Županijska razina natjecanja mladih tehničara

Automatika

Pripreme

4-znamenkasti 7-segmentni LED displej

Multipleksiranje i organizacija prikaza podataka

Školska godina 2025. / 2026.

Radni materijal za pripremu učenika osnovnih škola

Hrvoje Vrhovski, Hrvatska zajednica tehničke kulture

HRVATSKA ZAJEDNICA TEHNIČKE KULTURE

Sadržaj

Uvod

Dosad smo se upoznali sa **7-segmentnim LED displejem**. Znamo da se sastoji od sedam svjetlećih segmenata označenih slovima a–g te često i jedne dodatne svjetleće diode za prikaz decimalne točke (dp).

U ovoj skripti obrađujemo upravljanje četveroznamenkastim 7-segmentnim LED displejem, kakav se često koristi u projektima iz područja automatike i mikroupravljačkih sustava.

Napomena:
Svi programski primjeri u skripti pisani su i ispitani u Arduino IDE okruženju uz korištenje DASDUINO CORE mikroupravljačkog sučelja.

Ako koristite klasični Arduino core, kod će u pravilu raditi jednako, ali nazivi pločica i odabir procesora u izborniku mogu se razlikovati.

Cilj je postupno razumjeti:

- kako radi jedna znamenka,
- kako se upravlja višeznamenkastim displejem,
- zašto je potrebno multipleksiranje,
- te kako prikaz organizirati pomoću polja podataka i maski segmenata.

Gradivo je strukturirano tako da učenik najprije razumije osnovni princip rada segmenata, zatim uoči problem zajedničkih vodova, a potom samostalno dođe do rješenja kroz postupno uvođenje multipleksiranja.

Nakon toga prelazimo s “ručne” kontrole pinova na rad s podacima (maskama i poljima), što je pristup koji se koristi u ozbiljnijim i natjecateljskim zadacima.

U završnom dijelu skripte povezujemo prikaz s podacima dobivenim iz stvarnog sustava — s analognih ulaza mikroupravljača. Time se cjelina zaokružuje: broj više nije unaprijed zadan u programu, nego dolazi sa senzora, obrađuje se i zatim prikazuje na displeju.

Na taj način učenik prolazi cjelovit put:
mjerenje → obrada podataka → organizacija prikaza → stabilan prikaz.

Takav pristup odražava tipične zadatke iz područja automatike i čini temelj za uspješno rješavanje složenijih natjecateljskih zadataka.

1. 7-segmentni LED displej

Postoje dvije osnovne izvedbe takvih displeja:

- displej sa **zajedničkom katodom (Common Cathode – CC)**
- displej sa **zajedničkom anodom (Common Anode – CA)**

Razlika između njih je u načinu spajanja zajedničkog izvoda i logici upravljanja segmentima.

(a) 1. Displej sa zajedničkom katodom (CC)

Kod displeja sa zajedničkom katodom:
- svi segmenti i decimalna točka spajaju se na pojedinačne digitalne izvode mikroupravljača,
- zajednička katoda spaja se na uzemljenje (GND),
- pojedini segment svijetli kada je njegov izvod postavljen u stanje **HIGH (logička 1)**.

Dakle, kod CC displeja vrijedi pravilo: **segment ON = HIGH**

(b) 2. Displej sa zajedničkom anodom (CA)

Kod displeja sa zajedničkom anodom:
- svi segmenti i decimalna točka spajaju se na pojedinačne digitalne izvode mikroupravljača,
- zajednička anoda spaja se na +5 V (ili +3,3 V),
- pojedini segment svijetli kada je njegov izvod postavljen u stanje **LOW (logička 0)**.

Dakle, kod CA displeja vrijedi pravilo: **segment ON = LOW**

(c) 3. Upravljanje zajedničkim izvodom putem mikroupravljača

U praksi se zajednički izvod (anoda ili katoda) često također spaja na digitalni izvod mikroupravljača. Time možemo:

- jednostavno uključiti ili isključiti cijeli displej,
- kasnije primijeniti postupak multipleksiranja kod višeznamenkastih displeja.

U tom slučaju vrijedi:

- kod **CC displeja** segmenti mogu svijetliti samo ako je zajednička katoda postavljena na **LOW**, a pojedini segment na **HIGH**
- kod **CA displeja** segmenti mogu svijetliti samo ako je zajednička anoda postavljena na **HIGH**, a pojedini segment na **LOW**

Na taj način cijeli displej možemo ugasiti jednostavnom promjenom stanja zajedničkog izvoda:
- kod CC displeja postavljanjem zajedničke katode na **HIGH**
- kod CA displeja postavljanjem zajedničke anode na **LOW**

(d) Važna napomena

Svaki segment je svjetleća dioda i mora imati serijski otpornik za ograničenje struje (najčešće 220–330 Ω pri napajanju 5 V).

Mini vježba (jedna znamenka): “prikaži 0–9”

Cilj: učenik shvati da je znamenka samo „uzorak segmenata“.

Segmenti koji svijetle za znamenke (bez dp):

- 0: a b c d e f
- 1: b c
- 2: a b d e g
- 3: a b c d g
- 4: b c f g
- 5: a c d f g
- 6: a c d e f g
- 7: a b c
- 8: a b c d e f g
- 9: a b c d f g

Primjer programa:

```
// Jedna 7-seg znamenka (COMMON CATHODE)
// segment ON = HIGH

const uint8_t A = 4;
const uint8_t B = 5;
const uint8_t C = 6;
const uint8_t D = 7;
const uint8_t E = 8;
const uint8_t F = 9;
const uint8_t G = 10;

void setup() {
  pinMode(A, OUTPUT);
  pinMode(B, OUTPUT);
  pinMode(C, OUTPUT);
  pinMode(D, OUTPUT);
  pinMode(E, OUTPUT);
  pinMode(F, OUTPUT);
  pinMode(G, OUTPUT);
}

void allOff() {
  digitalWrite(A, LOW);
  digitalWrite(B, LOW);
  digitalWrite(C, LOW);
  digitalWrite(D, LOW);
  digitalWrite(E, LOW);
  digitalWrite(F, LOW);
```

```
  digitalWrite(G, LOW);
}

void loop() {

  // 0
  allOff();
  digitalWrite(A, HIGH);
  digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
  digitalWrite(D, HIGH);
  digitalWrite(E, HIGH);
  digitalWrite(F, HIGH);
  delay(1000);

  // 1
  allOff();
  digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
  delay(1000);

  // 2
  allOff();
  digitalWrite(A, HIGH);
  digitalWrite(B, HIGH);
  digitalWrite(D, HIGH);
  digitalWrite(E, HIGH);
  digitalWrite(G, HIGH);
  delay(1000);

  // itd...
}
```

Što se događa:

- prije svake znamenke gasimo sve segmente
- zatim palimo samo potrebne
- znamenka je samo “uzorak upaljenih LED-ica”

Pitanja za razmišljanje

1. Što će se dogoditi ako zaboravimo pozvati *allOff()* prije nove znamenke?
2. Koji segmenti moraju svijetliti da bi se dobila znamenka 9?
3. Možemo li ovaj kod skratiti da ne ponavljamo *digitalWrite* za svaku znamenku?

2. 4-znamenkasti 7-segmentni displej

2.1 Što se mijenja u odnosu na jednu znamenku?

Kod jedne znamenke bilo je jednostavno:
- 7 segmenata + decimalna točka
- jedan zajednički izvod (anoda ili katoda)

Ako želimo prikazati četveroznamenkasti broj (npr. 2025), intuitivno bismo mogli pomisliti:
Trebamo 4 × 9 = 36 izvoda mikroupravljača. Naime, ako bismo svaku znamenku upravljali zasebno, trebali bismo:
- 7 segmenata (a–g)
- 1 decimalnu točku (dp)
- 1 zajednički izvod (anoda ili katoda)

Ukupno: **9 vodova po znamenki,** za četiri znamenke to bi značilo:

4 × 9 = 36 izvoda mikroupravljača

Takav pristup je nepraktičan iz dva razloga:
1. većina mikroupravljača nema dovoljan broj dostupnih digitalnih izvoda,
2. program bi bio nepotrebno složen, teško proširiv i sklon pogreškama.

Zbog toga je kod ovog displeja situacija sljedeća:
- svi segmenti iste oznake (a, b, c, d, e, f, g, dp) međusobno su spojeni
- svaka znamenka ima **svoj zajednički izvod (DIG1, DIG2, DIG3, DIG4)**

To znači:
- segment "a" ima samo jedan vod
- segment "b" ima samo jedan vod
- …
- ali postoje 4 upravljačka voda za znamenke

