

Poglavlje 11

Softverske mašine stanja

11.1 Uvod

Mašine sa konačnim brojem stanja (FSM, eng. *Finite State Machine*), mašine stanja, odnosno konačni automati, predstavljaju matematički model izračunljivosti. U osnovi se zasnivaju na apstraktnoj mašini koja se u proizvoljnom trenutku može naći tačno u jednom stanju (eng. *state*). Ova mašina može izvršiti prelaz (eng. *transition*) iz jednog u drugo stanje kao odziv na događaje na ulazima (eng. *inputs*). Oslanjajući se na ove principe, mašinama stanja se mogu implementirati sistemi vođeni događajima (eng. *event-driven system*, koji menjaju svoj odziv na pobudu u zavisnosti od unutrašnjeg stanja u kom se, u trenutku, pobude nalaze.

Mašine stanja predstavljaju osnovni koncept koji se upotrebljava prilikom dizajna kako digitalnog hardvera, tako i pojedinih tipova softvera. Svođenje algoritma na oblik mašine stanja može u mnogome pojednostaviti dizajn softvera za mikrokontrolere, čime će on biti pregledniji, razumljiviji i generalno jednostavniji za implementaciju.

Softverske mašine stanja se mogu definisati pomoću šest pojmove:

1. *Skup ulaza* – predstavlja skup događaja na koje mašina stanja treba da reaguje, tj. nakon čije pojave mašina treba da izvrši prelaz i da realizuje odgovarajući izlaz.
2. *Skup izlaza* – je skup postupaka koje je potrebno izvršiti kao odgovor na pojavu odgovarajućih ulaza.
3. *Skup stanja* – predstavlja skup pravila po kojima sistem postupa u situaciji kada se pojavi neki ulaz. Nakon pojave ulaza, sistem posmatra skup pravila za stanje u kom se, u tom trenutku, nalazi i odlučuje da li je potrebno da generiše prelaz i izlaz, ili nije potrebno ništa da preduzme. Svako stanje se može tumačiti kao jedinstven način ponašanja sistema, odnosno jedinstven način reagovanja na ulaze.

4. *Početno stanje* – je inicijalno stanje u kom se sistem nalazi nakon uključivanja ili resetovanja mikrokontrolera.
5. *Funkcija prelaza* – predstavlja funkciju koja na osnovu trenutnog stanja i aktivnog ulaza određuje naredno stanje u koje je neophodno preći. Ovim se određuje kada je potrebno da sistem promeni svoje ponašanje, što se postiže prelaskom u drugo stanje.
6. *Izlazna funkcija* – je funkcija koja na osnovu trenutnog stanja i/ili aktivnog ulaza određuje koji izlaz je potrebno realizovati.

Ukoliko izlazna funkcija zavisi od trenutnog stanja u kom se mašina nalazi i ulaza, definicija odgovara Milijevim (eng. *Mealy*) modelom, i može se modelovati kao Milijeva mašina. Međutim, ako izlazna funkcija zavisi samo od trenutnog stanja u kom se mašina nalazi, definicija prikazuje Murov (eng. *Moore*) model, na osnovu kog se konačni automat može modelovati kao Murova mašina. U okviru ove knjige, pod pojmom mašine stanja će se uvek podrazumevati Milijev model, osim ukoliko se eksplicitno ne naglasi suprotno.

11.2 Načini reprezentacije mašina stanja

Kako mašine stanja predstavljaju koncept koji se koristi u različitim oblastima, razvijen je širok spektar različitih načina reprezentacija. Svaki način ima svoje prednosti i mane, i shodno tome se može jednostavnije ili komplikovanije primeniti u konkretnoj oblasti. Prilikom razvoja softvera za mikrokontrolere, može se primeniti nekoliko načina, kao što su opisni, tabelarni i dijagrami stanja.

Opisni način reprezentacije mašina stanja podrazumeva da se automat opiše помоћу govornog jezika. Iako deluje da je ovaj način reprezentacije najjednostavniji, on ima značajne probleme, koji se ogledaju u dužini, nepotpunosti detalja, različitim tumačenjima istih iskaza od strane dva ili više programera itd. Iz navedenih razloga, ovaj način reprezentacije se obično koristi samo prilikom neformalnog opisa mašine, gde se najčešće govorи о мајини uopšteno, bez detalja.

Tabelarni prikaz mašina stanja se sastoji od tabele prelaza i tabele izlaza. Ove dve tabele reprezentuju funkciju prelaza i izlaznu funkciju, redom. Tabele po redovima razlikuju trenutna stanja u kom se nalazi mašina stanja, a po kolonama moguće ulaze. Svako polje u tabeli prelaza odgovara stanju u koje je potrebno preći u slučaju odgovarajuće kombinacije trenutnog stanja i ulaza. U tabeli izlaza, ova kombinacija određuje izlaz koji je potrebno realizovati. Tabelarni prikaz jednoznačno određuje ponašanje mašine stanja, što znači da se ovim načinom može u potpunosti i nedvosmisleno zadati automat. Međutim, usled porasta broja stanja i ulaza, tabela postaje značajno veća, čime se može izgubiti na preglednosti i razumljivosti generalne funkcionalnosti i ponašanja mašine. Primeri tabele prelaza i tabele izlaza su redom prikazani u tabelama 11.1 i 11.2.

