Arduino programsko okruženje

Arduino je zajednički naziv za familiju mikrokontrolerskih platformi, namenjenih razvoju aplikacija kod kojih je potrebno na jednostavan način ostvariti spregu između hardverskih i softverskih komponenti. Hardver se sastoji od štampane ploče zasnovane na 8-bitnom Atmel AVR, ili 32-bitnom Atmel ARM mikrokontroleru. Razvoj softvera je omogućen korišćenjem *Open-source* integrisanog razvojnog okruženja. Veza sa hardverom ostvarena je preko serijskog USB interfejsa koji se nalazi na samoj ploči i preko kojeg se program koji je napisan i kompajliran u razvojnom okruženju prebacuje u programsku FLASH memoriju mikrokontrolera. U ovom dokumentu biće opisani osnovni elementi Arduino programskog okruženja, zajedno sa opisom sintakse i semantike jezika koji se koristi za pisanje programa. Takođe, biće opisane i funkcije koje se nalaze u okviru standardnih ugrađenih biblioteka i koje olakšavaju programiranje i skraćuju vreme potrebno za razvoj softverskih aplikacija.

Arduino UNO

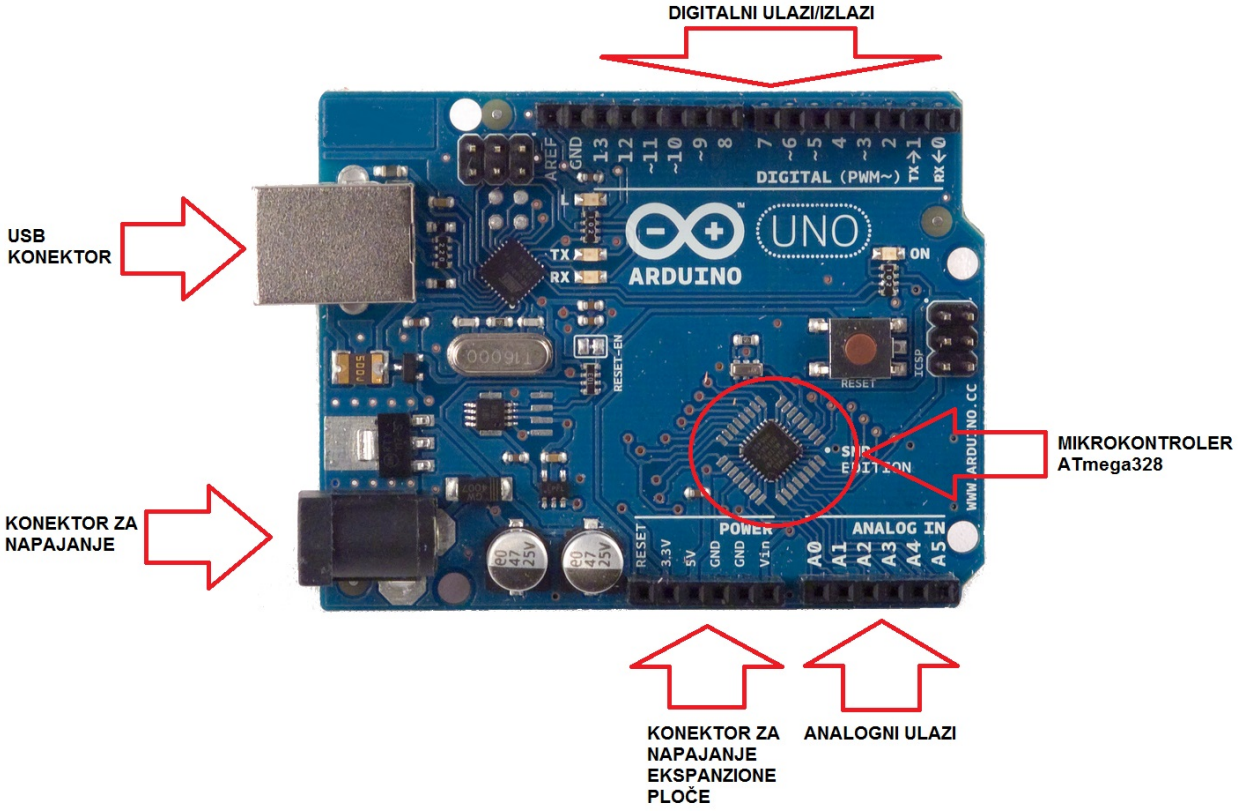
Model Arduino UNO sadrži 8-bitni ATMega328 mikrokontroler. Aktuelna verzija je opremljena USB interfejsom, zajedno sa 6 analognih ulaza i 14 digitalnih ulaza/izlaza opšte namene[1]. Analogni i digitalni ulazno/izlazni priključci su izvedeni na konektore koji su smešteni na obodima ploče i preko kojih je moguće spajanje osnovne kontrolerske ploče sa različitim ekspanzionim pločama, koje se nazivaju *Štitovi* (engl. *Shield*). Dakle, štitovi su ploče na kojima se mogu nalaziti hardverski uređaji različite namene, npr. tasteri, displeji, moduli za kontrolu različitih vrsta pogonskih uređaja (aktuatora), senzora, moduli za beičnu komunikaciju, itd.

