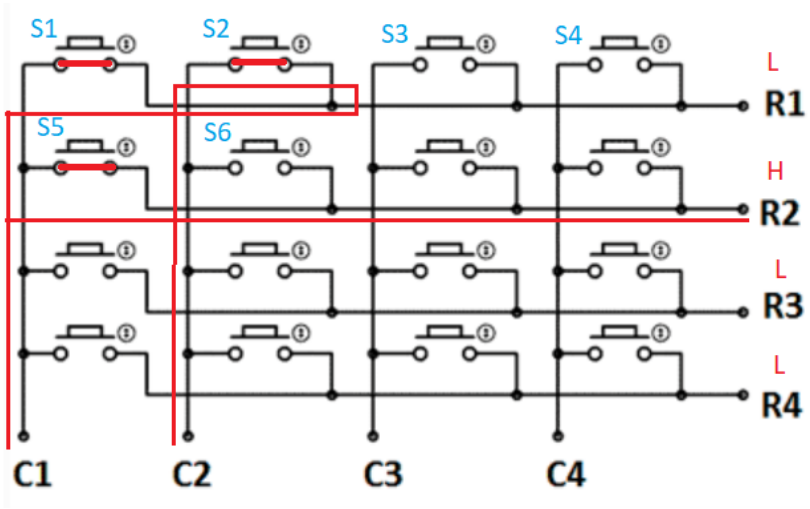
gvarajućem pinu koji je povezan na kolonu. Na ovaj način, moguće je odrediti stanje tastera u okviru indeksirane vrste (vrsta na koju smo postavili visok naponski nivo). Ilustracija opisanog algoritma prikazana je na Slici 162. Visok naponski nivo postavljen je na prvu vrstu, dok je na ostalim vrstama postavljen nizak naposki nivo (sa H označen je visok naponski nivo, dok je sa L označen nizak naponski nivo). Tasteri koji smo označili sa $S1$ i $S3$ su pritisnuti. Kao što vidimo na kolonama $C1$ i $C3$ detektujemo visok naponski nivo što je znak da su pritisnuti tasteri $S1$ i $S3$. Sa druge strane, tasteri $S2$ i $S4$ nisu pritisnuti, te na odgovarajućim kolonoma ne detektujemo visok naponski nivo. U ovom trenutku je potrebno naglasiti da je pretpostavljeno da postoje „pull down" otpornici na pinovima koji odgovaraju kolonama, što nije prikazano na slici radi jednostavnijeg prikaza. Nakon skeniranja jedne vrste, menja se indeksirana vrsta i ponavlja se posupak. Opisani algoritam predstavlja osnovu za pisanje softvera za skeniranje matrične tastature.



Slika 162: Ilustracija skeniranja prve vrste matrične tastature ukoliko su pritisnuti tasteri $S1$ i $S3$.

Još jedan nedostatak ove tastature je takozvana detekcija lažnog tastera. Da bismo pojasnili ovaj problem potrebno je posmatrati ilustraciju koja je data na Slici 163.

Kao što se vidi na slici, u toku je skeniranje druge vrste, pa je na taj pin postavljen visok naponski nivo, dok je na ostalim vrstama postavljen niski naponski nivo. U drugoj vrsti pritisnut je samo taster koji je označen sa $S5$. Njegovo stanje je uspešno detektovano, što se jasno vidi na slici. Međutim, tasteri $S1$ i $S2$ su takođe pritisnuti. Pošto je $S5$ pritisnut, visok naponski nivo pojaviti i na $S1$ koji je pritisnut pa će se visok naponski nivo pojaviti i na taksteru $S2$, a kako je i on pritisnut, visok naponski nivo će se pojaviti i na $C2$, a to znači da je pritisnut taster $S6$ (jer se skenira druga vrsta). Dakle, imamo lažnu detekciju stanja taste-



Slika 163: Ilustracija scenarija koji dovodi do detekcije lažnog tastera na matričnoj tastaturi.

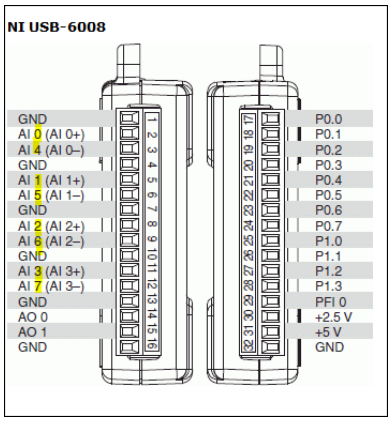
ra *S6*. Problem detekcije lažnog tastera može se rešiti tako što će se na svaki taster redno vezati dioda. To će obezbediti proticanje struje samo u jednom smeru, čime bi taster *S1* onemogućen da provede struju, pa bi detekcija lažnog tastera bila sprečena.