Tabela 11.1: Tabela prelaza

Ulazi Stanja \ Stanja	Ulaz0	Ulaz1	Ulaz2
S0	S0	S1	S2
S1	S2	S0	S1
S2	S1	S2	S0

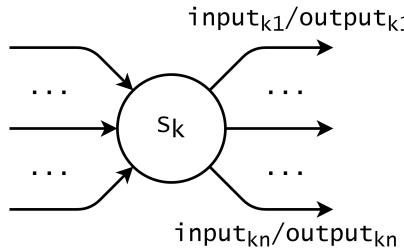
Tabela 11.2: Tabela izlaza

Ulazi Stanja \ Stanja	Ulaz0	Ulaz1	Ulaz2
S0	Izlaz0	Izlaz1	-
S1	-	-	Izlaz1
S2	Izlaz1	Izlaz2	Izlaz0

Dijagrami stanja predstavljaju grafički način reprezentacije mašina stanja. Klasičana forma dijagrama stanja namenjena za reprezentaciju mašina sa konačnim brojem stanja je direktni graf koji se sastoji od sledećih elemenata:

1. *Čvorovi* – predstavljaju konačan skup stanja uobičajno prikazanih u obliku kruga u čijem centru se nalazi ime stanja.
2. *Ulazni simboli* – su kolekcija imena ulaza, tj. imena događaja na koje mašina stanja reaguje.
3. *Izlazni simboli* – su kolekcija imena izlaza, tj. imena postupaka koje mašina stanja izvršava.
4. *Grane* – predstavljaju prelaze iz jednog u drugo stanje. Grana se uobičajno prikazuje u obliku strele koja je usmerena od tekućeg stanja, ka stanju u koje se prelazi. Uređeni par ulaza i izlaza se navodi uz samu granu u formatu "ulazX/izlazY". Ovaj par označava da se prelaz izvršava pod uslovom da se dogodio događaj *ulazX*, pri čemu se kao izlaz izvršava postupak *izlazY*. Ukoliko se ulaz izostavi, podrazumeva se da dolazi do bezuslovnog prelaza u naredno stanje, što podrazumeva da nije potrebno da se dogodi niti jedan događaj. U slučaju da se izostavi izlaz, podrazumeva se da se prilikom prelaza ne izvršava ni jedan postupak.
5. *Početno stanje* – je inicijalno stanje u kom se nalazi mašina stanja. Obično se označava strehom bez izvora koja je usmerena ka čvoru ovog stanja ili dodatnim koncentričnim krugom unutar čvora.

Primer standardnog čvora koji se upotrebljava u dijagramima stanja, sa uvirućim i izvirućim granama je prikazan na slici 11.1. Uvirući čvorovi predstavljaju prelaze u stanje koje odgovara čvoru, a izviruće grane prikazuju prelaze iz ovog stanja. Uz svaku izlaznu granu se nalazi uslov pod kojim se prelaz realizuje i izlaz koji se aktivira tom prilikom.



Slika 11.1: Primer stanja softverske mašine stanja

11.3 Implementacija mašina stanja

Implementacija mašina stanja u programskom jeziku C se najčešće realizuje upotrebom *switch-case* naredbe. Promenljiva stanja se u okviru ove naredbe upotrebljava u cilju odabira slučaja koji je potrebno izvršiti. Svaki slučaj predstavlja jedno stanje, a ono što je u njemu implementirano predstavlja funkciju prelaza i izlaznu funkciju, tj. pojedinačne grane koje izviru iz čvora ovog stanja. Grane se realizuju upotrebljavajući *if-else if* naredbe, gde se u slučaju više od dve grane dodatno upotrebljava *else if* naredba. Ulazi mašine stanja predstavljaju uslove *if* naredbi. Tela ovih naredbi čine prelazi, tj. promene vrednosti promenljive stanja i izlazi. Primer implementacije stanja sa slike 11.1 je dat u okviru listinga 11.1.