Ukupno:
- 8 vodova za segmente
- 4 voda za znamenke
  = **12 vodova umjesto 36**

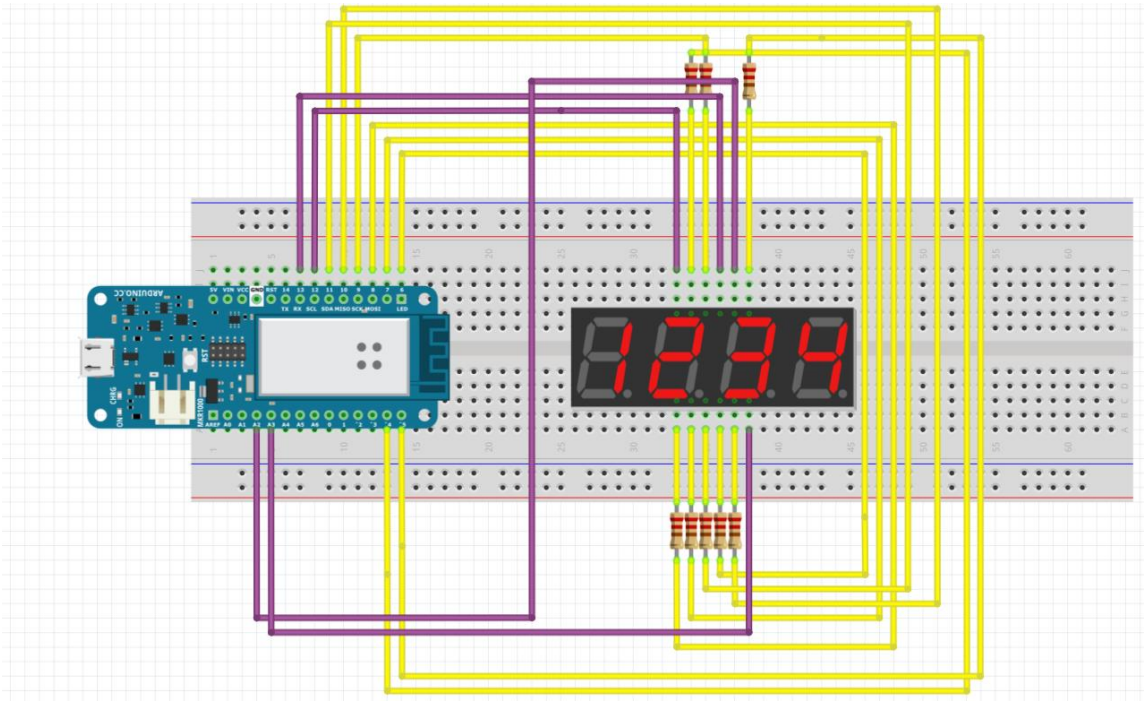
Sada smo smanjili broj potrebnih izvoda mikroupravljača tako da su svi segmenti (a–g i dp) zajednički za sve znamenke, a svaka znamenka ima svoj upravljački izvod (DIG1–DIG4). Time smo riješili problem velikog broja izvoda, ali se pojavljuje novi problem.
- Ako želimo prikazati isti broj na sve četiri znamenke, primjerice **„1111“**, postupak je jednostavan: uključimo segmente koji čine znamenku 1 i aktiviramo sve četiri znamenke. Tada će se na sve četiri pozicije prikazati broj 1.
- Međutim, ako želimo prikazati broj **„1234“**, više ne možemo istovremeno zadati četiri različita prikaza, jer su segmentni vodovi zajednički. Ako postavimo segmente za znamenku 1, tada će (dok su uključene znamenke) sve aktivne znamenke prikazivati 1. Kad zatim postavimo segmente za znamenku 2, sve aktivne znamenke će prikazivati 2, itd.

- Drugim riječima, bez dodatnog postupka upravljanja displejem, ne možemo dobiti stabilan prikaz „1234“ – dobili bismo samo izmjenu „1 pa 2 pa 3 pa 4“ na svim znamenkama.

Na donjoj slici prikazana je **okvirna shema spajanja**. U primjeru je za prikaz korišteno mikroupravljačko sučelje **MKR1000 WiFi**, ali ono je ovdje samo radi prikaza povezivanja izvoda.
Bez obzira koje sučelje koristite, obratite posebnu pozornost na **ispravno povezivanje displeja**. Ako displej spojite na druge izvode mikroupravljača, u programu je potrebno **promijeniti definicije pinova** (dio koda u kojem određujete koji je izvod displeja spojen na koji izvod mikroupravljača). U protivnom prikaz neće raditi ispravno ili će znamenke biti na krivim mjestima.

Opisani način povezivanja primjenjivat će se i u svim ostalim vježbama u ovoj skripti.

Tijekom svih vježbi koristi se ista hardverska konfiguracija. Time se omogućuje da se usmjerimo na razumijevanje programskog dijela zadatka (upravljanje segmentima, logika prikaza i multipleksiranje), bez dodatnog opterećenja promjenama u spajanju sklopa. Takav pristup olakšava uočavanje uzročno-posljedične veze između promjene u programu i promjene na prikazu.



2.2 Primjer A: „1111“ (jednostavno)

Ideja: segmenti se postave na uzorak za “1”, a zatim se aktiviraju sve znamenke.
Primjer koda (za tipični CC gdje je DIG aktivan na LOW):

```
// COMMON CATHODE tipično: segment ON=HIGH, digit ON=LOW
// Segment izvodi (prilagoditi svojem spajanju)
const uint8_t A = 4;
const uint8_t B = 5;
const uint8_t C = 6;
```

```
const uint8_t D = 7;
const uint8_t E = 8;
const uint8_t F = 9;
const uint8_t G = 10;
const uint8_t DP = 11; // ako koristimo decimalnu točku

// Digit (zajednički) izvodi (prilagoditi svojem modulu/spajanju)
const uint8_t DIG1 = 12;
const uint8_t DIG2 = 13;
const uint8_t DIG3 = A2;
const uint8_t DIG4 = A3;

// --- POMOĆNE FUNKCIJE ---

void jedan() { // b c
        digitalWrite(B, HIGH);
        digitalWrite(C, HIGH);
}

void setup() {
    pinMode(A, OUTPUT);
    pinMode(B, OUTPUT);
    pinMode(C, OUTPUT);
    pinMode(D, OUTPUT);
    pinMode(E, OUTPUT);
    pinMode(F, OUTPUT);
    pinMode(G, OUTPUT);
    pinMode(DP, OUTPUT);
    pinMode(DIG1, OUTPUT);
    pinMode(DIG2, OUTPUT);
    pinMode(DIG3, OUTPUT);
    pinMode(DIG4, OUTPUT);

    digitalWrite(A, LOW);
    digitalWrite(B, LOW);
    digitalWrite(C, LOW);
    digitalWrite(D, LOW);
    digitalWrite(E, LOW);
    digitalWrite(F, LOW);
    digitalWrite(G, LOW);
    digitalWrite(DP, LOW);

    digitalWrite(DIG1, HIGH);
    digitalWrite(DIG2, HIGH);
    digitalWrite(DIG3, HIGH);
    digitalWrite(DIG4, HIGH);


}

void loop() {
    // Primjer: prikaži "1 1 1 1"
    digitalWrite(DIG1, LOW);
```

```
    digitalWrite(DIG2, LOW);
    digitalWrite(DIG3, LOW);
    digitalWrite(DIG4, LOW);
    jedan();
}
```

Napomena: U ovom primjeru radi jednostavnosti u *setup()* smo prvo ugasili sve segmente (A…G i DP na LOW), pa u funkciji *jedan()* palimo samo potrebne segmente (B i C). U kasnijim primjerima uvodimo funkciju *allSegmentsOff()* kako bismo prije svake znamenke krenuli iz "čistog stanja" i izbjegli neželjene ostatke prethodnog prikaza.

2.3 Primjer B: „1234" (problem)

Pokušajmo ovako:

```
// COMMON CATHODE tipično: segment ON=HIGH, digit ON=LOW

// Segment izvodi (prilagoditi svojem spajanju)
const uint8_t A = 4;
const uint8_t B = 5;
const uint8_t C = 6;
const uint8_t D = 7;
const uint8_t E = 8;
const uint8_t F = 9;
const uint8_t G = 10;
const uint8_t DP = 11; // ako koristimo decimalnu točku

// Digit (zajednički) izvodi (prilagoditi svojem modulu/spajanju)
const uint8_t DIG1 = 12;
const uint8_t DIG2 = 13;
const uint8_t DIG3 = A2;
const uint8_t DIG4 = A3;

// --- POMOĆNE FUNKCIJE ---

void allSegmentsOff() { // CC: OFF = LOW
  digitalWrite(A, LOW);
  digitalWrite(B, LOW);
  digitalWrite(C, LOW);
  digitalWrite(D, LOW);
  digitalWrite(E, LOW);
  digitalWrite(F, LOW);
  digitalWrite(G, LOW);
  digitalWrite(DP, LOW);
}
void jedan() { // b c
  allSegmentsOff();
  digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
  delay(500);
}
```

```
void dva() { // a b d e g
  allSegmentsOff();
  digitalWrite(A, HIGH);
  digitalWrite(B, HIGH);
  digitalWrite(D, HIGH);
  digitalWrite(E, HIGH);
  digitalWrite(G, HIGH);
  delay(500);
}

void tri() { // a b c d g
  allSegmentsOff();
  digitalWrite(A, HIGH);
  digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
  digitalWrite(D, HIGH);
  digitalWrite(G, HIGH);
  delay(500);
}

void cetiri() { // b c f g
  allSegmentsOff();
  digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
  digitalWrite(F, HIGH);
  digitalWrite(G, HIGH);
  delay(500);
}

void setup() {
  pinMode(A, OUTPUT);
  pinMode(B, OUTPUT);
  pinMode(C, OUTPUT);
  pinMode(D, OUTPUT);
  pinMode(E, OUTPUT);
  pinMode(F, OUTPUT);
  pinMode(G, OUTPUT);
  pinMode(DP, OUTPUT);
  pinMode(DIG1, OUTPUT);
  pinMode(DIG2, OUTPUT);
  pinMode(DIG3, OUTPUT);
  pinMode(DIG4, OUTPUT);
}

void loop() {
  // Primjer: prikaži "1 2 3 4"
  digitalWrite(DIG1, LOW);
  digitalWrite(DIG2, LOW);
  digitalWrite(DIG3, LOW);
  digitalWrite(DIG4, LOW);
  jedan();
  dva();
  tri();
```

```
  cetiri();
}
```

Sada se na sve 4 znamenke **istovremeno** prikazuje najprije 1, zatim 2, zatim 3, zatim 4. Znači, umjesto stabilnog prikaza **1234**, dobivamo "1111 → 2222 → 3333 → 4444". To se događa zato što su segmenti zajednički za sve znamenke, u svakom trenutku sve aktivne znamenke dobiju isti uzorak segmenata.