Implementacija softvera za skeniranje matrične tastature

Pre nego što krenemo sa opisom implementacije softvera u okviru *LabView* okruženja, opisaćemo princip povezivanja matrične tastature na *NI DAQ 6008*. Naravno, pre svega potrebno je proveriti da li je *NI DAQ 6008* uspešno povezan na računar, a taj postupak je detaljno opisan ranije, prilikom opisa samog uređaja. Na Slici 160 prikazan je izled matrične tastature koja je korištena u ovom projektu. Recimo da su pinovi numerisani od 1 do 8, počevši sa vrha. Ukoliko se posmatra slika koja prikazuje izgled tastature sa prednje strane, može se uočiti da je pin 5 povezan na prvi red, pin 6 na drugi red, pin 7 na treći red i konačno pin 8 na četvrti red.

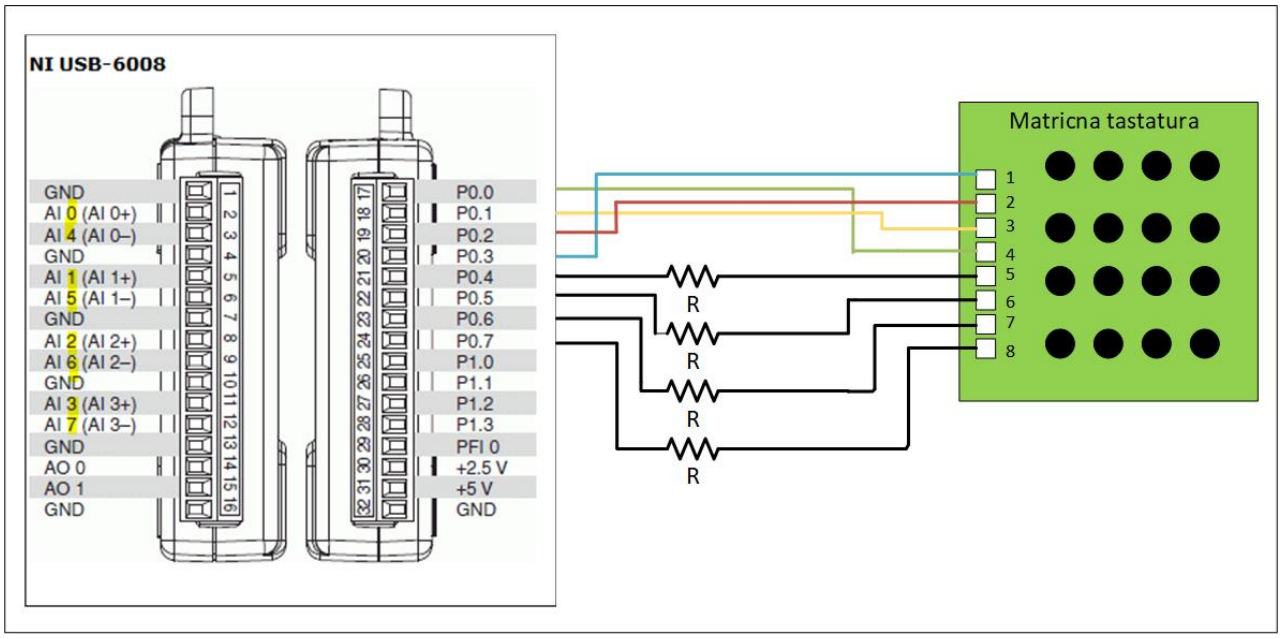
Da bismo videli kako su kolone povezane potrebno je obratiti pažnju na sliku koja prikazuje izgled tastature sa zadnje strane. Može se uočiti da je pin 4 povezan na prvu kolonu, pin 3 na drugu kolonu, pin 2 na treću kolonu i pin 1 na poslednju, četvrtu, kolonu.