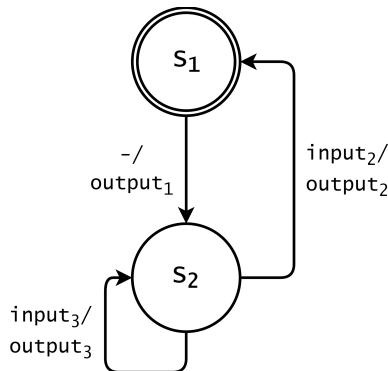
```

switch (state)
{
    ...
    case S_k:
        if (input_k_1)
        {
            state = S_1;
            output_k_1;
        }
        else if (input_k_2)
        {
            state = S_2;
            output_k_2;
        }
        ...
        else if (input_k_n)
        {
            state = S_n;
            output_k_n;
        }
        break;
    ...
}
  
```

Listing 11.1: Primeri pisanja stanja u C jeziku

Ulazi mašine stanja u opštem slučaju mogu biti različiti događaji. Kako se mašina stanja u ovom slučaju izvršava na mikrokontroleru, događaji mogu biti hardverskog i softverskog porekla. Događaji hardverskog porekla koji se mogu koristiti kao ulazi mašine stanja mogu biti pritisak tastera, upotreba matrične tastature, promena vrednosti prekidača, pokreti džojstika, događaji izazvani različitim prekidima, prihvat poruke putem serijskog porta itd. Softverski događaji obuhvataju provere istinitosti bulovih izraza, čiji operandi u opštem slučaju mogu biti proizvoljni. To mogu biti provere vrednosti promenljivih ili povratnih vrednosti funkcija itd. Česta pojava je da ulazi mašine stanja predstavljaju kombinacije više različitih događaja. U ovom slučaju, potrebno je voditi računa da mašina stanja bude deterministička, tj. da ne sme biti omogućen prelaz iz jednog stanja u dva različita pod istim uslovom.

Izlazi mašine stanja, takođe, mogu biti različiti postupci. U slučaju upotrebe u okviru softvera za mikrokontrolere, izlazi mogu biti hardverskog i softverskog tipa. Izlazi hardverskog tipa su, u najčešćem slučaju, pozivi funkcija koji upravljaju hardverskim komponentama koje su povezane sa mikrokontrolerom. Na primer, to mogu biti paljenje i gašenje dioda, podešavanje vrednosti sedmo-segmentnog displeja, prikaz na TFT displeju, slanje poruke putem serijskog porta itd. Izlazi softverskog tipa obuhvataju sve promene vrednosti promenljivih i pozive funkcija čiji su rezultati čisto softverskog tipa, tj. ne izlaze iz okvira mikrokontrolera. Kao i u slučaju ulaza, izlazi mogu biti složeni i predstavljati kombinaciju postupaka različitog tipa.



Slika 11.2: Primer mašine stanja

Na slici 11.2 je prikazana mašina stanja koju čine dva stanja S_1 i S_2 , što će, kao posledicu, imati postojanje dva slučaja u okviru *switch-case* naredbe, koja implementira automat. U slučaju stanja S_1 prelaz je bezuslovan, što podrazumeva direktni prelaz u stanje S_2 , prilikom čega je potrebno izvršiti *output1*. U slučaju

drugog stanja, neophodno je proveriti da li je *input₂* zadovoljen, ukoliko jeste, potrebno je izvršiti prelaz u *S₁* i *output₂*. U suprotnom, potrebno je proveriti da li je *input₃* zadovoljen. Ukoliko jeste, stanje se ne menja, ali je potrebno izvršiti *output₃*. Implementacija opisane mašine stanja je prikazana u ovikru listinga 11.2.

```
switch (state)
{
    case S_1:
        output_1;
        state = S_2;
    break;
    case S_2:
        if (input_2)
        {
            output_2;
            state = S_1;
        }
        else if (input_3)
        {
            output_3;
            state = S_2;
        }
    break;
}
```

Listing 11.2: Primer opisa mašine stanja

Prilikom definisanja promenljive koja će da reprezentuje stanje unutar mašine stanja, uobičajno se upotrebljavaju enumerisani tipovi. Enumerisani tipovi su celobrojni tipovi čije vrednosti mogu biti dodatno predstavljene pomoću konstantnih imena. Kreiranje enumerisanih tipova se vrši pomoću ključne reči enum na sledeći način.

```
enum enum_name {const0, const1, const2, ... , constN};
```

Primer definisanja enumerisanog tipa za promenljivu koja će predstavljati stanje iz listinga 11.2 je dat u nastavku.

```
enum state_t {S_1, S_2};
```

Nakon definicije tipa, potrebno je kreirati promenljivu ovog tipa, koju je moguće istovremeno inicijalizovati, što se obično radi dodelom vrednosti inicijalnog stanja. Primer ovoga je dat u nastavku.

```
enum state_t state = S_1;
```

Kao što je rečeno, vrednosti enumerisanih tipova se mogu predstavljati i koristiti pomoću imena konstanti. Međutim one su, u svojoj osnovi, bazirane na celobrojnim vrednostima, što podrazumeva da se svaka konstanta koduje odgovarajućom

celobrojnom vrednošću. Ukoliko se eksplisitno ne navedu, prva konstanta ima vrednost nula, dok se vrednost svake sledeće dobija uvećavanjem za jedan. U definiciji enumerisanog tipa `state_t` vrednost konstante `S_1` je 0, a `S_2` je 1. U specifičnim situacijama neophodno je tačno navesti vrednosti svake konstante enumerisanog tipa, što se može eksplisitno navesti prilikom definicije tipa na sledeći način.