Osnovna ploča Arduino UNO sistema prikazana je na slici 1. Na ploči se nalazi mikrokontroler, zajedno sa neophodnim komponentama koje omogućavaju njegov rad, kao i povezivanje sa računarom preko kog se vrši programiranje i sa drugim periferijskim pločama. Povezivanje sa računarom vrši se preko serijskog USB interfejsa koji ima trojaku ulogu:

- Kada je potrebno reprogramirati kontroler, tj. upisati novi program u njegovu Flash memoriju, aktivira se pomoćni program koji je prisutan u njegovoj programskoj memoriji i naziva se *Bootloader*. Uloga ovog programa je prijem novog programskog koda preko serijske veze i njegovo smeštanje u programsku memoriju kontrolera.

[1] Ulazno-izlazni priključci mikrokontrolera se nazivaju *pinovi*. Pinovi su obično grupisani u logičke celine koje se nazivaju *portovi*

- Prilikom uobičajenog režima rada u kojem kontroler izvršava program koji je već smešten u Flash memoriju, serijski interfejs se može koristiti za razmenu podataka između aplikacija koja se izvršavaju na Arduinu i na računaru.
- Arduino ploča obično prima napon napajanja od 5V sa USB linije. Ukoliko je aplikacija takva da se od Arduina zahteva da radi samostalno, bez povezivanja sa računarom (engl. *Stand-alone application*), napajanje se može dovesti sa posebnog konektora, ili sa ekspanzione ploče.



Slika 1: *Izgled i raspored konektora osnovne ploče Arduino UNO sistema*

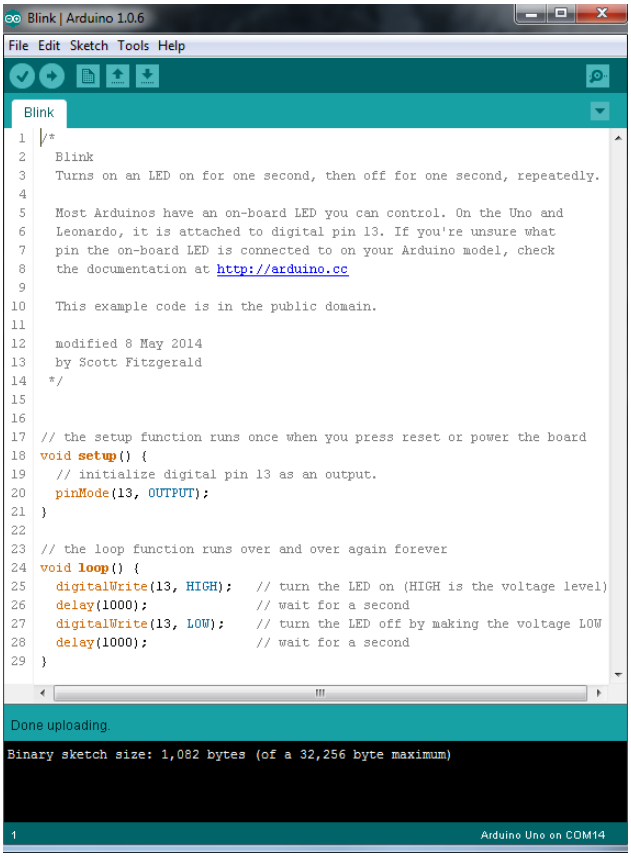
Arduino programsko razvojno okruženje

Arduino IDE (engl. Integrated Development Environment) je okruženje koje se koristi za razvoj softvera za različite tipove Arduino platformi. Pisano je u programskom jeziku Java i postoji u varijantama za operativne sisteme Windows, Linux i Mac OS X. Moguće ga je besplatno preuzeti preko web stranice http://www.arduino.cc/.