Da bi povezali tastaturu na *NI DAQ 6008* potrebno je odrediti koji pinovi će se koristiti za indeksiranje vrsta, odnosno kolona. Na Slici 164 prikazan je raspored pinova uređaja. Pogodno je da se port 0 iskoristi za povezivanje. Dobar izbor bi bio da se pinovi od 4 do 7 iskoriste za indeksiranje vrsta, dok će se pinovi od 0 do 3 iskoristiti za indeksiranje kolona. Ovakav izvor je značajno bolji od nasumičnog rasporeda



Slika 164: Raspored pinova *NI DAQ 6008* uređaja.

zbog jednostavnije konfiguracije samih izlaza/ulaza unutar *LabView* okruženja. Bitno je naglasiti da *NI DAQ 6008*, kao i veliki broj mikrokontrolera, može da detektuje pojavljivanje niskog naponskog nivoa na pinovima koji se konfigurišu kao ulazi. To znači da će se koristiti obrnuta logika nego što je opisano u sekciji koja se bavi karakteristikama tastature. Razlika je samo u tome što će se vrste indeksirati niskim naponskim nivoom, a pritisnut taster će se detektovati pojavom niskog naponskog nivoa na pinu uređaja. Takođe, bitno karakteristika *NI DAQ 6008*-a je postojanje internih „pull up" otpornika na pinovima, što znači da nema potrebe dodavati dodatne otpornike prilikom povezivanja kolona, moguće ih je direktno povezati sa tastature na *NI DAQ 6008*. Sa druge strane, poželjno je dodati otpornike prilikom povezivanja pinova koji indeksiraju vrste radi sprečavanja eventualnog kratkog spoja. Konačna šema povezivanja prikazana je na Slici 165.

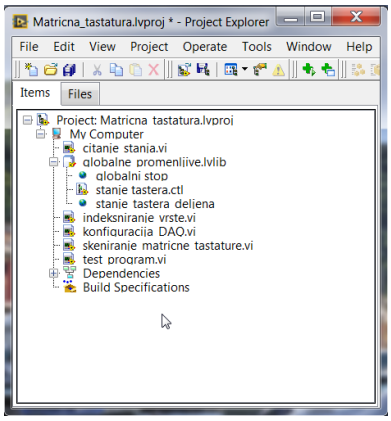


Slika 165: Prikaz povezivanja matrične tastature sa *NI DAQ 6008* uređajem.

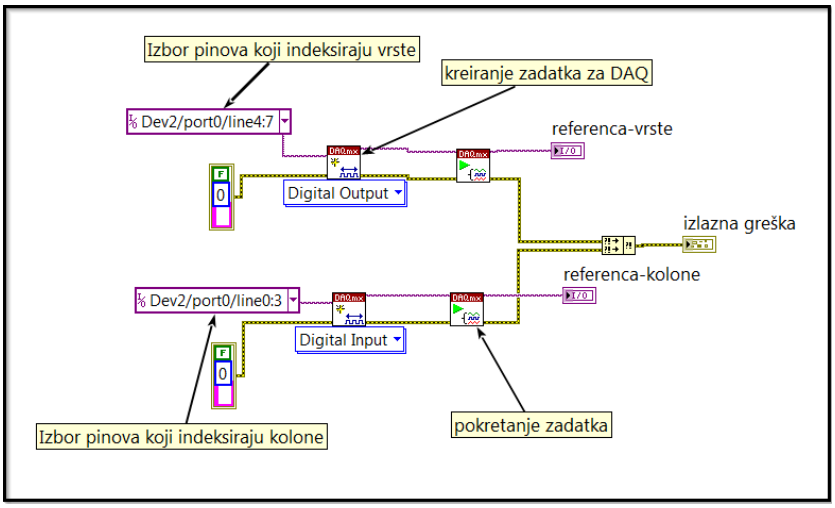
Nakon hardverskog povezivanja može se početi sa implementacijom sofrvera. Prvo je neophodno konfigurisati port uređaja *NI DAQ 6008*. Takođe, potrebno je kreirati i pokrenuti zadatke koje će izvršavati *DAQ*. Radi lepšeg izgleda, ova funkcionalnost odvojena je u poseban *subVI* koji je nazvan *konfiguracija_DAQ*. Ovaj *subVI* nema ulaze jer se u njemu podešava inicijana konfiguracija, dok su izlazi referenca-vrste (referenca na zadatak koji je zadužen za indeksiranje vrsta), referenca-kolone (referenca na zadatak koji je zadužen za čitanje stanja na pinovima koji predstavljaju kolone) i signal greške. Implementacija navedene funkcionalnosti prikazana je na Slici 167.

Prvo je potrebno odrediti konfiguraciju pinova i kreiranje zadataka za *DAQ*, što je moguće uraditi upotrebom funkcije *DAQmx Create Virtual Channel*.