Što sad? Kako prikazati različite znamenke kad su segmenti zajednički?
Rješenje je da u jednom trenutku uključimo samo jednu znamenku, ali taj postupak ponavljamo dovoljno brzo u petlji:

- Aktiviramo prvu znamenku (DIG1) i postavimo segmente za broj „1". Zadržimo prikaz kratko *vrijemePrikaza* i zatim ugasimo znamenku.
- Aktiviramo drugu znamenku (DIG2) i postavimo segmente za broj „2". Zadržimo prikaz *vrijemePrikaza* i zatim ugasimo znamenku.
- Aktiviramo treću znamenku (DIG3) i postavimo segmente za broj „3". Zadržimo prikaz *vrijemePrikaza* i zatim ugasimo znamenku.
- Aktiviramo četvrtu znamenku (DIG4) i postavimo segmente za broj „4". Zadržimo prikaz *vrijemePrikaza* i zatim ugasimo znamenku.

Ako se ovaj ciklus ponavlja dovoljno brzo (npr. svakih nekoliko milisekundi), ljudsko oko će ga doživjeti kao stabilan prikaz broja „1234".

Primjer varijable:

uint16_t vrijemePrikaza = 5; // ms

U jednom trenutku svijetli samo jedna znamenka, ali dovoljno brzo izmjenjujemo sve četiri tako da ljudsko oko vidi stabilan prikaz sve 4 znamenke odjednom.

2.4 Rješenje:

```
// 4-digit 7-seg – paljenje znamenki jedna po jedna (bez pravog multipleksa)
// COMMON CATHODE tipično: segment ON=HIGH, digit ON=LOW

// Segment izvodi (prilagodi svojem spajanju)
const uint8_t A = 4;
const uint8_t B = 5;
const uint8_t C = 6;
const uint8_t D = 7;
const uint8_t E = 8;
const uint8_t F = 9;
const uint8_t G = 10;
const uint8_t DP = 11; // ako koristimo decimalnu točku

// Digit (zajednički) izvodi (prilagodimo svojem modulu/spajanju)
const uint8_t DIG1 = 12;
const uint8_t DIG2 = 13;
const uint8_t DIG3 = A2;
const uint8_t DIG4 = A3;
```

```
uint16_t vrijemePrikaza = 5;   // trajanje uključenja znamenke (ms) - probaj: 200, 50, 10, 5, 2 (ms)
uint16_t vrijemePauze   = 0;   // kratka pauza između znamenki (ms)

// --- POMOĆNE FUNKCIJE ---

void allSegmentsOff() { // CC: OFF = LOW
 digitalWrite(A, LOW);
 digitalWrite(B, LOW);
 digitalWrite(C, LOW);
 digitalWrite(D, LOW);
 digitalWrite(E, LOW);
 digitalWrite(F, LOW);
 digitalWrite(G, LOW);
 digitalWrite(DP, LOW);
}

void allDigitsOff() { // CC tipično: digit OFF = HIGH
 digitalWrite(DIG1, HIGH);
 digitalWrite(DIG2, HIGH);
 digitalWrite(DIG3, HIGH);
 digitalWrite(DIG4, HIGH);
}

// Odabir znamenke (upaljena je samo jedna)
void digit1() { allDigitsOff(); digitalWrite(DIG1, LOW); }
void digit2() { allDigitsOff(); digitalWrite(DIG2, LOW); }
void digit3() { allDigitsOff(); digitalWrite(DIG3, LOW); }
void digit4() { allDigitsOff(); digitalWrite(DIG4, LOW); }

// --- ZNAMENKE (segmenti) ---
// Sve funkcije prvo ugase segmente pa upale potrebne (CC: ON=HIGH)

void nula() { // a b c d e f
 allSegmentsOff();
 digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
 digitalWrite(D, HIGH); digitalWrite(E, HIGH); digitalWrite(F, HIGH);
}

void jedan() { // b c
 allSegmentsOff();
 digitalWrite(B, HIGH); digitalWrite(C, HIGH);
}

void dva() { // a b d e g
 allSegmentsOff();
 digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(D, HIGH);
 digitalWrite(E, HIGH); digitalWrite(G, HIGH);
}

void tri() { // a b c d g
 allSegmentsOff();
```

```
  digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(G, HIGH);
}

void cetiri() { // b c f g
  allSegmentsOff();
  digitalWrite(B, HIGH); digitalWrite(C, HIGH); digitalWrite(F, HIGH);
  digitalWrite(G, HIGH);
}

void pet() { // a c d f g
  allSegmentsOff();
  digitalWrite(A, HIGH); digitalWrite(C, HIGH); digitalWrite(D, HIGH);
  digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}

void sest() { // a c d e f g
  allSegmentsOff();
  digitalWrite(A, HIGH); digitalWrite(C, HIGH); digitalWrite(D, HIGH);
  digitalWrite(E, HIGH); digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}

void sedam() { // a b c
  allSegmentsOff();
  digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
}

void osam() { // a b c d e f g
  allSegmentsOff();
  digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(E, HIGH); digitalWrite(F, HIGH);
  digitalWrite(G, HIGH);
}

void devet() { // a b c d f g
  allSegmentsOff();
  digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}

// Prikaz znamenke pozivom odgovarajuće funkcije (bez matrice)
void prikaziZnamenku(uint8_t x) {
  switch (x) {
    case 0: nula(); break;
    case 1: jedan(); break;
    case 2: dva(); break;
    case 3: tri(); break;
    case 4: cetiri(); break;
    case 5: pet(); break;
    case 6: sest(); break;
    case 7: sedam(); break;
    case 8: osam(); break;
    case 9: devet(); break;
```

```
    default: allSegmentsOff(); break; // “prazno”
  }
}

void setup() {
  pinMode(A, OUTPUT); pinMode(B, OUTPUT); pinMode(C, OUTPUT); pinMode(D,
OUTPUT);
  pinMode(E, OUTPUT); pinMode(F, OUTPUT); pinMode(G, OUTPUT);
  pinMode(DP, OUTPUT);
  pinMode(DIG1, OUTPUT); pinMode(DIG2, OUTPUT); pinMode(DIG3, OUTPUT);
pinMode(DIG4, OUTPUT);
  allSegmentsOff();
  allDigitsOff();
}

void loop() {
  // Primjer: prikaži "1 2 3 4" – ali SVAKU znamenku zasebno, sporo i vidljivo
  digit1();
  prikaziZnamenku(1);
  delay(vrijemePrikaza);
  allDigitsOff();
  allSegmentsOff(); //namjerno stavljeno, da se vidi odvajanje znamenki. Za "mekši"
prijelaz", može se ostaviti samo allDigitsOff()
  delay(vrijemePauze);

  digit2();
  prikaziZnamenku(2);
  delay(vrijemePrikaza);
  allDigitsOff();
  allSegmentsOff(); //namjerno stavljeno, da se vidi odvajanje znamenki. Za "mekši"
prijelaz", može se ostaviti samo allDigitsOff()
  delay(vrijemePauze);

  digit3();
  prikaziZnamenku(3);
  delay(vrijemePrikaza);
  allDigitsOff();
  allSegmentsOff(); //namjerno stavljeno, da se vidi odvajanje znamenki. Za "mekši"
prijelaz", može se ostaviti samo allDigitsOff()
  delay(vrijemePauze);

  digit4();
  prikaziZnamenku(4);
  delay(vrijemePrikaza);
  allDigitsOff();
  allSegmentsOff(); //namjerno stavljeno, da se vidi odvajanje znamenki. Za "mekši"
prijelaz", može se ostaviti samo allDigitsOff()
  delay(vrijemePauze);
}
```

U gornjem primjeru možemo mijenjati vrijeme prikaza u varijabli:

uint16_t vrijemePrikaza = 50;

Pokušajte s drugim vrijednostima…

2.5 Pojašnjenje koda

Definicije pinova

const uint8_t A = 4; ... const uint8_t DP = 11;
const uint8_t DIG1 = 12; ... const uint8_t DIG4 = A3;

- Varijable A…G i DP predstavljaju izvode mikroupravljačkog sučelja na koje su spojeni segmenti displeja.
- Varijable DIG1…DIG4 su izvodi mikroupravljačkog sučelja koji upravljaju zajedničkim izvodom svake znamenke (koja znamenka je "aktivna").

Zašto *uint8_t?*

Zato što je to tip podataka (0–255) idealan za brojeve izvoda i zauzima malo memorije.

- **Varijable za "brzinu" demonstracije**

uint16_t vrijemePrikaza = 5;
uint16_t vrijemePauze = 0;

- vrijemePrikaza određuje koliko dugo (u ms) ostaje upaljena jedna znamenka.
- vrijemePauze je dodatna pauza između znamenki.

Ovo je namjerno "laboratorijska" postavka: mijenjanjem *vrijemePrikaza* promatramo razliku između **sporog prebacivanja** i dojma **stabilnog prikaza**.

- **allSegmentsOff() – gašenje segmenata**

```
void allSegmentsOff() { // CC: OFF = LOW
  digitalWrite(A, LOW); ... digitalWrite(DP, LOW);
}
```

Kod **common cathode (CC)**:

- segment svijetli kad je na HIGH
- zato je gašenje segmenta LOW

Ova funkcija služi da uvijek krenemo iz "čistog stanja", bez neželjenih ostataka prethodne znamenke.

- **allDigitsOff() – gašenje svih znamenki**

```
void allDigitsOff() { // CC tipično: digit OFF = HIGH
```

```
 digitalWrite(DIG1, HIGH); ... digitalWrite(DIG4, HIGH);
}
```

Kod većine CC 4-digit modula vrijedi:

- znamenka je aktivna na LOW
- zato je isključena na HIGH

To znači: dok su svi DIG izvodi na HIGH, **nijedna znamenka nije aktivna**.

- **digit1()...digit4() – odabir znamenke**

```
void digit1() {
        allDigitsOff();
        digitalWrite(DIG1, LOW);
}
```

Svaka od ovih funkcija radi isto:

1. prvo ugasi sve znamenke (allDigitsOff())
2. zatim uključi točno jednu znamenku (postavi njen DIG pin na LOW)

Tako osiguravamo pravilo:

u jednom trenutku aktivna je samo jedna znamenka.