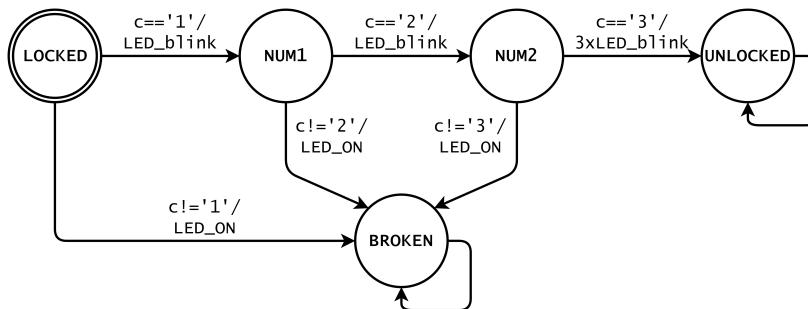
```
enum enum_name {const0 = value0,
                const1 = value1,
                ...,
                constN = valueN};
```

Takođe, vredi napomenuti da je moguće dodeliti alternativno ime za enumerisane tipove upotrebom ključne reči **typedef**. Ovo je korisno iz razloga što upotrebo alternativnih imena možemo značajno da povećamo preglednost i čitljivost koda, budući da dve reči možemo zameniti sa jednom. Prema tome, prethodni primer sa enumerisanim tipom `state_t` možemo imenovati na sledeći način:

```
typedef enum state_t {S_1, S_2} state_t;
```

Sada se definisanje promenljive ovog tipa može izvršiti na sledeći način:

```
state_t state = S_1;
```



Slika 11.3: Primer maštine stanja za otključavanje sefa

Na slici 11.3 je dat primer maštine stanja za otključavanje sefa, koja ima dodatno osiguranje u vidu kvara prilikom unosa neispravne šifre. Unošenjem ispravne šifre, koja je u ovom slučaju “123”, sef se otključava. Prilikom unosa svake cifre, vrši se provera njene validnosti. U slučaju ispravne cifre, prelazi se na proveru naredne uz vizuelnu signalizaciju u vidu treptanja diode. Ukoliko je cifra pogrešna, dioda će biti trajno uključena, što će signalizirati onesposobljenost brave. Postupak se ponavlja za preostale cifre, gde se nakon validnog unosa poslednje cifre izvršava signalizacija otključanosti sefa trostrukim treptajem diode.

U nastavku su data tri načina realizacije ove funkcionalnosti. Prvi je dat na listingu 11.3 uobičajenim načinom realizacije, dok je na listingu 11.4 prikazana

implementacija korišćenjem metodologije mašina stanja. Listing 11.5 prikazuje realizaciju upotreboru mašina stanja i funkcija koje odgovaraju ulazima i izlazima.

```
#include <stdint.h>
#include "uart.h"
#include "timer0.h"
#include "pin.h"

int main()
{
    pinInit(PORT_B, 5, OUTPUT);
    timer0InterruptInit();
    usartInit(115200);

    int8_t c;
    int8_t key[3] = "123";
    int16_t cnt = 0;

    pinSetValue(PORT_B, 5, LOW);

    while(1)
    {
        while(usartAvailable() == 0);
        c = usartGetChar();

        if (c == key[cnt])
        {
            pinSetValue(PORT_B, 5, HIGH);
            timer0DelayMs(1000);
            pinSetValue(PORT_B, 5, LOW);

            cnt++;

            if(cnt == 3)
            {
                for(int i = 0; i < 2; i++)
                {
                    timer0DelayMs(1000);
                    pinSetValue(PORT_B, 5, HIGH);
                    timer0DelayMs(1000);
                    pinSetValue(PORT_B, 5, LOW);
                }
                while(1);
            }
        }
        else
        {
            pinSetValue(PORT_B, 5, HIGH);
            while(1);
        }
    }
}
```

```
    }
    return 0;
}
```