Potrebno je odabrati tip ulaza ili izlaza za koji se treba kreirati zadatak. Sa stanovišta *DAQ*-a indeksiranje vrsta predstavlja digitalne izlaze, dok čitanje kolona predstavlja digitalne ulaze, pa je tako potrebno odabrati prilikom dodavanja funckije *DAQmx Create Virtual Channel*. Nakon definisanja tipa zadaka, potrebno je dodeliti pinove. To je moguće uraditi pomoću ulaza *lines* tako što se dovede konstanta kao što je prikazano na slici. Prvo dio putanje je ime uređaja koje je zadano u *NI MAX*-u, nakon toga ide izvor porta, pa izbor samih pinova u okviru porta. Prilikom povezivanja određeno je da pinovi od 4 do 7 porta o predstavljaju indeksiranje vrsta, pa se to navodi kod konfiguracije digitalnih izlaza. Takođe, pinovi od 0 do 3 porta 0 su povezani na kolone, pa njih navodimo kao pinove kod konfiguracije digitalnih ulaza. Kao izlaz dobija se referenca na kreirani zadatak sa zadatim pinovima. Nakon konfiguracije, potrebno je pokrenuti oba zadatka, što je moguće izvršiti upotrebnom funkcije *DAQmx Start Task*, čime se dobija referenca na pokrenut zadatak, te je moguće upravljati *DAQ* uređajem.



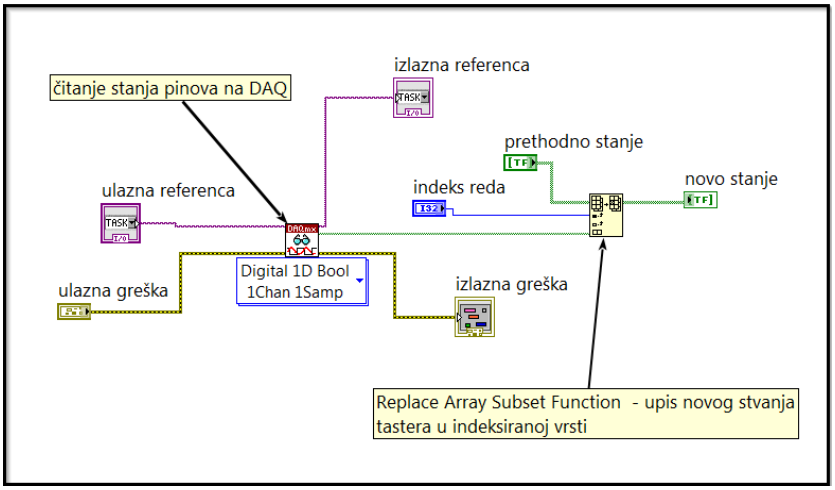
Slika 166: Prikaz stabla projekta. Sadrži sve pomoćne funkcionalnosti i glavni program.



Slika 167: Prikaz implementacije *subVI*-a koji vrši konfiguraciju *DAQ* uređaja.

Pored funkcionalnosti za konfiguraciju uređaja, dobra praksa je da se neprave pomoćni *subVI*-evi za očitavanje stanja na kolonama, kao i za indeksiranje vrste. Pošto je tastatura matrična, pogodno je i stanje svih tastera čuvati u takvom obliku.

Počinjemo od funkcionalnosti za skeniranje kolona. Ulazi u ovaj *subVI* treba da budu referenca na zadatak koji ima mogućnost čitanja pinova koji su povezani na kolone, prethodno stanje tastera, indeks vrste koja se trenutno skenira i naravno signal greške. Izlazi će biti referenca na zadatak koji opslužuje skeniranje kolona, novo stanje tastera i

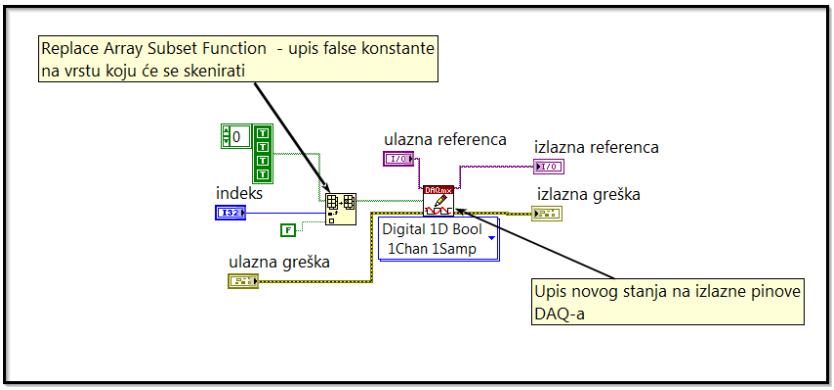


Slika 168: Prikaz implementacije *subVI*-a koji vrši čitanje stanja tastera u okviru indeksirane vrste.