Funkcije nula()...devet() – "ručno" paljenje segmenata

Primjer:

```
void dva() { // a b d e g
        allSegmentsOff();
        digitalWrite(A, HIGH); digitalWrite(B, HIGH); ...
}
```

Svaka funkcija:

1. ugasi sve segmente (allSegmentsOff())
2. upali samo one segmente koji tvore željenu znamenku

Komentar uz funkciju (npr. // a b d e g) odmah govori **koji segmenti moraju svijetliti**.

- **prikaziZnamenku(x) – odabir znamenke preko switch**

```
void prikaziZnamenku(uint8_t x) {
        switch (x) {
                case 0: nula(); break;
        ...
        }
```

}

Ovo je “preklopnik”:

- ako je x=3, pozove se tri()
- ako je x=7, pozove se sedam()

Prednost: u loop() pišemo *prikaziZnamenku(4)* umjesto da ručno zovemo cetiri().

- **setup() – priprema izvoda**

pinMode(A, OUTPUT); ... pinMode(DIG4, OUTPUT);
allSegmentsOff();
allDigitsOff();

U setup():

- sve izvode koje koristimo postavljamo kao izlaze (OUTPUT)
- zatim ugasimo segmente i znamenke, da displej ne “zasvijetli slučajno” pri pokretanju

- **loop() – demonstracija “1 2 3 4” znamenku po znamenku**

Primjer prve znamenke:
digit1();
    prikaziZnamenku(1);
    delay(vrijemePrikaza);
    allDigitsOff();
    allSegmentsOff();
    delay(vrijemePauze);

Što se događa:

1. digit1() aktivira prvu znamenku (DIG1)
2. prikaziZnamenku(1) postavlja segmente tako da se prikaže “1”
3. delay(vrijemePrikaza) drži prikaz određeno vrijeme
4. allDigitsOff() ugasi sve znamenke
5. allSegmentsOff() ugasi segmente
    - **namjerno**: da se jasnije vidi odvajanje znamenki
    - za “mekši” prijelaz može se ostaviti samo allDigitsOff()
6. delay(vrijemePauze) (opcionalno) doda pauzu

Isto se ponavlja za DIG2, DIG3 i DIG4.

U ovom programu već radimo princip “jedna znamenka u trenutku”, ali još ne govorimo o pravom multipleksiranju jer koristimo delay(). U pravom multipleksu prebacivanje znamenki mora biti brzo i vremenski kontrolirano bez zaustavljanja programa.

3. Na tragu multipleksiranja

U prethodnom programu svaku znamenku uključujemo zasebno i namjerno koristimo delay() kako bismo **vidjeli** prebacivanje između znamenki. Kada *vrijemePrikaza* smanjujemo (npr. s 50 ms na 10 ms, 5 ms ili 2 ms), prikaz počinje izgledati sve stabilnije. To znači da smo već **na tragu multipleksiranja**:
u jednom trenutku je uključena samo jedna znamenka, ali se znamenke izmjenjuju dovoljno brzo da oko doživljava prikaz kao istovremen.
Međutim, ovakav pristup s delay() nije prikladan za konačno rješenje jer delay() zaustavlja izvođenje programa pa mikroupravljač u tom vremenu ne može obavljati druge zadatke (npr. čitanje tipkala, serijsku komunikaciju, mjerenje senzora).
U pravom multipleksiranju cilj je postići takvu brzinu osvježavanja da nema vidljivog treperenja, ali i da program i dalje može obavljati druge zadatke. U sljedećem koraku napravit ćemo **pravo multipleksiranje**, gdje se prebacivanje znamenki odvija vremenski kontrolirano (npr. pomoću micros()), bez zaustavljanja programa.

U multipleksu je u jednom trenutku aktivna samo jedna znamenka. Stabilan prikaz dobivamo brzim ponavljanjem ciklusa.

Primjer koda s multipleksiranjem:

```
// 4-digit 7-seg – PRAVO multipleksiranje (bez delay)
// COMMON CATHODE: segment ON = HIGH, digit ON = LOW

// ---------------- PINOVI ----------------

// Segment pinovi
const uint8_t A  = 4;
const uint8_t B  = 5;
const uint8_t C  = 6;
const uint8_t D  = 7;
const uint8_t E  = 8;
const uint8_t F  = 9;
const uint8_t G  = 10;
const uint8_t DP = 11;

// Digit pinovi
const uint8_t DIG1 = 12;
const uint8_t DIG2 = 13;
const uint8_t DIG3 = A2;
const uint8_t DIG4 = A3;

// ---------------- MULTIPLEKS VARIJABLE ----------------

uint8_t trenutnaZnamenka = 0;
unsigned long zadnjePrebacivanje = 0;
const uint16_t MUX_PERIOD = 2000; // 2 ms po znamenki

// Broj koji želimo prikazati (1 2 3 4)
uint8_t broj[4] = {1, 2, 3, 4};
```

```
// ---------------- SETUP ----------------

void setup() {

 pinMode(A, OUTPUT);  pinMode(B, OUTPUT);  pinMode(C, OUTPUT);
 pinMode(D, OUTPUT);  pinMode(E, OUTPUT);  pinMode(F, OUTPUT);
 pinMode(G, OUTPUT);  pinMode(DP, OUTPUT);

 pinMode(DIG1, OUTPUT); pinMode(DIG2, OUTPUT);
 pinMode(DIG3, OUTPUT); pinMode(DIG4, OUTPUT);

 allSegmentsOff();
 allDigitsOff();
}

// ---------------- LOOP ----------------

void loop() {

 if (micros() - zadnjePrebacivanje >= MUX_PERIOD) {

  zadnjePrebacivanje = micros();

  allDigitsOff();   // ugasi sve znamenke
  allSegmentsOff(); // ugasi segmente

  switch (trenutnaZnamenka) {

   case 0:
    digit1();
    prikaziZnamenku(broj[0]);
    break;

   case 1:
    digit2();
    prikaziZnamenku(broj[1]);
    break;

   case 2:
    digit3();
    prikaziZnamenku(broj[2]);
    break;

   case 3:
    digit4();
    prikaziZnamenku(broj[3]);
    break;
  }

  trenutnaZnamenka++;
  if (trenutnaZnamenka > 3) {
   trenutnaZnamenka = 0;
  }
```

```
  }

  // Ovdje može ići drugi kod (npr. tipkalo, Serial itd.)
}

// ---------- POMOĆNE FUNKCIJE ----------

void allSegmentsOff() {
  digitalWrite(A, LOW); digitalWrite(B, LOW); digitalWrite(C, LOW);
  digitalWrite(D, LOW); digitalWrite(E, LOW); digitalWrite(F, LOW);
  digitalWrite(G, LOW); digitalWrite(DP, LOW);
}

void allDigitsOff() {
  digitalWrite(DIG1, HIGH);
  digitalWrite(DIG2, HIGH);
  digitalWrite(DIG3, HIGH);
  digitalWrite(DIG4, HIGH);
}

void digit1() { digitalWrite(DIG1, LOW); }
void digit2() { digitalWrite(DIG2, LOW); }
void digit3() { digitalWrite(DIG3, LOW); }
void digit4() { digitalWrite(DIG4, LOW); }

// ---------- ZNAMENKE ----------

void nula() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(E, HIGH); digitalWrite(F, HIGH);
}

void jedan() {
  digitalWrite(B, HIGH); digitalWrite(C, HIGH);
}

void dva() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH);
  digitalWrite(D, HIGH); digitalWrite(E, HIGH);
  digitalWrite(G, HIGH);
}

void tri() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH);
  digitalWrite(C, HIGH); digitalWrite(D, HIGH);
  digitalWrite(G, HIGH);
}

void cetiri() {
  digitalWrite(B, HIGH); digitalWrite(C, HIGH);
  digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}
```

```
void pet() {
  digitalWrite(A, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(F, HIGH);
  digitalWrite(G, HIGH);
}

void sest() {
  digitalWrite(A, HIGH); digitalWrite(C, HIGH);
  digitalWrite(D, HIGH); digitalWrite(E, HIGH);
  digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}

void sedam() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH);
  digitalWrite(C, HIGH);
}

void osam() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH);
  digitalWrite(C, HIGH); digitalWrite(D, HIGH);
  digitalWrite(E, HIGH); digitalWrite(F, HIGH);
  digitalWrite(G, HIGH);
}

void devet() {
  digitalWrite(A, HIGH); digitalWrite(B, HIGH);
  digitalWrite(C, HIGH); digitalWrite(D, HIGH);
  digitalWrite(F, HIGH); digitalWrite(G, HIGH);
}

void prikaziZnamenku(uint8_t x) {
  allSegmentsOff();

  switch (x) {
    case 0: nula(); break;
    case 1: jedan(); break;
    case 2: dva(); break;
    case 3: tri(); break;
    case 4: cetiri(); break;
    case 5: pet(); break;
    case 6: sest(); break;
    case 7: sedam(); break;
    case 8: osam(); break;
    case 9: devet(); break;
  }
}
```

Pojašnjenje uz gornji kod

- **Što ovaj kod radi drugačije od prethodnog?**

U prethodnom primjeru koristili smo delay() pa se program zaustavljao i znamenke su se palile "ručno" i sporo.

Ovdje radimo isto pravilo:

u jednom trenutku aktivna je samo jedna znamenka

ali razlika je u tome što:

- **ne koristimo delay()**
- prebacivanje znamenki radimo **vremenski kontrolirano** pomoću micros()

Zato prikaz može biti stabilan, a program istovremeno može čitati tipkala, Serial, senzore itd.