Listing 11.3: "Klasičan" način implementacije

```
#include <stdint.h>
#include "uart.h"
#include "timer0.h"
#include "pin.h"

int main()
{
    pinInit(PORT_B, 5, OUTPUT);
    timer0InterruptInit();
    usartInit(115200);

    int8_t c;
    enum state_t {LOCKED, NUM1, NUM2, UNLOCKED, BROKEN};
    enum state_t state = LOCKED;

    pinSetValue(PORT_B, 5, LOW);

    while(1)
    {
        switch(state)
        {
            case LOCKED:
                while(usartAvailable() == 0);
                c = usartGetChar();
                if (c == '1')
                {
                    pinSetValue(PORT_B, 5, HIGH);
                    timer0DelayMs(1000);
                    pinSetValue(PORT_B, 5, LOW);
                    state = NUM1;
                }
                else
                {
                    pinSetValue(PORT_B, 5, HIGH);

                    state = BROKEN;
                }
                break;
            case NUM1:
                while(usartAvailable() == 0);
                c = usartGetChar();
                if (c == '2')
                {
                    pinSetValue(PORT_B, 5, HIGH);
                }
        }
    }
}
```

```

        timer0DelayMs(1000);
        pinSetValue(PORT_B, 5, LOW);
        state = NUM2;
    }
    else
    {
        pinSetValue(PORT_B, 5, HIGH);

        state = BROKEN;
    }
    break;
case NUM2:
    while(usartAvailable() == 0);
    c = usartGetChar();
    if (c == '3')
    {
        for(int i = 0; i < 3; i++)
        {
            pinSetValue(PORT_B, 5, HIGH);
            timer0DelayMs(1000);
            pinSetValue(PORT_B, 5, LOW);
            timer0DelayMs(1000);
        }

        state = UNLOCKED;
    }
    else
    {
        pinSetValue(PORT_B, 5, HIGH);

        state = BROKEN;
    }
    break;
case UNLOCKED:
    break;
case BROKEN:
    break;
}
}
return 0;
}

```

Listing 11.4: Implementacija upotreboom mašina stanja

```

#include <stdint.h>
#include "usart.h"
#include "timer0.h"
#include "pin.h"

int8_t checkDigit(int8_t c)

```

```
{  
    while(usartAvailable() == 0);  
  
    if(usartGetChar() == c)  
        return 1;  
    else  
        return 0;  
}  
  
int main()  
{  
    pinInit(PORT_B, 5, OUTPUT);  
    timer0InterruptInit();  
    usartInit(115200);  
  
    enum state_t {LOCKED, NUM1, NUM2, UNLOCKED, BROKEN};  
    enum state_t state = LOCKED;  
  
    pinSetValue(PORT_B, 5, LOW);  
  
    while(1)  
    {  
        switch(state)  
        {  
            case LOCKED:  
                if (checkDigit('1'))  
                {  
                    pinPulsing(PORT_B, 5, 2000, 1);  
  
                    state = NUM1;  
                }  
                else  
                {  
                    pinSetValue(PORT_B, 5, HIGH);  
  
                    state = BROKEN;  
                }  
                break;  
            case NUM1:  
                if (checkDigit('2'))  
                {  
                    pinPulsing(PORT_B, 5, 2000, 1);  
  
                    state = NUM2;  
                }  
                else  
                {  
                    pinSetValue(PORT_B, 5, HIGH);  
  
                    state = BROKEN;  
                }  
        }  
    }  
}
```

```

    }
    break;
case NUM2:
    if (checkDigit('3'))
    {
        pinPulsing(PORT_B, 5, 2000, 3);

        state = UNLOCKED;
    }
    else
    {
        pinSetValue(PORT_B, 5, HIGH);

        state = BROKEN;
    }
    break;
case UNLOCKED:
    break;
case BROKEN:
    break;
}
}
return 0;
}

```

Listing 11.5: Implementacija upotrebom mašina stanja i funkcija za ulaze i izlaze

11.4 Primena mašina stanja u ostvarivanju pseudo-paralelnog načina izvršavanja

Mašine stanje se često koriste i za ostvarivanje naizgled istovremenog izvršavanja više različitih funkcija. Naime, njihovom upotrebom je moguće ostvariti pseudo-paralelni rad (eng. *pseudo-multitasking*) na embeded platformama bez upotrebe prekida i operativnog sistema. Pri tome, potrebno je obratiti pažnju da je ovde reč o *prividnom* paralelizmu, budući da se govori o sistemima koji poseduju jedan procesor, kod kojih ne postoji mogućnost pravog paralelizma.

Osnovna ideja pristupa se zasniva na particonisanju programa na više nezavisnih zadataka (stanja) koji se izvršavaju ciklično u vremenskom multipleksu, gde se u svakom trenutku izvršava samo jedan zadatak, pri čemu je prioritet izvršavanja svakog zadatka isti. Zapravo, ovaj način izvršavanja predstavlja poznatu *round-robin* softversku arhitekturu. Detaljnije o njoj je moguće pronaći u Delu III.

Međutim, trajanje izvršavanja pojedinačnih zadataka može biti različito prilikom svakog prolaza kroz stanja mašine, u zavisnosti od složenosti pojedinih koraka unutar stanja. U opštem slučaju, ovo nije idealno ukoliko je potrebno postići pseudo-paralelno izvršavanje. Naime, radi postizanja pseudo-paralelnog izvrša-

vanja, važna stvar jeste ta, da se svako stanje unutar mašine stanja dovoljno brzo izvršava. Dodatno, poželjno je da stanja u kojima se mašina nalazi *ne budu* blokiraajuća. Na ovaj način, obezbeđuje se ujednačeno vreme provedeno unutar svakog stanja. Na kraju, što se samog načina implementacije mašine stanja tiče, on ostaje nepromenjen.