signal greške. Implementacija ove funkcionalnosti prikazana je na Slici 168. Čitanje sa samog *DAQ* uređaja vrši se uz pomoć ugrađene *LabView* funkcije *DAQmx Read*. Naravno, potrebno je izabrati odgovarajuću konfiguraciju. U ovom slučaju, čitaju se digitalni ulazi, sa jednog kanala, jedan sempl, odnosno samo trenutno stanje pinova. Moguće je konfigurisati *DAQ* da prikuplja informacije sa određenom frekvencijom, pa da nakon definisanog vremena isporuči čitav set podataka, međutim sada to nije slučaj i potrebno je izabrati opcije koje su navedena ranije. Nakon izbora opcije, potrebno je samo povezati referencu na zadatak, koja je ulaz u ovaj *subVI*, na ulaz *task* funkcije *DAQmx Read*. Na izlazu *data* funkcije *DAQmx Read* pojaviće se niz od četiri elementa koji predstavljaju stanje četiri tastera u okviru indeksirane vrste. Taj niz je potrebno upisati u matricu stanja tastera. To se radi upotrebom ugrađene *LabView* funkcije *Replace Array Subset*, na čiji ulaz *n-dimension array* se dovodi niz koji je potrebno promeniti, što je u našem slučaju prethodno stanje tastera. Na ulaz *index (row)* potrebno je dovesti indeks reda koji treba promenti, što je indeks upravo skenirane vrste, i konačno na ulaz *new element (subarray)* se dovodi upravo pročitano stanje tastera u okviru indeksirane vrste. Na izlazu funkcije *Replace Array Subset* pojaviće se novo stanje tastera koje sadrži informacije o stanju tastera na indeksiranoj vrsti. Naravno, novo stanje tastera treba da bude izlaz iz samog *subVI*-a.

Naredna funkcionalnost je *indeksiranje_vrste*. Ulazi u ovu funkionalnost su referenca na zadatak koji je zadužen za indeksiranje vrsta, indeks vrste koju je potrebno indeksirati i signal greške. Izlazi su referenca na zadatak i signal greške. Implementacija navedenog *subVI*-a prikazana je na Slici 169.

Ranije je opisano da će se vrste indeksirati tako što će se na indeksi-

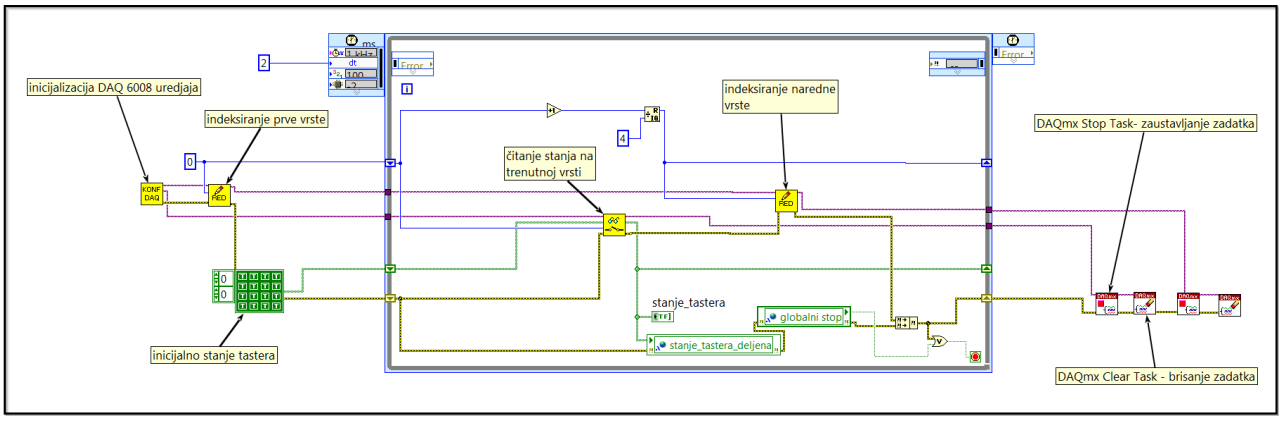


Slika 169: Prikaz implementacije *subVI*-a koji vrši indeksiranje vrste matrične tastature.