- **Multipleks varijable**

```
uint8_t trenutnaZnamenka = 0;
unsigned long zadnjePrebacivanje = 0;
const uint16_t MUX_PERIOD = 2000; // 2 ms po znamenki
```

- *trenutnaZnamenka* kaže koja se znamenka upravo prikazuje (0–3).
- *zadnjePrebacivanje* pamti vrijeme (u mikrosekundama) kada smo zadnji put promijenili znamenku.
- *MUX_PERIOD* je vrijeme koliko dugo jedna znamenka ostaje uključena prije prelaska na sljedeću.

Zašto micros()?

Jer mjeri vrijeme u **mikrosekundama**, što daje dovoljno finu kontrolu (npr. 2000 µs = 2 ms).

- **Polje broj[4] – što se prikazuje**

```
uint8_t broj[4] = {1, 2, 3, 4};
```

Ovo polje sadrži znamenke koje želimo vidjeti na displeju:

- broj[0] → ide na DIG1
- broj[1] → ide na DIG2
- broj[2] → ide na DIG3
- broj[3] → ide na DIG4

Kasnije ćemo baš ovo polje puniti (npr. iz unosa preko Serial Monitora ili iz izračuna).

- **setup() – priprema izvoda**

U setup() sve segmente i DIG izvode postavljamo kao OUTPUT, zatim gasimo sve:
allSegmentsOff();

allDigitsOff();

To sprečava da se pri pokretanju pojavi slučajni prikaz ili “bljesak”.

- **Ključ multipleksa je u loop() i uvjetu s vremenom**

if (micros() - zadnjePrebacivanje >= MUX_PERIOD) {

Ovaj uvjet znači:

- svaki put kada prođe **2 ms**, napravi “korak” multipleksiranja
- između tih koraka program se ne zaustavlja (nema delay())

Odmah zatim:
zadnjePrebacivanje = micros();

Postavljamo novo “zadnje vrijeme” kako bi sljedeće prebacivanje opet čekalo 2 ms.

- **Zašto se prvo gase znamenke i segmenti?**

allDigitsOff();
allSegmentsOff();

To je vrlo važno zbog dva razloga:

1. **sprječava tzv.“ghosting”** (kratko neželjeno svjetlucanje pogrešne znamenke pri promjeni segmenata)
2. osigurava da uvijek krećemo iz čistog stanja prije nego postavimo novu znamenku

Tek nakon gašenja biramo što prikazujemo.

- **switch(trenutnaZnamenka) – biramo koju znamenku palimo**

```
switch (trenutnaZnamenka) {
  case 0: digit1(); prikaziZnamenku(broj[0]); break;
  case 1: digit2(); prikaziZnamenku(broj[1]); break;
  case 2: digit3(); prikaziZnamenku(broj[2]); break;
  case 3: digit4(); prikaziZnamenku(broj[3]); break;
}
```

Ovdje se događa “srce multipleksa”:

- aktivira se samo jedna znamenka (DIG1–DIG4)
- na segmente se postavlja uzorak za odgovarajuću znamenku iz polja broj[]

Primijetimo redoslijed:

1. odaberi znamenku
2. prikaži znamenku segmentima

U praksi može raditi i obrnuto, ali ovdje je sigurno jer smo malo prije ugasili sve.

- **trenutnaZnamenka++ – kružno prebacivanje 0–3**

trenutnaZnamenka++;
if (trenutnaZnamenka > 3) trenutnaZnamenka = 0;

To znači da se prikaz vrti:

DIG1 → DIG2 → DIG3 → DIG4 → DIG1 → …

- **Koliko često se osvježava cijeli broj?**

Ako je MUX_PERIOD = 2000 µs (2 ms):

- svaka znamenka svijetli 2 ms
- 4 znamenke → 8 ms za cijeli ciklus
- 1000 ms / 8 ms ≈ 125 ciklusa u sekundi

To je dovoljno brzo da oko vidi stabilan prikaz.

- **Zašto je ovo "pravo" multipleksiranje?**

Zato što:

- nema delay()
- program radi kontinuirano
- prikaz se osvježava u pravilnim vremenskim koracima

Zato u donjem dijelu loop() možemo dodati još logike:
// tipkalo, Serial, senzori...

i prikaz će svejedno ostati stabilan.

- **Funkcije za segmente i znamenke**

(e) allDigitsOff() / digit1()...digit4()

Kod CC modula:

- DIG ON = LOW
- DIG OFF = HIGH

Zato je ovako napisano.

(f) prikaziZnamenku(x)

Ova funkcija prvo ugasi sve segmente, a onda upali potrebne segmente za znamenku 0–9.

To je i dalje “ručno” definirano (bez maski), što je prijelazni korak prije bitmaski.

- **Zadatak – Istraživanje multipleksiranja**

U programu je definirana konstanta:
const uint16_t MUX_PERIOD = 2000; // 2 ms po znamenki

Ova vrijednost određuje koliko dugo je svaka znamenka uključena prije prelaska na sljedeću.

- **Zadatak 1 – Promatranje treperenja**

Promijenite vrijednost MUX_PERIOD na sljedeće vrijednosti:

- 5000 µs
- 2000 µs
- 1000 µs
- 500 µs

Za svaku vrijednost odgovorite:

1. Je li prikaz stabilan ili primjećujete treperenje?
2. Mijenja li se subjektivna svjetlina znamenki?
3. Koja vrijednost daje najugodniji prikaz?

- **Zadatak 2 – Izračun frekvencije osvježavanja**

Ako je:
MUX_PERIOD = 2000 µs

izračunajte:

1. Koliko traje prikaz cijelog četveroznamenkastog broja?
2. Koliko puta u sekundi se cijeli broj osvježi?

Pomoć:

- 1 ms = 1000 µs
- 1 s = 1 000 000 µs

- **Zadatak 3 – Razmišljanje**

Objasnite:

1. Zašto se prije uključivanja nove znamenke poziva funkcija *allDigitsOff()*?
2. Što bi se moglo dogoditi kada bismo je izostavili?
3. Zašto je multipleksiranje učinkovitije od rješenja s delay()?

- **Dodatni izazov**

Promijenite polje:
uint8_t broj[4] = {1, 2, 3, 4};

tako da se prikaže broj:

- 2026
- 0007
- 9876

bez izmjene multipleks logike.

- **Rješenje – Zadatak 1 (opažanja za različite MUX_PERIOD)**

Vrijedi pravilo:
- **manji MUX_PERIOD → brže prebacivanje → stabilniji prikaz**
- ali **previše mali MUX_PERIOD** može smanjiti svjetlinu (svaka znamenka je uključena kraće) i povećati "posao" procesoru.

(g) *MUX_PERIOD = 5000 µs* (5 ms po znamenki)
- Cijeli ciklus: 4 × 5 ms = 20 ms → 50 Hz osvježavanje cijelog broja
- Opažanje: često se vidi **treperenje** (posebno periferijskim vidom).
- Svjetlina: relativno dobra.

(h) *MUX_PERIOD = 2000 µs* (2 ms po znamenki)
- Cijeli ciklus: 8 ms → 125 Hz
- Opažanje: uglavnom **stabilno**, treperenje minimalno ili neprimjetno.
- Svjetlina: dobra (tipična "sweet spot" vrijednost).

(i) *MUX_PERIOD = 1000 µs* (1 ms po znamenki)
- Cijeli ciklus: 4 ms → 250 Hz
- Opažanje: potpuno stabilno.
- Svjetlina: može biti mrvicu manja nego na 2 ms, ali i dalje dobra.

(j) *MUX_PERIOD = 500 µs* (0,5 ms po znamenki)
- Cijeli ciklus: 2 ms → 500 Hz
- Opažanje: stabilno.
- Svjetlina: može biti nešto manja; ako su segmenti ionako slabi, učenici to mogu primijetiti.
- Napomena: CPU "češće" ulazi u multipleks dio, pa manje vremena ostaje za ostale zadatke (Serial, tipkala…).

Zaključak (preporuka): za školsku primjenu i natjecanje najčešće su dobre vrijednosti **1000–3000 µs**, a često baš oko **2000 µs**.

- **Rješenje – Zadatak 2 (izračun)**

(k) Zadano: *MUX_PERIOD = 2000 µs*

1. **Trajanje cijelog ciklusa (4 znamenke):**
   Svaka znamenka 2000 µs, ukupno 4 znamenke:

- 2000 µs × 4 = 8000 µs

Pretvorba:

- 8000 µs = 8 ms

Cijeli četveroznamenkasti broj se "prođe" za **8 ms**.

2. **Koliko puta u sekundi se osvježi cijeli broj?**
   U jednoj sekundi ima 1 000 000 µs.

- 1 000 000 µs / 8000 µs = 125

Osvježavanje cijelog broja je **125 puta u sekundi (125 Hz)**.

- **Rješenje – Zadatak 3 (objašnjenja)**

1. **Zašto *allDigitsOff()* prije nove znamenke?**
   Da se najprije ugase sve znamenke prije promjene segmenata. Tako se sprječava da u trenutku promjene segmenata "na pogrešnoj znamenki" nakratko zasvijetli nešto što ne želimo.

2. **Što se može dogoditi ako se izostavi *allDigitsOff()***

   Može se pojaviti tzv. **ghosting**:
   - kratko neželjeno "preslikavanje" segmenata na susjednoj znamenki
   - zamućen prikaz (posebno kod brzih promjena)

3. **Zašto je multipleks bolje od *delay()* rješenja?**

   Jer *delay()* zaustavlja program. Kod multipleksa s *micros():*
   - prikaz radi stabilno,
   - a mikroupravljač može paralelno raditi druge zadatke (tipkala, Serial, senzori…).