Izgled prozora Arduino IDE razvojnog okruženja prikazan je na slici 2. U okruženje između ostalog spadaju sledeći programski moduli:

- **Okruženje za unos teksta (editor)**, u kojem se piše izvorni kod programa
- **Prevodilac (kompajler)**, koji prevodi izvorni kod programa u izvršni mašinski kod

- **Programator**, koji se koristi za komunikaciju sa bootloaderom na ploči i služi za prebacivanje kompajliranog programa u programsku flash memoriju mikrokontrolera
- **Serijski monitor**, koji služi za komunikaciju sa Arduinom posredstvom virtuelnog USB serijskog porta, za vreme izvršavanja aplikacije



Slika 2: *Izgled prozora Arduino IDE razvojnog okruženja*

Programski jezik Arduino aplikacija koristi osnovne elemente jezika C i C++. Pisanje programa koji će se izvršavati na Arduino ploči podrazumeva da korisnik mora da definiše dve funkcije, koje sačinjavaju izvršni program. Te funkcije su:

- **setup()** - funkcija koja se izvršava jednom na početku i služi za početna podešavanja (inicijalizaciju parametara)
- **loop()** - funkcija koja se nakon inicijalizacije izvršava u beskonačnoj petlji i poziva se sve dok se ne isključi napajanje osnovne ploče, ili dok ne nastupi reset, koji dovodi do ponovnog izvršenja programa iz početka

Budući da se koncept Arduina zasniva na otvorenom kodu (engl. *open-source*), kao i zahvaljujući činjenici da su razvojni alati intuitivni i jednostavni za upotrebu, ova platforma je vrlo brzo stekla široku popularnost i veliki broj sledbenika širom sveta. Vremenom, korisnici su razvili veliki broj primera programa, kao i biblioteka korisnih funkcija, koje su stavili na

raspolaganje drugim korisnicima. Takođe, uobičajena je praksa samostalnog razvoja ekspanzionih ploča (štitova) od strane samih korisnika, kao i deljenja tehničke dokumentacije vezane za njihov dizajn. Profesionalci i entuzijasti koji su uključeni u razvoj harvera i softvera na internetu sačinjavaju vrlo živu i aktivnu zajednicu koja međusobno razmenjuje znanja i iskustva, kako putem zvanične Arduino stranice, tako i na brojnim web portalima i forumima.

Kreatori Arduino platforme su programske aplikacije nazvali *skicama* (engl. *sketch*), a pri snimanju na disk, automatski im se dodeljuje ekstenzija .ino. Reprezentativni primeri programa, kao i najčešće korišćene biblioteke su uključeni u samo razvojno okruženje. Primere gotovih programa moguće je otvoriti korišćenjem opcije $File \rightarrow Examples$, nakon čega se otvara meni u okviru kojeg se izabere željeni primer. Biblioteke sa gotovim funkcijama moguće je uključiti u program korišćenjem opcije $Sketch \rightarrow Import\ Library$

Jezičke reference

U ovom delu biće opisani osnovni koncepti jezika koji se koristi za razvoj Arduino aplikacija, u koje spadaju *jezičke strukture*, *podaci (varijable i konstante)* i *funkcije*.