ranu vrstu upisivati niski naponski nivo, što u softveru predstavlja *false* konstanta. Upisivanje na same pinove *DAQ* uređaja vrši se uz pomoć ugrađene *LabView* funkcije *DAQmx Write*. Identično, kao i kod čitanja, potrebno je izabrati opciju *Digital-Single Channel-Single Sample-1D Boolean*. Izborom ove opcije dovoljno je da se na ulaz *task* funkcije *DAQmx Write* dovede referenca koja predstavlja zadatak povezan na odgovarajuće pinove, a na ulaz *data* dovede niz *bool* konstanti koji će se upisati na pinove. Taj niz se može kreitati upotrebom ugrađene *LabView* funckije *Replace Array Subset*. U ovom slučaju niz treba da sadrži sve *true* konstante, osim na mestu vrste koju je potrebno indeksirati. Dakle, na ulaz *n-dimension array* treba povezati niz koji ima 4 elementa (zbog četiri vrste), a pritom su svi elementi *true* konstante. Na ulaz *index (row)* potrebno je povezati indeks vrste koja se indeksira što je ulaz samog *subVI*-a. Konačno na ulaz *new element* treba povezati *false* konstantu. Po izvšetku *Replace Array Subset*, na izlazu će se pojaviti niz sa *false* konstantom na odgovarajućem mestu u nizu. Na kraju, dobijeni niz je potrebno upisati na pinove, tj. povezati na ulaz *data* funkcije *DAQmx Write*.

Kadu su definisane sve pomoćne funkcionalnosti, može se implementirati glavna funkcionalnost koja ima za zadatak da skenira stanje tastera na matričnoj tastaturi. Ideja je da se svakih 2*ms* menja indeksirana vrsta, a da se stanje tastera koje se zatekne u trenutno indeksiranoj vrsti upisuje u matricu koja čuva stanja svih tastera. Implementacija ovog *subVI*-a prikazana je na Slici 170.

Na samom startu ove funkcionalnosti potrebno je izvšiti konfiguraciju *DAQ* uređaja, što je jednostavno upotrebom *subVI*-a *konfiguracija_DAQ* koji je ranije opisan. Kao izlaz iz ovog *subVI*-a dobijaju se reference na zadatke koji upravljaju vrstama, odnosno kolonama. Nakon toga potrebno je indeksirati prvu vrstu, što se opet jednostavno radi upotrebom prethodno opisanog *subVI-a indeksiranje_vrste*. Naravno, potrebno je povezati referencu koja upravlja vrstama, kao i zadati



Slika 170: Prikaz implementacije funkcionalnosti koji vrši ciklično skeniranje matrične tastature.

indeks vrste, što je na početku 0 pošto indeksiramo prvu vrstu. Naravno, jasno je da indeksiranje u *LabView*-u kreće od nule. Predthodnim postupkom postavljen je niski naponski nivo na prvu vrstu i sada je neophodno implementirati petlju koja će ciklično skenirati stanje tastera i menjati indeksiranu vrstu. Dobra praksa je izabrati *Real Time* petlju koja će se precizno izvršavati svake 2*ms* sa velikom preciznošću. Unutar same petlje potrebno je prvo skenirati stanje u okviru trenutne vrste, što se može uraditi uz pomoć *subVI*-a *čitanje_stanja* koji je ranije opisan. Na ulaz ovog *subVI*-a dovodi se zadatak koji je zadužen za upravljanje kolonoma, indeks vrste koju trebamo skenirati i prethodno stanje tastera. Indeks trenutne vrste se pomoću šift registra prenosi kroz iteracije, a početna vrednost je postavljena na 0 pošto se kreće od prve vrste. Takođe stanje tastera na matričnoj tastaturi se prenosi kroz iteracije uz pomoć šift registara, a početno stanje je matrica 4x4 jer je to oblik tastature koji se koristi. Nakon pročitanog stanja u okviru indeksirane vrste, na izlazu *subVI*-a *čitanje_stanja* pojavljuje se novo stanje tastera koje se dalje prosleđuje na šift registar da bi se koristilo u narednoj iteraciji. Takođe, radi provere funkcionisanja softvera, dodaje se indikator *stanje tastera* da bi se mogle posmatrati promene kada se neki taster aktivira.

Nakon skeniranja stanja na trenutno indeksiranoj vrsti, potrebno je preći na narednu vrstu. Indeks trenutne vrste se inkrementuje i nakon toga odredi se ostatak pri celobrojnom deljenju sa 4 (pošto postoje 4 vrste). Na ovaj način se sprečava pojava indeksa koji je izvan granica. Taj novi indeks se prosleđuje na šift registar i biće iskorišten u sledećoj iteraciji. Pritom, pre kraja iteracije petlje, potrebno je indeksirati narednu vrstu, što se opet vrši upotrebom *subVI*-a *indeksiranje_vrste* na način koji je nešto ranije opisan.