- **Rješenje – Dodatni izazov (promjena polja broj[4])**

U originalu:

uint8_t broj[4] = {1, 2, 3, 4};
(l) Prikaži 2026
uint8_t broj[4] = {2, 0, 2, 6};
(m) Prikaži 0007
uint8_t broj[4] = {0, 0, 0, 7};
(n) Prikaži 9876
uint8_t broj[4] = {9, 8, 7, 6};

4. Prijelaz na rad s poljima i maskama

U dosadašnjim primjerima svaki broj (0–9) prikazivali smo pomoću zasebne funkcije u kojoj smo ručno određivali koji segment treba biti uključen, a koji isključen. Takav pristup je dobar za razumijevanje rada pojedinih segmenata, ali ima nekoliko nedostataka:

- program postaje dug i nepregledan
- ponavlja se mnogo sličnog koda
- teško je brzo promijeniti ili generirati prikaz broja

Uočimo da svaka znamenka zapravo predstavlja **točno određenu kombinaciju segmenata**. Ta kombinacija se može zapisati u obliku binarnog broja, gdje svaki bit predstavlja stanje jednog segmenta (uključen ili isključen).

Na taj način znamenke više ne opisujemo funkcijama, nego **podatkom** – maskom segmenata.

Time rad s displejem prelazi iz domene „upravljanja pinovima“ u domenu **obrade podataka**. Ovakav pristup identičan je načinu na koji rade profesionalne biblioteke i komercijalni upravljački sklopovi za displeje.

Multipleks dio programa ostaje isti – on samo prikazuje ono što se nalazi u polju podataka.

Ono što se mijenja jest način na koji pripremamo podatke za prikaz.
U sljedećem primjeru svaku znamenku opisat ćemo 8-bitnom maskom, a četveroznamenkasti broj spremati u polje frame[4].

4.1 Segmenti kao bitovi

Dogovorimo redoslijed bitova u jednom bajtu (8 bitova):

Bit	Segment
bit 0	A
bit 1	B
bit 2	C
bit 3	D
bit 4	E
bit 5	F
bit 6	G
bit 7	DP

Dakle, jedan uint8_t (8 bitova) može opisati stanje svih segmenata jedne znamenke.

- **Primjer: znamenka 2**

Znamo da za broj **2** moraju svijetliti segmenti:

A, B, D, E i G

To znači:

Segment	Uključen?	Bit
A	1	bit0
B	1	bit1
C	0	bit2
D	1	bit3
E	1	bit4
F	0	bit5
G	1	bit6
DP	0	bit7

Zapis u binarnom obliku (od bit7 do bit0):
0b01011011

Ili zapisano u programu:
0b01011011 // znamenka 2

- **Primjer: znamenka 1**

Svijetle samo B i C:

Segment	Bit
B	bit1
C	bit2

Zapis:
0b00000110

Što ovime dobivamo?

Umjesto funkcije:

```
void dva() {
  digitalWrite(A, HIGH);
  digitalWrite(B, HIGH);
  ...
}
```

sada imamo samo podatak:
MASK[2]

I to je sve.

Prikaz znamenke više nije niz naredbi, nego **jedan bajt podataka**.

U nastavku ćemo znamenke opisivati podacima (maskama), a multipleks dio će samo prikazivati sadržaj polja frame[4].

- **Sljedeći korak**

Sada:

1. Napravimo polje maski za znamenke 0–9.
2. Napravimo polje *frame[4]* koje čuva maske za četiri znamenke.
3. Multipleks dio samo čita *frame[muxIndex]* i prikazuje ga.

Time:

- multipleks dio postaje univerzalan,
- promjena broja znači samo promjenu podataka.

Idemo redom. Prvi korak je tablica maski za 0–9, pa funkcija koja masku "izlije" na pinove.

4.2 Tablica maski MASK[10]

Dogovor: **bit0=A, bit1=B, bit2=C, bit3=D, bit4=E, bit5=F, bit6=G, bit7=DP**.

Napomena: bit0 je "najniži bit" (LSB), bit7 je "najviši bit" (MSB).

```
// Maske segmenata za znamenke 0–9
// bit0=A, bit1=B, bit2=C, bit3=D, bit4=E, bit5=F, bit6=G, bit7=DP
const uint8_t MASK[10] = {
  0b00111111, // 0: A B C D E F
  0b00000110, // 1: B C
  0b01011011, // 2: A B D E G
  0b01001111, // 3: A B C D G
  0b01100110, // 4: B C F G
  0b01101101, // 5: A C D F G
  0b01111101, // 6: A C D E F G
  0b00000111, // 7: A B C
  0b01111111, // 8: A B C D E F G
  0b01101111  // 9: A B C D F G
};
```

Ove maske vrijede za **COMMON CATHODE (CC)** gdje je **segment ON = HIGH**.

4.3 Funkcija koja postavlja segmente prema maski

Da ne pišemo 8 *digitalWrite()* svaki put, napravimo polje pinova i petlju:

```
const uint8_t SEG_PINS[8] = {A, B, C, D, E, F, G, DP};

void setSegmentsFromMask(uint8_t mask) {
  for (uint8_t i = 0; i < 8; i++) {
    bool on = (mask >> i) & 0x01;        // je li bit i = 1?
    digitalWrite(SEG_PINS[i], on ? HIGH : LOW); // CC: ON=HIGH
  }
}
```

4.4 frame[4] – 4 maske za 4 znamenke

frame[i] neće više sadržavati brojčanu znamenku, nego **masku segmenata** za tu poziciju.

volatile uint8_t frame[4] = {0, 0, 0, 0}; // maske za DIG1..DIG4

- *frame[0] → prikaz na DIG1 (tisućice)*
- *frame[1] → prikaz na DIG2 (stotice)*
- *frame[2] → prikaz na DIG3 (desetice)*
- *frame[3] → prikaz na DIG4 (jedinice)*

4.5 Funkcija postaviBroj() – pretvara broj u maske

Za raspon 0–9999:

```
void postaviBroj(uint16_t vrijednost) {
 frame[0] = MASK[(vrijednost / 1000) % 10];
 frame[1] = MASK[(vrijednost / 100)  % 10];
 frame[2] = MASK[(vrijednost / 10)   % 10];
 frame[3] = MASK[vrijednost % 10];
}
```

(o) Primjer

Ako pozovemo:
postaviBroj(2026);

dobijemo:

- *frame[0] = MASK[2]*
- *frame[1] = MASK[0]*
- *frame[2] = MASK[2]*
- *frame[3] = MASK[6]*

4.6 (Opcionalno) gašenje vodećih nula

Ako želimo da se npr. 26 prikaže kao “ 26” umjesto “0026”, možemo “praznu” znamenku prikazati maskom 0 (ništa ne svijetli):

```
void postaviBrojBezVodecihNula(uint16_t vrijednost) {
 uint8_t t = (vrijednost / 1000) % 10;
 uint8_t s = (vrijednost / 100)  % 10;
 uint8_t d = (vrijednost / 10)   % 10;
 uint8_t j = vrijednost % 10;

 frame[0] = (vrijednost >= 1000) ? MASK[t] : 0;
 frame[1] = (vrijednost >= 100)  ? MASK[s] : 0;
 frame[2] = (vrijednost >= 10)   ? MASK[d] : 0;
```

```
  frame[3] = MASK[j]; // jedinice uvijek prikazujemo
}
```

Sljedeći korak je da multipleks dio umjesto *prikaziZnamenku(broj[i])* koristi: *setSegmentsFromMask(frame[muxIndex]);*

i time cijeli prikaz postaje "data-driven".

Sad spajamo sve: multipleks + frame[4] + maske.

4.7 Multipleks dio sada prikazuje *frame[muxIndex]*

Umjesto:

- *prikaziZnamenku(broj[i])*

radimo:

- setSegmentsFromMask(frame[muxIndex])

I ne treba nam više *nula()...devet()* ni *prikaziZnamenku().*

- **Cijeli primjer (CC): multipleks + maske + frame**

```
// 4-digit 7-seg – multipleks s maskama i frame[4]
// COMMON CATHODE: segment ON = HIGH, digit ON = LOW

#include <Arduino.h>

// ---------------- PINOVI ----------------

// Segment pinovi
const uint8_t A  = 4;
const uint8_t B  = 5;
const uint8_t C  = 6;
const uint8_t D  = 7;
const uint8_t E  = 8;
const uint8_t F  = 9;
const uint8_t G  = 10;
const uint8_t DP = 11;

// Digit pinovi
const uint8_t DIG1 = 12;
const uint8_t DIG2 = 13;
const uint8_t DIG3 = A2;
const uint8_t DIG4 = A3;

// Polja pinova (radi petlji)
```

```
const uint8_t SEG_PINS[8] = {A, B, C, D, E, F, G, DP};     // bit0..bit7
const uint8_t DIG_PINS[4] = {DIG1, DIG2, DIG3, DIG4};

// ---------------- MASKE ----------------
// bit0=A, bit1=B, bit2=C, bit3=D, bit4=E, bit5=F, bit6=G, bit7=DP

const uint8_t MASK[10] = {
  0b00111111, // 0
  0b00000110, // 1
  0b01011011, // 2
  0b01001111, // 3
  0b01100110, // 4
  0b01101101, // 5
  0b01111101, // 6
  0b00000111, // 7
  0b01111111, // 8
  0b01101111  // 9
};

// ---------------- MULTIPLEKS VARIJABLE ----------------

volatile uint8_t frame[4] = {0, 0, 0, 0};  // maske za DIG1..DIG4
volatile uint8_t muxIndex = 0;

unsigned long lastMux = 0;
const uint16_t MUX_PERIOD = 2000; // mikrosekunde (2 ms po znamenki)

// ---------------- FUNKCIJE ----------------

void allDigitsOff() {
  for (uint8_t i = 0; i < 4; i++) {
    digitalWrite(DIG_PINS[i], HIGH); // CC: OFF=HIGH
  }
}

void setSegmentsFromMask(uint8_t mask) {
  for (uint8_t i = 0; i < 8; i++) {
    bool on = (mask >> i) & 0x01;
    digitalWrite(SEG_PINS[i], on ? HIGH : LOW); // CC: ON=HIGH
  }
}

void postaviBroj(uint16_t vrijednost) {
  frame[0] = MASK[(vrijednost / 1000) % 10];
  frame[1] = MASK[(vrijednost / 100)  % 10];
  frame[2] = MASK[(vrijednost / 10)   % 10];
  frame[3] = MASK[vrijednost % 10];
}

// ---------------- SETUP ----------------

void setup() {
  for (uint8_t i = 0; i < 8; i++) pinMode(SEG_PINS[i], OUTPUT);
```

```
  for (uint8_t i = 0; i < 4; i++) pinMode(DIG_PINS[i], OUTPUT);

  allDigitsOff();
  setSegmentsFromMask(0);

  postaviBroj(2026); // probaj: 7, 26, 1234, 9999...
}

// ---------------- LOOP ----------------

void loop() {
  unsigned long now = micros();

  if (now - lastMux >= MUX_PERIOD) {
    lastMux = now;

    allDigitsOff();                  // 1) ugasi sve znamenke
    setSegmentsFromMask(frame[muxIndex]); // 2) postavi segmente za tu znamenku
    digitalWrite(DIG_PINS[muxIndex], LOW);// 3) upali tu znamenku (CC)

    muxIndex++;
    if (muxIndex >= 4) muxIndex = 0;
  }

  // Ovdje kasnije: Serial unos, tipkala, logika...
}
```

- **Mini napomena (DP)**

Ako želimo DP na npr. drugoj znamenki:
frame[1] |= 0b10000000; // bit7 = DP

Ugasiti DP:
frame[1] &= 0b01111111;

- **3 zadatka za maske**

Zadatak 6.1 – Provjera maski (osnovno)

U *setup()* postavi da se prikaže broj **2026**.