Čest razlog sporijeg izvršavanja nekog stanja jesu petlje. Kako bi se to prevazišlo, vrši se tzv. *dekompozicija* petlji (*for*, *while*, *do-while*) unutar mašine stanja, čime se otvara mogućnost naizgled istovremenog izvršavanja više petlji u okviru iste niti izvršavanja programa.

U narednoj tabeli data su osnovna pravila dekompozicije.

Tabela 11.3: Osnovna pravila dekompozicije

Tip petlje	Način dekompozicije
<code>for(init; test; fin) body;</code>	<pre>case STATE0: init; state = STATE1; break; case STATE1: if (test) { body; fin; } else state = STATE_NXT; break;</pre>
<code>while (test) body;</code>	<pre>case STATE0: if (test) body; else state = STATE_NXT; break;</pre>
<code>do body; while (test);</code>	<pre>case STATE0: body; if (!test) state = STATE_NXT; break;</pre>

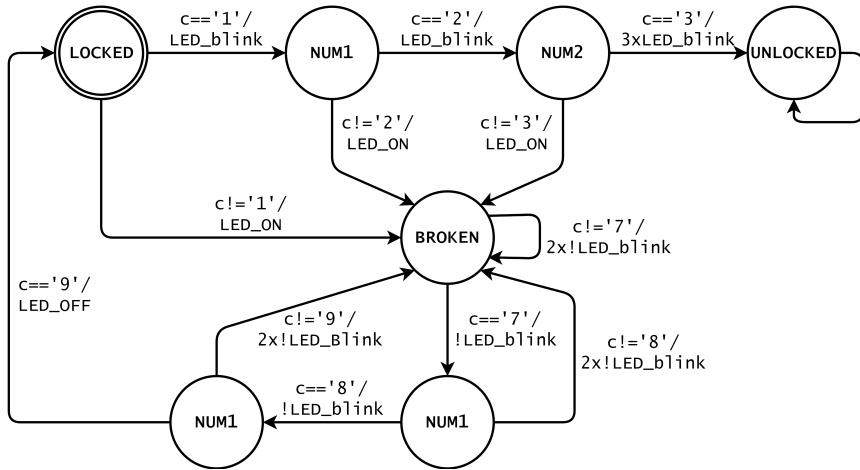
Kao što se može videti iz tabele 11.3, za dekompoziciju *for* petlje neophodna su dva stanja (jedno za inicijalizaciju, a drugo za test i finalizaciju). Sa druge strane, za dekompoziciju *while* i *do-while* petlje dovoljno je samo jedno stanje. Stanje STATE_NXT, označava proizvoljno stanje (različito od stanja STATE0 i STATE1) u mašini stanja, odnosno situaciju u kojoj se napušta petlja.

Detaljno teorijsko objašnjenje pseudo-paralelnog načina izvršavanja je moguće

pronaći u Delu III.

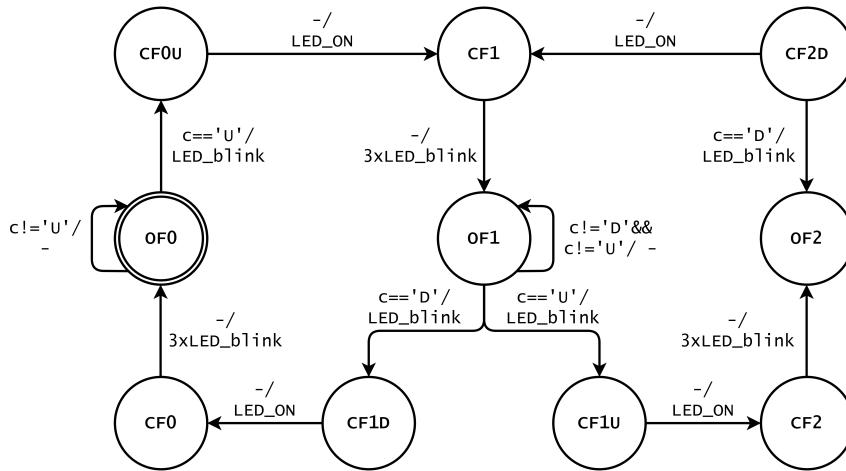
11.5 Zadaci za vežbu

Zadatak 11.5.1. Modifikovati primer sa listinga 11.4 tako da reprezentuje mašinu stanja sa slike 11.4. Modifikovani sef će, nakon unetih izmena, sadržati opciju za ponovnu mogućnost otključavanja nakon unosa pogrešne šifre. Ovim se postiže da sef nakon unosa pogrešne šifre nije trajno onesposobljen. Ponovno osposobljavanje se vrši unosom šifre za oporavak "789". Ispravan unos cifre u okviru šifre za oporavak se označava inverznim treptajem, odnosno treptaj se implementira gašenjem diode, budući da je po pravilu u neispravnom stanju dioda neprestano uključena. Nakon ispravnog unosa cifre, omogućava se unos nove, sve dok se ispravno ne unesu sve tri cifre za oporavak. Nakon toga, prelazi se u zaključano stanje i dioda se gasi. Ukoliko se prilikom oporavka unese pogrešna cifra, vrši se signalizacija dvostrukim inverznim treptajem.