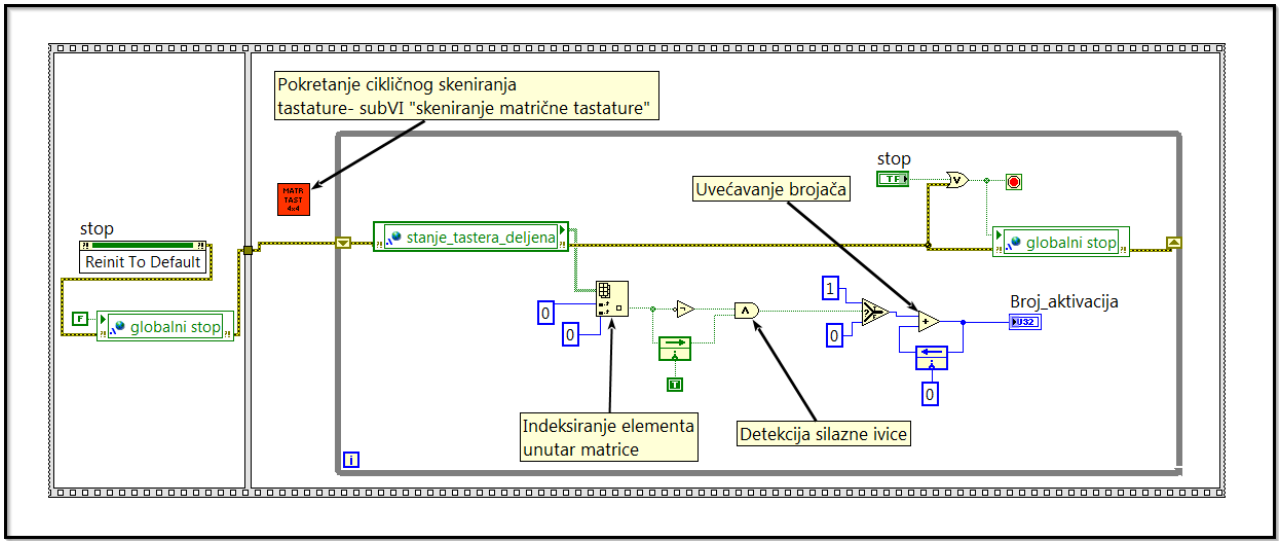
Pošto skeniranje tastature bez primene na nekom konkretnom zadatku nema upotrebnu vrednost, korisno bi bilo da se ova funckional-

nost može pokrenuti i zaustaviti iz neke druge funkcionalnosti. U tom slučaju, ciklično skeniranje tastature koje je opisano moglo bi se koristiti kao neka vrsta drajvera matrične tastature. Fokus ovog poglavlja je na opisu primene *NI DAQ 6008* uređaja kao primer načina rukovanja hardverom u okviru *LabView*-a, pa te je test program prilično jednostavan i biće opisan kasnije. Da bi se opisano skeniranje ponašalo kao drajver, potrebno je omogućiti da se stanje tastera prosledi u druge funkcionalnosti. Naravno, najjednostavniji način je da se napravi deljena promenljiva. Pošto se stanje tastera čuva kao matrica koja ima oblik kao fizička tastatura, potrebno je napraviti kontrolu koja opisuje navedeni tip promenljive, pošto u okviru standardnih kontrola ne postoji *2D boolean* tip. Dakle prvo je potrebno kreirati kontrolu, što je prilično jednostavno, samo se kreira matrica 4x4 sa *bool* kontolama. Taj tip podataka se sačuva kao kontrola (*.ctl* tip) u biblioteku. Nakon toga kreira se deljena promenljiva *stanje_tastera_deljena* u kojoj se kao tip podataka izabere prethodno kreirana kontrola. Na kraju, samo je potrebno upisati stanje tastera u ovu kontrolu, paralelno sa upisom u lokalni indikator *stanje_tastera* kao što je prikazano na Slici 170. Pored promenljive koja služi za razmenu informacija o stanju tastera, potrebno je obezbediti način za zaustavljanje skeniranja, tj. potrebno je kreirati deljenu promenljivu *globalni_stop* koja zaustavlja glavnu petlju. Pored zaustavljanja prirodnim putem, petlja se zaustavlja ukoliko se desi neka greška u toku izvršavanja programa. Kada se glavna petlja zaustavi, potrebno je zaustaviti i obrisati zadatke koji su zaduženi za upravljanje vrstama i kolonama. To se radi uz pomoć ugrađenih *LabView* funkcija *DAQmx Stop Task* i *DAQmx Clear Task*.