Uputa: koristi funkciju *postaviBroj(...).*

Rješenje:
postaviBroj(2026);

Zadatak 6.2 – Decimalna točka (DP)

Uključi decimalnu točku na **drugoj znamenki** (DIG2) tako da se prikaže npr. **20.26** (dp iza druge znamenke).

Napomena: DP je bit7.

Rješenje:
frame[1] |= 0b10000000; // upali DP na DIG2
// ili: frame[1] |= (1 << 7);

Za gašenje:
frame[1] &= 0b01111111; // ugasi DP
// ili: frame[1] &= ~(1 << 7);

Zadatak 6.3 – "Prazna" znamenka (bez vodećih nula)

Prikaži broj **26** tako da se ne prikazuju vodeće nule, nego da lijeve dvije znamenke budu prazne:

"_ _ 2 6"

Uputa: prazna znamenka je maska 0.

Rješenje– najjednostavnije ručno:
frame[0] = 0;
frame[1] = 0;
frame[2] = MASK[2];
frame[3] = MASK[6];

5. Upotreba podataka s analognih ulaza

Do sada smo učili kako:

- **upravljati segmentima jedne znamenke,**
- **riješiti problem zajedničkih vodova pomoću multipleksiranja,**
- **organizirati prikaz pomoću polja frame[4] i tablice maski MASK[10].**
-

U svim tim primjerima broj koji se prikazivao bio je unaprijed zadan u programu.

Sada prelazimo na sljedeći korak: broj više ne dolazi iz programa, nego iz stvarnog svijeta.

U mnogim situacijama koristimo senzore spojene na analogne ulaze mikroupravljačkog sučelja. Ti senzori mjere neku fizičku veličinu (npr. temperaturu, svjetlost, udaljenost, tlak…) i pretvaraju je u električni signal. Mikroupravljač taj signal ne može izravno obrađivati jer radi s digitalnim vrijednostima.
Analogni ulaz ima **AD pretvarač (analogno-digitalni pretvarač, AD konverter)** koji pretvara analogni napon u digitalnu brojčanu vrijednost.
Kod većine pločica (npr. Arduino Nano) AD konverter ima rezoluciju 10 bita, što znači da analogni ulaz može poprimiti vrijednosti u rasponu:

Mikroupravljač signal s nekog analognog ulaza očitava naredbom:

analogRead(A0);

Rezultat je (u slučaju 10-bitnog AD pretvarača) broj u rasponu 0–1023.

Napon na ulazu može varirati od 0 do 5 V (ili 3,3, **ovisno u mikroupravljačkom sučelju**).
Maksimalni napon koji se može pojaviti na ulazu mikroupravljačkog sučelja je 5V, minimalni 0 V, a vrijednosti koje AD pretvarač daje su u rasponu od 0 do 1023.
Postavlja se pitanje kako se taj napon pretvara u određenu vrijednost.

AD konverter kontinuirani analogni signal pretvara u diskretne numeričke vrijednosti. To znači da svakom rasponu napona odgovara određeni broj između 0 i 1023.
U praksi, se međutim, u većini zadataka ne bavimo izravno naponom, nego koristimo dobivenu brojčanu vrijednost kao podatak.
Kod složenijih senzora često koristimo gotove biblioteke koje već sadrže potrebne funkcije za pretvorbu napona u fizičku veličinu (npr. temperaturu u °C ili udaljenost u cm).

Za nas je u ovom trenutku najvažnije sljedeće:

Analogni ulaz daje broj.
Taj broj možemo matematički obraditi i dalje koristiti.

Najjednostavniji primjer je prikaz podataka sa senzora na serijskom monitoru:

Ako je na analogni ulaz A0 spojen potenciometar, promjenom položaja klizača mijenjamo napon na ulazu, a time i digitalnu vrijednost koju očitava mikroupravljač.

Primjer programa:

```
void setup() {
  Serial.begin(9600);
```

```
}

void loop() {
  int vrijednost = analogRead(A0);
  Serial.println(vrijednost);
  delay(200);
}
```

Što se događa?

analogRead(A0) očitava vrijednost u rasponu 0–1023
Serial.println() tu vrijednost ispisuje na serijski monitor
delay(200) usporava ispis da bude čitljiv

Ako okrećemo potenciometar:

- u jednom krajnjem položaju vidjet ćemo vrijednost blizu 0
- u drugom krajnjem položaju vidjet ćemo vrijednost blizu 1023
- između toga dobivamo sve međuvrijednosti

Time dobivamo numerički podatak koji možemo dalje koristiti u programu.

Taj broj možemo prikazati i na 7-segmentnom LED displeju, ali to nije baš "jedna naredba". Recimo da je vrijednost koju želimo prikazati **1002**. Ne možemo samo u programu napisati nešto tipa: *prikaži 1002 na displeju*.

Razlog je jednostavan: 4-znamenkasti displej ne zna sam "rastaviti" broj na znamenke. On samo prikazuje **ono što mu mi pošaljemo** za svaku poziciju (DIG1–DIG4). Zato moramo:

1. **rastaviti broj na četiri znamenke**
2. **svaku znamenku poslati na svoju poziciju** (DIG1–DIG4)

Dva pristupa: broj ili tekst (String)

Postoje dva česta pristupa, ovisno o tome u kojem obliku dolazi podatak:

(p) 1) Ako imamo broj (npr. uint16_t vrijednost = 1002;)

Tada je najprirodnije koristiti matematiku (/ i %) i izvući znamenke:

- tisućice, stotice, desetice, jedinice

Ovo je brz i pouzdan pristup, posebno za mikroupravljače.

(q) 2) Ako imamo tekst (String) (npr. "1002")

Tada možemo raditi s pojedinim znakovima u stringu.

Kod Stringa su dostupne funkcije koje mogu izdvojiti dio teksta, npr.:

- **uzeti znak s određenog mjesta**
- **izvući dio teksta** (podstring)
- **uzeti nekoliko znakova od početka ili kraja**

Ali i ovdje se vraćamo na istu ideju:
moramo dobiti **četiri znamenke** (kao znakove ili brojeve), pa ih zatim prikazati jednu po jednu na displeju.

U nastavku ćemo rastavljati broj matematički, a zatim (po potrebi) pokazati i verziju kada znamenke dolaze kao tekst iz Serial Monitora.

5.1 Rastavljanje četveroznamenkastog broja na četiri znamenke

U ovom poglavlju prvo ćemo pokazati kako **rastaviti broj matematički**, a zatim (po potrebi) i verziju kada znamenke dolaze kao **tekst (String)**, npr. iz Serial Monitora.
Cilj je uvijek isti: dobiti četiri znamenke i smjestiti ih na pozicije displeja:

- **DIG1** → tisućice
- **DIG2** → stotice
- **DIG3** → desetice
- **DIG4** → jedinice

A) Matematičko rastavljanje broja (preporučeno)

Ako je broj u varijabli tipa uint16_t, znamenke dobijemo dijeljenjem i ostatkom pri dijeljenju:

- tisućice: broj / 1000
- stotice: (broj / 100) % 10
- desetice: (broj / 10) % 10
- jedinice: broj % 10

Primjer:

```
uint16_t vrijednost = 1002;

uint8_t t = (vrijednost / 1000) % 10;  // 1
uint8_t s = (vrijednost / 100)  % 10;  // 0
uint8_t d = (vrijednost / 10)   % 10;  // 0
uint8_t j =  vrijednost % 10;          // 2
```

Ovaj način je brz, jednostavan i najčešće najbolji za mikroupravljače.

B) String pristup (kad znamenke dolaze kao tekst)

Ako broj dolazi kao tekst, npr. "1002", tada možemo uzeti znakove s određenih pozicija.

Primjer:

```
String s = "1002";
```

```
char c0 = s.charAt(0);  // '1'
char c1 = s.charAt(1);  // '0'
char c2 = s.charAt(2);  // '0'
char c3 = s.charAt(3);  // '2'

// pretvori znak u broj: '0' -> 0, '7' -> 7 itd.
uint8_t z0 = c0 - '0';
uint8_t z1 = c1 - '0';
uint8_t z2 = c2 - '0';
uint8_t z3 = c3 - '0';
```

Ovaj pristup je koristan kad čitamo unos iz Serial Monitora, ali i dalje na kraju moramo doći do znamenki 0–9.

5.2 Povezivanje s prikazom na displeju (frame + MASK)

Kad imamo znamenke (0–9), na 7-seg displeju ih najčešće ne prikazujemo direktno kao broj, nego kao **masku segmenata**.