Slika 11.4: Modifikovana mašina stanja za otključavanje sefa

Zadatak 11.5.2. Na slici 11.5 je data mašina stanja kontrolera jednostavnog lifta. U liftu je omogućeno neposredno kretanje za jedan sprat dole ili gore, ukoliko je to moguće. Stanja su kodovana na način da, prvi karakter označava da li su vrata otvorena (O) ili zatvorena (C). Drugi karakter označava sprat (eng. floor), a treći redni broj sprata. Konačno, četvrti karakter označava da li se lift kreće na gore (U), na dole (D) ili će se u sledećem trenutku otključati (bez karaktera). Implementirati datu mašinu stanja.



Slika 11.5: Mašina stanja kontrolera lifta

Zadatak 11.5.3. Konstruisati i implementirati mašinu stanja koja omogućava pretragu šablonu "abba" u nizu karaktera prosleđenom od strane korisnika putem serijskog terminala. Prilikom pronađaska odgovarajućeg šablonu, potrebno je putem serijskog terminala ispisati indeks elementa unetog niza sa kojim započinje zadati šablon. Uzeti u obzir da je moguća pojava više šablonu u okviru unetog niza. Primer unosa je dat u nastavku.

ULAZ: aabbacdabbbbabbaabba

IZLAZ: 1,10,15

Zadatak 11.5.4. Konstruisati i implementirati mašinu stanja koja omogućava konverziju niza karaktera, unetog putem serijskog terminala, u realan broj. U cilju verifikacije mašine stanja, konvertovanu vrednost je potrebno ispisati putem serijskog terminala. Mašina treba da poseduje pet stanja INPUT, INTEGER, DECIMAL, PRINT i ERROR. U okviru stanja INPUT, potrebno je izvršiti unos niza karaktera koji je potrebno konvertovati u realan broj. Unutar stanja INTEGER i DECIMAL se vrši konverzija celobrojnog i razlomljenog dela unetog niza karaktera. Konačno, u okviru stanja PRINT se vrši ispis konvertovane vrednosti, dok stanje ERROR predstavlja stanje greške, u kom se mašina nalazi u slučaju pogrešnog formata ulaznog niza. Primer unosa je dat u nastavku.

IZLAZ: Unesite niz karaktera:

ULAZ: 10.09

IZLAZ: Uneli ste broj:10.09

IZLAZ: Unesite niz karaktera:

ULAZ: -10.09

IZLAZ: Uneli ste broj:-10.09

IZLAZ: Unesite niz karaktera:

ULAZ: +10

IZLAZ: Uneli ste broj:10

Zadatak 11.5.5. Implementirati sledeće funkcije bez upotrebe petlji (koristiti pravila dekompozicije iz Tabele 11.3). Jedina petlja koja može da se koristi jeste petlja – `while(1)`.

- `int16_t SumGeo(int16_t first_element, int16_t ratio, int16_t n, int8_t mode)`
 - **Opis:** Funkcija koja izračunava sumu prvih n članova geometrijskog niza opisanog pomoću početne vrednosti niza i odnosa, na osnovu parametra `mode`. Ukoliko `mode` ima vrednost `ITERATIVE`, vrednost sume je potrebno odrediti iterativno (sumom svih elemenata do n -tog), dok u slučaju vrednosti `FORMULA`, potrebno je odrediti sumu po odgovarajućoj formuli. `ITERATIVE` i `FORMULA` su vrednosti definisane pomoću makro konstante.
 - **Povratna vrednost:** Vrednost sume prvih n članova geometrijskog niza.
- `int16_t SumArith(int16_t first_element, int16_t ratio, int16_t n, int8_t mode)`
 - **Opis:** Funkcija koja izračunava sumu prvih n članova aritmetičkog niza opisanog pomoću početne vrednosti niza i odnosa, na osnovu parametra `mode`. Ukoliko `mode` ima vrednost `ITERATIVE`, vrednost sume je potrebno odrediti iterativno (sumom svih elemenata do n -tog elementa), dok u slučaju vrednosti `FORMULA`, potrebno je odrediti sumu po odgovarajućoj formuli. `ITERATIVE` i `FORMULA` su vrednosti definisane pomoću makro konstante.
 - **Povratna vrednost:** Vrednost sume prvih n članova aritmetičkog niza.
- `int16_t EvalPolynomial(int8_t order, int8_t *coeff, int8_t x);`
 - **Opis:** Funkcija koja izračunava vrednost polinoma zadatog njegovim redom (parametar `order`) i koeficijentima (parametar `coeff`) u tački `x` (parametar `x`).
 - **Povratna vrednost:** Vrednost polinoma u tački `x`.
- `int32_t Fibonacci(int8_t number);`
 - **Opis:** Funkcija koja implementira algoritam za računanje n -tog člana Fibonačijevog niza.
 - **Povratna vrednost:** Vrednost n -tog člana Fibonačijevog niza.