Jezičke strukture(1)

Kontrolne strukture

- if
- if..else
- switch..case
- for
- while
- do..while
- break
- continue
- return
- goto

Još sintakse

- ; (tačka-zarez)
- {} (vitičaste zagrade)
- // (komentar u liniji)
- /* */ (komentar u više linija)
- #define
- #include

Aritmetički operatori

- = (dodela vrednosti)
- + (sabiranje)
- - (oduzimanje)
- * (množenje)
- / (deljenje)
- % (ostatak pri deljenju)

Podaci(1)

Konstante

- HIGH | LOW
- INPUT | OUTPUT | INPUT_PULLUP
- LED_BUILTIN
- true | false
- Celobrojne konstante
- Realne konstante

Tipovi podataka

- void
- boolean
- char
- unsigned char
- byte
- int
- unsigned int
- word
- long
- unsigned long
- short
- float
- double
- string - niz karaktera
- String - objekat
- nizovi

Funkcije(1)

- setup()
- loop()

Digitalni ulazi/izlazi

- pinMode()
- digitalWrite()
- digitalRead()

Analogni ulazi/izlazi

- analogReference()
- analogRead()
- analogWrite() - PWM

Matematičke funkcije

- min()
- max()
- abs()
- constrain()
- map()
- pow()
- sqrt()
- sin()
- cos()
- tan()

Jezičke strukture(2)

Operatori poređenja

- == (jednako)
- != (nije jednako)
- < (manje od)
- > (veće od)
- ≤ (manje ili jednako)
- ≥ (veće ili jednako)

Logički operatori

- && (logičko I)
- | (logičko ILI)
- ! (logičko NE)

Bitski operatori

- & (bitsko I)
- || (bitsko ILI)
- ^ (bitsko EKS-ILI)
- ~ (bitsko invertovanje)
- << (pomeranje ulevo)
- >> (pomeranje udesno)

Složeni operatori

- ++ (inkrement)
- −− (dekrement)
- + =
- − =
- ∗ =
- / =
- & =
- | =

Podaci(2)

Konverzije između tipova

- char()
- byte()
- int()
- word()
- long()
- float()

Doseg promenljivih i kvalifikatori

- Doseg promenljivih
- static
- volatile
- const

Pomoćni operatori

- sizeof()

Funkcije(2)

Vreme

- millis()
- micros()
- delay()
- delayMicroseconds()

Slučajni brojevi

- randomSeed()
- random()

Serijska komunikacija (klasa Serial)

LCD displej (biblioteka LiquidCrystal)

if (uslov)

Kontrolna struktura **if**, koja se koristi u zajedno sa operatorom poređenja, testira da li je određen logički uslov ispunjen, npr. da li je vrednost varijable veća od određenog broja. Logički test zadovoljava sledeći format:

```
if (someVariable > 50)
{
  // uradi nesto
}
```

Program vrši proveru da li je vrednost varijable *someVariable* veća od 50. Ako jeste, program obavlja akciju koja je određena blokom naredbi unutar vitičastih zagrada. U suprotnom, program preskače kompletan kod koji je unutar zagrada. Ukoliko je u slučaju da je logički uslov ispunjen, potrebno izvršiti samo jednu naredbu, zagrade mogu biti izostavljene, a sledeća linija (čiji kraj je definisan znakom ;) postaje jedini uslovni izraz.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
    digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
    digitalWrite(LEDpin1, HIGH);
    digitalWrite(LEDpin2, HIGH);
}// svi gore navedeni izrazi su korektni
```

Logički izraz koji se navodi unutar zagrada i čija istinitosna vrednost se određuje zahteva upotrebu jednog od sledećih operatora poređenja:

Operatori poređenja

- $x == y$ (x je jednako y)
- $x\ !=\ y$ (x nije jednako y)
- $x\ <\ y$ (x je manje od y)
- $x\ >\ y$ (x je veće od y)
- $x\ <=\ y$ (x je manje ili jednako y)
- $x\ >=\ y$ (x je veće ili jednako y)

Upozorenje:

Potrebno je obratiti pažnju da se umesto operatora poređenja "==" slučajno ne upotrebi operator "=" (npr. u izrazu *if (x = 10)*). Pojedinačan znak jednakosti predstavlja operator *dodele* vrednosti,

koji u ovom slučaju dodeljuje varijabli x vrednost 10. Sa druge strane, operator poređenja koji se označava dvostrukim znakom jednakosti (npr. u izrazu *if (x == 10)*), testira da li je vrednost varijable x jednaka 10 ili ne. Ovaj poslednji izraz je tačan jedino ako je vrednost promenljive već bila jednaka 10 i pri tome ne utiče na vrednost promenljive. Sa druge strane, vrednost prvog izraza će uvek biti tačna i pri tome će vrednost promenljive biti postavljena na 10. Ovo je stoga što jezik C računa vrednost izraza *if (x = 10)* na sledeći način:

Prvo se promenljivoj x dodeljuje vrednost 10 (usled toga što "=" predstavlja operator dodele). Prema tome, sada promenljiva x ima vrednost 10. Zatim, "if" kontrolna struktura računa istinitosnu vrednost izraza 10, što se tumači kao TAČNO, budući da se u C-u svakom pozitivnom broju dodeljuje logička vrednost TAČNO. Stoga, vrednost izraza *if (x=10)* će uvek biti TAČNO, što nije željeno svojstvo "if" izraza. Uz to, menja se trenutna vrednost promenljive, što takođe predstavlja neželjen efekat.

if() takođe može predstavljati deo složenije if..else kontrolne strukture.

Nazad na jezičke reference

if..else

Kontrolna struktura **if..else** omogućava veću kontrolu nad tokom programa od običnog "if" izraza, time što omogućava grupisanje višestrukih logičkih testova. Naprimer, može se testirati vrednost koja se očitava sa analognog ulaza i pri tome se izvršava jedna akcija ako je očitana vrednost manja od 500, a druga ako je vrednost na ulazu manja ili jednaka 500. Kod bi mogao biti realizovan na sledeći način:

```
if (pinFiveInput < 500)
{
  // akcija A
}
else
{
  // akcija B
}
```

Nakon **else** može da sledi novi "if" test, čime se postiže realizacija višestrukih, međusobno isključujućih logičkih slučajeva. Svaki test vodi ka sledećem u nizu, sve dok ne bude dostignut onaj test koji daje vrednost TAČNO. Ukoliko se ispostavi da nijedan test ne daje tačnu vrednost, izvršava se blok koji sledi iza poslednjeg **else** u nizu. Pri tome, broj višestrukih **if..else** testova u nizu nije ograničen i može da bude završen bez realizacije poslednjeg **else** bloka.

```
if (pinFiveInput < 500)
{
  // akcija A
}
else if (pinFiveInput >= 1000)
{
  // akcija B
}
else
{
  // akcija C
}
```

Drugi način za realizaciju višestrukih grananja u zavisnosti od međusobno isključujućih testova je korišćenjem kontrolne strukture **switch..case**.

Pogledati još:
switch..case

Nazad na jezičke reference

switch..case

Poput **if** testova, **switch..case** konstrukcija utiče na tok programa dozvoljavajući programeru da specificira različite kkodne sekvence koje trebaju da budu izvršene u različitim uslovima. Konkretno, **switch** izraz poredi vrednost varijable sa vrednostima naznačenim u **case** izrazima. Kada je pronađen **case** izraz koji se po vrednosti poklapa sa vrednošću promenljive, izvršava se kod koji sledi nakon tog **case** izraza.

Ključna reč **break** prekida izvršavanje kodne sekvence i tipično se koristi na kraju svake od sekvenci koje slede nakon **case** izraza. Bez korišćenja **break**, krenulo bi izvršenje kodne sekvence koja sledi nakon sledećeg **case** izraza (došlo bi do "propadanja"), sve dok se negde ne naiđe na **break**, ili se dođe do kraja **switch..case** izraza.

Primer:

```
switch (var) {
    case 1:
        // akcija koja se izvrsava ako je var jednako 1
        break;
    case 2:
        // akcija koja se izvrsava ako je var jednako 2
        break;
    default:
        // ako se vrednost promenljive var ne poklapa ni sa jednom
        // od vrednosti u case izrazima, izvrsava se ova akcija

        // default je opciona akcija, koja ne mora da postoji u kodu
}
```

Sintaksa:

```
switch (var) {
    case value_1:
        // kodna sekvenca
        break;
    case value_2:
        // kodna sekvenca
        break;
    default:
        // kodna sekvenca
}
```

Parametri:

- var: promenljiva čija vrednost se poredi sa onima u **case** izrazima
- value_x: vrednost sa kojom se promenljiva poredi

Pogledati još: if..else

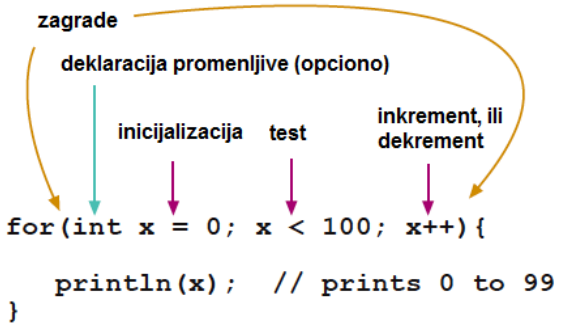
Nazad na jezičke reference

for petlja

Opis:

for petlja se koristi za ciklično ponavljanje bloka naredbi obuhvaćenih vitičastim zagradama. Pri tome se obično koristi brojačka promenljiva koja se inkrementira u svakoj iteraciji i koristi se pri konstruisanju uslova za izlazak iz petlje. Ovakav izraz je generalno moguće upotrebiti kod proizvoljne repetitivne operacije, a često se primenjuje i kod operacija koje manipulišu nizovima podataka. Zaglavlje **for** petlje sadrži sledeće osnovne elemente:

```
for (inicijalizacija; uslov; inkrement){
 // blok naredbi
}
```



Inicijalizacija se dešava tačno jednom i to na početku. Pri svakom prolasku kroz petlju (iteraciji), testira se logički uslov; ako je ispunjen, izvršava se blok naredbi unutar vitičastih zagrada, a zatim inkrement. Nakon toga, ponovo se prelazi na testiranje uslova čime započinje nova iteracija. Prvi put kada logički test ne bude ispunjen, petlja se okončava.

Primer:

```
// Dimovanje LED diode koriscenjem PWM pina
int PWMpin = 10;     // LED dioda je vezana redno
                     // sa otpornikom od 470 oma na pinu 10

void setup()
{
    // nema potrebe za inicijalizacijom
}

void loop()
{
    for (int i=0; i <= 255; i++){
        analogWrite(PWMpin, i);
        delay(10);
    }
}
```

Korisni saveti:

U programskom jeziku C, **for** petlja je mnogo fleksibilnija u odnosu na for petlje koje postoje u nekim drugim jezicima, kao što je npr. BASIC. Bilo koji, ili čak sva 3 elementa zaglavlja mogu da budu izostavljeni, ali moraju ostati simboli ";" koji ih razdvajaju. Takođe, izrazi za inicijalizaciju, uslov i inkrement mogu biti bilo koji validni C izrazi, u kojima mogu da učestvuju i promenljive nevezane za rad same petlje, a moguće je i korišćenje svih tipova podataka, uključujući i realne brojeve. Takve "neuobičajene" konstrukcije **for** petlje mogu da predstavljaju rešenje pojedinih "egzotičnih" problema u programiranju. Na primer, korišćenjem množenja u izrazu za inkrement generiše se eksponencijalna progresija:

```
for (int x = 2; x < 100; x = x * 1.5){
    println(x);
}
// petlja generise eksponencijalnu sekvencu brojeva:
// 2,3,4,6,9,13,19,28,42,63,94
```

Još jedan primer, koji u okviru iste petlje povećava, pa zatim smanjuje nivo osvetljenosti LED diode:

```
void loop()
{
    int x = 1;
    for (int i = 0; i > -1; i = i + x){
        analogWrite(PWMpin, i);
        if (i == 255)
            x = -1; // na vrhuncu se menja smer promene brojaca
        delay(10);
   }
}
```

Pogledati još:
while petlja

Nazad na jezičke reference

while petlja

Opis:

while petlja se izvršava kontinualno, dok god je ispunjen logički uslov u zagradama. Ukoliko neki izraz unutar petlje ne promeni vrednost logičkog izraza, **while** petlja će se beskonačno izvršavati. Ova promena može biti postignuta npr. inkrementom logičke promenljive, ili promenom stanja hardverskog okruženja, kao što je očitavanje senzora.

Sintaksa:

```
while(uslov){
 // blok naredbi
}
```

Parametri:

- uslov: logički izraz, čija vrednost može biti TAČNO ili NETAČNO

Primer:

```
var = 0;
while(var < 200){
    // akcija koja se ponavlja 200 puta
    var++;
}
```

Pogledati još:
for petlja
do..while petlja

Nazad na jezičke reference

do..while petlja

Opis:

do..while petlja funkcioniše na isi način kao **while** petlja, s tom razlikom da se logički izraz testira na kraju petlje, tako da je neminovno da će se blok naredbi unutar petlje izvršiti bar jednom.

Sintaksa:

do{
// blok naredbi
} while (uslov);

Primer:

```
do
{
    delay(50);           // pauza, dok se senzori stabilizuju
    x = readSensors();   // ocitavanje senzora

} while (x < 100);
```

Pogledati još:
while petlja

Nazad na jezičke reference

break

Opis:

break se koristi za izlazak iz **for**, **while**, ili **do..while** petlje, ignorišući pri tome redovan uslov za ostanak u petlji. Takođe se koristi za izlazak iz **switch** izraza.

Primer:

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // izlazak iz petlje ako je ocitavanje senzora
                                // vece od postavljenog praga
        x = 0;
        break;
    }
    delay(50);
}
```

Nazad na jezičke reference

continue

Opis:

Izraz **continue** preskače ostatak trenutne iteracije petlje (**for**, **while** ili **do..while**). Nakon **continue**, odmah se prelazi na proveru logičkog uslova petlje i nastavlja se sa daljim iteracijama.

Primer:

```
for (x = 0; x < 255; x ++)
{
    if (x > 40 && x < 120){ // preskace se podinterval izlaznih vrednosti
        continue;
    }

    digitalWrite(PWMpin, x);
    delay(50);
}
```

Nazad na jezičke reference

return

Opis:

return prekida izvršenje funkcije i vraća vrednost funkciji iz koje je obavljen poziv, ukoliko je to potrebno.

Sintaksa:

return;
return value; // obe forme su validne

Parametri:

- value: bilo koja vrednost koja je istog tipa kao i funkcija

Primer:

```
// funkcija vraca odgovor da li je ocitavanje senzora
// vece od unapred odredjenog praga
int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;
    else{
        return 0;
    }
}
```

Ključna reč **return** je zgodna kada se radi testiranje sekcije koda, bez potrebe da se "zakomentarišu" veće sekcije koda koje potencijalno sadrže greške.

```
void loop(){

// brilijantna ideja koju treba istestirati

return;

// disfunkcionalni ostatak koda
// koji nikad nece biti izvrsen
}
```

Nazad na jezičke reference

goto

Opis:

Vrši bezuslovni skok, odnosno preusmerava tok programa od obeležene linije koda.

Sintaksa:

label:
goto label; // nastavlja izvrsenje programa od labele

Savet:

Upotreba **goto** skokova se ne preporučuje u C programiranju i generalno se smatra lošom programerskom praksom. Iako je dokazano da se umesto korišćenja **goto**, isti efekat uvek može postići korišćenjem ostalih "standardnih" izraza, ipak u pojedinim slučajevima može doprineti pojednostavljenju programskog koda. Jedan od razloga zašto mnogi programeri zaziru od ove mogućnosti je to što se nepažljivom upotrebom **goto** izraza dobija kod sa teško predvidljivim tokom, koji je gotovo nemoguće debagovati.

Sa druge strane, u nastavku je prikazan primer gde je **goto** izraz iskorišćen u svrhu pojednostavljenja koda. Jedna od takvih situacija je izlazak iz strukture duboko ugneždenih **for** petlji, ili **if** logičkih blokova, pod određenim uslovima.

Primer:

```
for (byte r = 0; r < 255; r++){
    for (byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto izlaz;}
            // jos ugnezdenih "if" ili "for" izraza
        }
    }
}
izlaz:
```

Nazad na jezičke reference

; (tačka-zarez)

Opis:

Koristi se kao terminator kojim se završava izraz.

Primer:

int a = 13;

Savet:

Jedan od podmuklijih tipova grešaka pri kompajliranju nastaje usled slučajno izostavljenog simbola **tačka-zarez**. Kompajlerski izveštaj o grešci može, a i ne mora nedvosmisleno da ukazuje na izostavljen terminator. Ako se pri kompajliranju pojavi nelogičan ili naizgled zbunjujući izveštaj o grešci, jedna od prvih stvari na koju treba posumnjati je upravo izostavljena **tačka-zarez** u neposrednoj blizini, obično neposredno ispred linije na kojoj je kompajler prijavio grešku.