Ako već koristimo:

- *MASK[10] (tablica segmenata za 0–9)*
- *frame[4] (što se prikazuje na DIG1..DIG4)*

tada je prikaz broja samo "punjenje" *frame[].*

Primjer za broj 1002:

```
void postaviBroj(uint16_t vrijednost) {
 frame[0] = MASK[(vrijednost / 1000) % 10];  // DIG1
 frame[1] = MASK[(vrijednost / 100)  % 10];  // DIG2
 frame[2] = MASK[(vrijednost / 10)   % 10];  // DIG3
 frame[3] = MASK[vrijednost % 10];           // DIG4
}
```

Nakon toga multipleks dio samo prikazuje *frame[muxIndex]* i broj se pojavljuje na displeju.

Važno je uočiti sljedeće:

- displej **ne zna** što je broj 1002
- displej zna samo upaliti određene segmente
- zato broj uvijek moramo rastaviti na znamenke prije prikaza

Drugim riječima, mikroupravljač mora:

1. primiti podatak (npr. s analognog ulaza),
2. pretvoriti ga u broj,
3. rastaviti broj na znamenke,
4. pripremiti podatke za prikaz,

5. a multipleks dio samo ih brzo izmjenjuje na displeju.

- **Zadatak 1**

Ako analogni ulaz daje vrijednosti u rasponu 0–1023:

- Hoće li sve četiri znamenke uvijek biti iskorištene?
- Kada će lijeve znamenke biti nule?
- Trebaju li se vodeće nule prikazivati ili skrivati?

- **Zadatak 2**

Ako se broj mijenja vrlo brzo (npr. brzim okretanjem potenciometra):

- Hoće li displej uvijek prikazivati stabilan broj?
- Što određuje brzinu osvježavanja prikaza?
- Može li multipleks utjecati na čitljivost?

- **Zadatak 3**

Ako želimo prikazati broj 7:

- Treba li to biti 0007 ili samo 7?
- Kako bismo “ugasili” prve tri znamenke?
- Koja je maska za “praznu” znamenku?

- **Zadatak 4**

Ako broj prelazi iz 999 u 1000:

- Što se događa s tisućicama?
- Koji matematički dio izraza se tada mijenja?
- Zašto je važno koristiti `% 10` nakon dijeljenja?

- **Zadatak 5**

Ako analogni ulaz daje maksimalno 1023, ali displej može prikazati do 9999:

- Trebamo li podatak skalirati?
- Je li prikaz 0–1023 dovoljan?
- U kojim situacijama bismo željeli drugačiji raspon?

- **Zadatak 6**

Ako bismo htjeli prikazati decimalnu točku (npr. 10.23):

- Koju znamenku trebamo “označiti” DP bitom?
- Kako to utječe na masku segmenata?
- Mora li se multipleks logika mijenjati?

5.3 Učitavanje broja sa Serial Monitora

Umjesto da broj upisujemo ručno u program, možemo ga unijeti putem **Serial Monitora**. Primjer: korisnik upiše broj 2026 i pritisne Enter, a taj broj se prikaže na displeju.

Primjer programa (dio vezan uz unos)

Pretpostavimo da već imamo:

- multipleks dio
- postaviBroj(uint16_t vrijednost)

Dodajemo samo serijsku komunikaciju.

```
uint16_t uneseniBroj = 0;

void setup() {
        Serial.begin(9600);

        // ostali setup (pinMode, multipleks itd.)
}
```

U loop() dodamo:

```
void loop() {
    // --- PROVJERA JE LI NEŠTO UNESENO ---
    if (Serial.available() > 0) {
        uneseniBroj = Serial.parseInt();  // čita cijeli broj
        // ograničenje na 0–9999
        if (uneseniBroj > 9999) {
                uneseniBroj = 9999;
        }
        postaviBroj(uneseniBroj);
        Serial.print("Prikazujem: ");
        Serial.println(uneseniBroj);
    }
    // multipleks dio i dalje radi
}
```

Kako ovo radi?

- *Serial.available()* provjerava postoji li uneseni podatak.
- *Serial.parseInt()* pročita cijeli broj iz unosa. Preskače sve što nije broj i može kratko "čekati" ako nema završetka unosa — pa je normalno da ponekad djeluje kao da kasni.
- Broj spremamo u *uneseniBroj.*
- Pozivamo *postaviBroj()* koja puni *frame[4*].
- Multipleks dio samo prikazuje sadržaj *frame[].*

Kako testirati?

U Serial Monitoru odaberi 'Newline' ili 'Both NL & CR' kako bi se poslao znak za kraj unosa.

1. Otvori Serial Monitor.
2. Postavi brzinu na **9600 baud**.
3. Upiši npr.: „1234“
4. Pritisni Enter.

Broj se odmah prikaže na displeju.

Važna napomena

Ako korisnik upiše:

- 7 → prikazat će se 0007 (osim ako ne koristimo verziju bez vodećih nula)
- 12345 → ograničit će se na 9999 (zbog zaštite)

6. Završne napomene

- **Natjecateljski mindset (kratki motivacijski odlomak)**

Na natjecanju nije presudno samo "znati kod", nego razumjeti sustav. Natjecatelj koji razumije:

- zašto se koristi multipleksiranje,
- što znači frekvencija osvježavanja,
- kako su segmenti organizirani u maske,
- kako se broj rastavlja na znamenke,

moći će prilagoditi rješenje i u situaciji kada zadatak nije identičan onome iz pripreme. Zadatak se može promijeniti — ali princip ostaje isti. Cilj pripreme nije naučiti rješenje napamet, nego razumjeti strukturu problema.

- **Napomena za mentore**

Preporučuje se da učenici:

- samostalno eksperimentiraju s vrijednošću MUX_PERIOD,
- promatraju pojavu treperenja i ghostinga,
- mijenjaju redoslijed aktivacije znamenki,
- dodaju jednostavnu logiku (tipkalo, brojač, promjenu broja).

Važno je poticati razumijevanje, a ne memoriranje gotovog koda.

Skripta je namjerno strukturirana tako da učenik prvo “vidi problem”, zatim ga razumije, a tek onda implementira rješenje. Takav pristup razvija sigurnost i samostalnost u radu.

- **Tehnički sažetak**

U skripti su korišteni sljedeći koncepti:

- digitalni izlazi mikroupravljača
- zajednička katoda (COMMON CATHODE) logika upravljanja
- vremensko upravljanje pomoću funkcije micros()
- rad s poljima (uint8_t frame[4])
- tablica maski segmenata (MASK[10])
- bitovne operacije (shift, AND, OR)
- analogno-digitalna pretvorba (analogRead)
- serijska komunikacija (Serial)

Ovakav skup koncepata predstavlja tipičnu osnovu za natjecateljske zadatke iz automatike i mikroupravljačkih sustava.

7. Ishodi učenja (kompetencijski okvir)

Nakon rada s ovim materijalom učenik će moći:

1. Objasniti razliku između displeja sa zajedničkom katodom (CC) i zajedničkom anodom (CA).
2. Analizirati problem zajedničkih vodova kod višeznamenkastog displeja.
3. Objasniti princip multipleksiranja i njegovu svrhu.
4. Implementirati vremenski kontrolirano multipleksiranje bez upotrebe funkcije *delay().*
5. Organizirati prikaz pomoću polja podataka (*frame[4]*).
6. Koristiti tablicu maski segmenata (*MASK[10]*) za generiranje prikaza.
7. Rastaviti višeznamenkasti broj na pojedine znamenke matematičkim postupkom.
8. Povezati podatke dobivene s analognog ulaza s prikazom na displeju.
9. Analizirati utjecaj frekvencije osvježavanja na stabilnost prikaza.
10. Samostalno prilagoditi programsko rješenje novom natjecateljskom zadatku.

8. Kompetencije

Radom na ovim zadacima učenici razvijaju:

- logičko i algoritamsko razmišljanje
- sposobnost analize tehničkog sustava
- razumijevanje vremenskog upravljanja u mikroupravljačkim sustavima
- preciznost u radu s digitalnim i analognim signalima
- sposobnost organizacije podataka
- sposobnost optimizacije programskog rješenja

Ove kompetencije čine temelj za uspješno sudjelovanje na natjecanjima iz područja automatike i tehničke kulture.

9. Zaključak

U ovoj skripti prošli smo cjelovit put od osnovnog razumijevanja rada jednog 7-segmentnog LED displeja do implementacije pravog multipleksiranja i organizacije prikaza pomoću podataka i maski.

Učenik je postupno:

- razumio da je znamenka kombinacija segmenata,
- uočio problem zajedničkih vodova kod višeznamenkastog displeja,
- savladao princip upravljanja jednom znamenkom u jednom trenutku,
- implementirao vremenski kontrolirano multipleksiranje bez korištenja delay(),
- prešao s ručnog upravljanja pinovima na podatkovni pristup (maske i polja),
- povezao prikaz s podacima dobivenim iz stvarnog sustava (analogni ulaz i serijska komunikacija).

Takav način rada razvija:

- analitičko razmišljanje,
- razumijevanje uzročno-posljedičnih veza,
- sposobnost organizacije programa,
- preciznost u radu s mikroupravljačkim sustavima.

Upravo takva znanja i način razmišljanja čine temelj uspješnog rješavanja natjecateljskih zadataka iz područja automatike.

10. Namjena dokumenta

Ovaj dokument izrađen je kao radni materijal za pripremu učenika za županijsko natjecanje mladih tehničara iz područja automatike.

Materijal je namijenjen:

- učenicima viših razreda osnovne škole,
- mentorima tehničke kulture,
- voditeljima izvannastavnih aktivnosti iz područja automatike i mikroupravljačkih sustava.

Dokument se može koristiti kao:

- pripremni materijal za natjecanje,
- dodatni obrazovni sadržaj,
- temelj za daljnju nadogradnju projekata iz područja automatike.