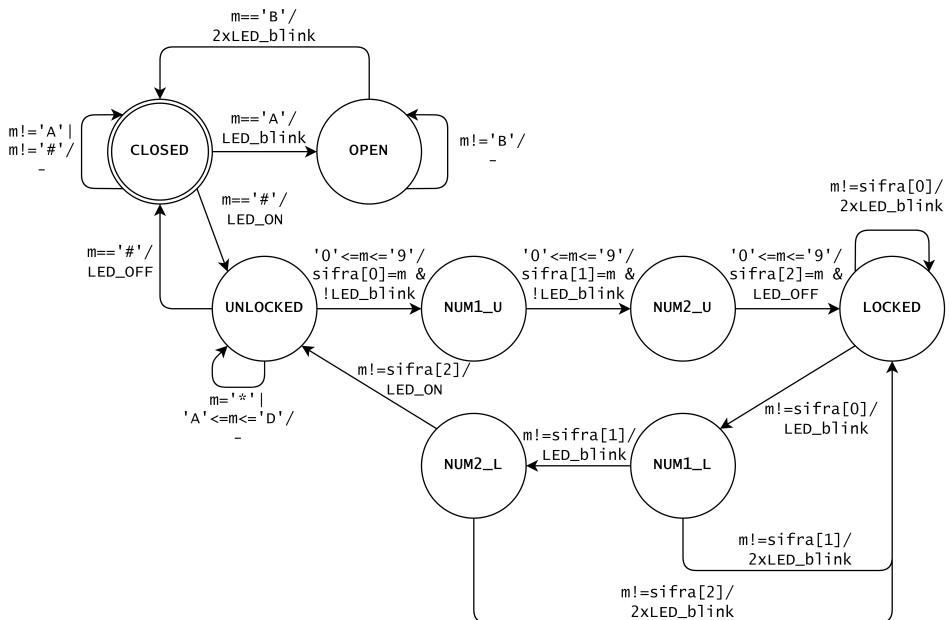
Zadatak 11.5.6. Konstruisati i implementirati mašinu stanja koja omogućava izvršavanje više zadataka (eng. *task*) istovremeno, odnosno tzv. pseudo-paralelni način izvršavanja (eng. *pseudo-multitasking*) zadataka. Pri tome, treba napomenuti da se ovi zadaci neće zaista izvršavati u paraleli, već u *vremenskom multipleksu*. Ovo podrazumeva da se u svakom trenutku izvršava isključivo jedan zadatak, pri čemu se, pod unapred odredenim pravilima, izvršavanje prebacuje sa jednog zadataka na drugi. Detaljnije teorijsko objašnjenje ovakvog načina izvršavanja je moguće pronaći u Delu III.

Mašina stanja, koju je potrebno implementirati u okviru ovog zadataka, treba da omogući izvršavanje zadataka navedenih u nastavku.

1. Unos vrednosti periode treperenja diode od strane korisnika, putem serijskog terminala. Vrednost unetu putem serijskog terminala je potrebno smestiti unutar promenljive `blink_period`.
2. Treperenje ugrađene diode na Arduino platformi svakih `blink_period` sekundi. Parametar `blink_period` se podešava u okviru zadatka 1.
3. Očitavanje vrednosti sa analognog ulaza 0 svakih *1s*.
4. Ispis vrednosti sa analognog ulaza 0 putem serijskog terminala svakih *2s*.
5. Isključivanje treperenja diode u slučaju da je vrednost sa analognog ulaza 0 veća od *512*.

Prilikom izvršavanja navedenih stanja, kako bi se ostvarilo pseudo-paralelno izvršavanje, neophodno je obezbediti da *ne postoji* blokirajuće stanje, odnosno stanje kod kog je uslov tranzicije pojava nekog događaja.

Zadatak 19.2.10. Na slici 19.5 je data mašina stanja koja implementira jednostavni sef. Na početku, sef se nalazi u zatvorenom stanju (CLOSED). Pritiskom na taster 'A', sef se otvara, odnosno prelazi u otvoreno stanje (OPEN). Pritiskom na taster 'B', sef se ponovo zatvara. Pritiskom na taster '#', sef prelazi u otključano stanje (UNLOCKED), u okviru kog je moguće započeti izbor trocifrene šifre. Kada se sef nađe u zaključanom stanju (LOCKED), potrebno je ispravno uneti izabranu trocifrenu šifru, kako bi se ono napustilo. Promenljiva m predstavlja ulaz matrične tastature. LED_ON, LED_OFF i LED_BLINK predstavljaju stanja ugrađene diode na Arduino UNO platformi.



Slika 19.5: Jednostavni sef upravljan pomoću matrične tastature