Test program

Kao što je ranije naglašeno, fokus ovog poglavlja je na prikazu rukovanja hardverom u okviru *LabView* okruženja. Zbog toga je testni program koji je opisan u ovoj sekciji prilično jednostovan. Ideja je samo da se pokaže da prethodno implementirana funkcionalnost za ciklično skeniranje može da se koristi kao drajver za matričnu tastaturu. Naime, ideja je da se implementira funkcionalnost koja će brojati koliko puta je pritisnut taster *S1* na matričnoj tastaturi (taster na preseku prve vrste i prve kolone). Implementacija ove funkcionalnosti prikazana je na Slici 171.

Na početku izvršavanja programa taster *stop* se vraća u inicijalno stanje da se program ne bi zaustavio u prvoj iteraciji petlje. Takođe, upisuje se *false* konstanta u deljenu promenljivu *globalni_stop* da bi se sprečilo eventualno zaustavljanje skeniranja odmah na početku. Da bi se obezbedilo da se resetovanje na početne vrednosti izvrši pre glavne petlje, iskorištena je *Flat sequence* struktura, i u prvi frejm je ubačeno re-



Slika 171: Prikaz implementacije test programa. Osnovni zadatak je brojanje aktivacija tastera na preseku prve vrste i prve kolone.

setovanje na početne vrednosti. Nakon resetovanja, pokreće se funkcionalnost koja vrši ciklično skeniranje matrične tastature. Ciklično skeniranje vrši *subVI skeniranje_matrične_tastature*, čija je implementacija opisana ranije. Očigledno je da će se skeniranje izvršavati paralelno sa glavnom petljom test programa. Unutar *while* petlje test programa prvo se čita stanje tastera na matričnoj tastaturi. Pošto je zadatak da se broji broj aktivacija tastera koji se nalazi na preseku prve vrste i prve kolone, iz matrice stanja se izvlači stanje tog tastera upotrebom ugrađene *LabView* funkcije *Index Array*. Kako je zadatak odrediti broj aktivacija, to znači da je potrebno odrediti broj silaznih ivica na ovom tasteru. To je moguće uraditi tako što se upoređuje trenutno stanje sa prethodnim stanjem tastera. Ukoliko je trenutno stanje *false*, a prethodno *true* znači da se desila silazna ivica. Prethodno stanje se može dobiti upotrebom šift registra ili *Feedback node*-a kao što je prikazano na slici. Kada se desi silazna ivica, potrebno je inkrementirati trenutni broj aktivacija koji se prenosi kroz iteracije upotrebom *Feedback node*-a. Takođe, trenutni broj aktivacija tastera *S1* prikazuje se na indikatoru *broj_aktivacija*. Zaustavljanje kompletne aplikacije vrši se klikom na taster *stop* ili pojavom greške u bilo kom delu programa.

U ovom poglavlju prikazano je praktično rukovanje hardverom u okviru *LabView*-a, opisan je *NI DAQ 6008* multifunkcijski uređaj, kao i matrična tastatura i način njenog funkcionisanja. Implementirana je funkcionalnost za ciklično skeniranje matrične tastature, i na kraju prikazan je jednostavan primer korištenja informacija koje se prikupljaju sa tastature. Čitaocu se ostavlja da smisli i pokuša da implementira praktične primere primene stanja tastera na matričnoj tastaturi. Kada

se implementira aplikacija na prethodno opisani način, može se desiti da aplikacija broji više aktivacija nego što je ispravno. Razlog za to je takozvani „bounce” koji postoji kod većine mehaničkih tastera. Čitaocu se ostavlja da reši navedeni problem, ukoliko se on pojavi.

Bibliografija

[1] Thomas Bress. *Effective LabView programming*. National Technology and Science, 2013.

[2] National Instruments. *CompactRIO Developers Guide*. National Instruments, May 2009.

[3] Jeffrey Travis; Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall, New Jersey, USA, third edition, 2006.

[4] Rick Bitter; Taqi Mohiuddin; Matt Nawrocki. *LabView Advanced Programming Techniques*. CRC Press, Taylor and Francis Group, second edition, 2007.

[5] Boris Jakovljević; Stefana Jocić; Tomislav Novak; Živko Kokolanski; Bodan Velkovski; Dariusz Tefelski; Angelika Tefelska; Milica Janković; Marko Barajaktarović; Kosta Jovanović; Nikola Knežević; Petar Atanasijević; Marija Novičić. *Control, virtual instrumentation and signal processing use cases practicum*. Fakultet tehničkih nauka, Novi Sad, Serbia, 2019.

[6] Karl J. Åström; Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. SA: The Instrumentation, Systems, and Automation Society, 2 sub edition edition, 1995.