Nazad na jezičke reference

{} (vitičaste zagrade)

Opis:

Vitičaste zagrade su važan deo programskog jezika C. Koriste se u nekoliko različitih jezičkih konstrukta, koji su prikazani u nastavku. Otvorena vitičasta zagrada "{" uvek i bez izuzetka mora biti praćena zatvorenom vitičastom zagradom "}". Ovakav zahtev se često naziva *balansiranjem zagrada*. Arduino IDE u sebi sadrži korisno svojstvo da olakšava proveru balansiranosti zagrada. Klikom na poziciju neposredno posle simbola vitičaste zagrade, automatski će biti istaknuta pozicija njenog logičkog para, bez obzira na to da li se radi o otvorenoj ili zatvorenoj zagradi.

Pošto je upotreba vitičastih zagrada vrlo raširena u tipičnom C programu, dobra je praksa odmah po otvaranju zagrade dodati zatvorenu zagradu, a zatim pritiskom tastera enter osloboditi mesto na kojem će se nalaziti blok naredbi ograničen zagradama.

Nebalansirane zagrade mogu da dovedu do "čudnih", naizgled često neobjašnjivih grešaka pri kompajliranju, koje je obično teško ispraviti, pogotovo u dužem programu. Usled različitih načina njihove upotebe, vitičaste zagrade su neverovatno važne za sintaksu koda i pomeranje zagrade za liniju ili dve koda može dramatično izmeniti funkcionalnost programa.

Najvažniji primeri upotrebe:

Funkcije

```
void myfunction(datatype argument){
    // blok naredbi
}
```

Petlje

```
while (boolean expression)
{
    // blok naredbi
}
```

```
do
{
    // blok naredbi
} while (boolean expression);
```

```
for (initialisation; termination condition; incrementing expr)
{
    // blok naredbi
}
```

Uslovni izrazi

```
if (boolean expression)
{
    // blok naredbi
}

else if (boolean expression)
{
    // blok naredbi
}
else
{
    // blok naredbi
}
```

Nazad na jezičke reference

Komentari

Opis:

Komentari su tekstualne napomene programu čija svrha je obaveštavanje (podsećanje) programera vezano za način na koji program funkcioniše. Kompajler ih ignoriše i ne prosleđuje procesoru, odnosno ne prevodi u mašinski kod, pa stoga ne zauzimaju prostor u programskoj memoriji mikrokontrolera. Ostavljanje komentara u programu nije obavezno, ali predstavlja deo dobre programerske prakse i vrlo je korisno u situacijama kada je potrebno vratiti se i vršiti izmene u kodu koji je napisan davno, ili od strane druge osobe. Postoje dva načina za postavljanje komentara.

Primer:

```
x = 5;  // Ovo je komentar u jednoj liniji. U komentar spada sve
        // sto je napisano iza dvostruke kose crte, do kraja linije.


/* ovo je komentar u vise linija - koristi se za komentarisanje blokova koda
if (gwb == 0){  // dozvoljeno je koriscenje komentara u jednoj liniji,
                // u okviru komentara koji se protezu na vise linija,
x = 3;           /* ali ne i koriscenje drugog "ugnezdenog" komentara u vise
    linija - ovo je greska! */
}
// ne zaboravite da "zatvorite" komentar - komentari moraju biti balansirani!
*/
```

Savet:

Prilikom eksperimentisanja sa kodom, ”zakometarisanje” dela koda je zgodan način za uklanjanje linija koje potencijalno sadrže greške. Ovim se takve linije ne uklanjaju iz koda, ali se postiže da ih kompajler ignoriše. To može biti naročito korisno pri pokušajima lociranja problema u programu, ili kada program odbija da se kompajlira, a kompajlerski izveštaj o grešci deluje zbunjujuće.

Nazad na jezičke reference

#define

Opis:

#define je korisna C direktiva koja omogućava programeru da dodeli naziv konstanti ili makrou. Makro je proizvoljan niz znakova koji može predstavljati brojnu konstantu, string, naredbu (ili čak niz naredbi) i sl. Definisani makroi zauzimaju prostor unutar Arduino programske memorije. Neposredno pre kompajliranja, kompajler pronalazi sva mesta u programskom kodu na kojima se pojavljuju slovne oznake makroa i zamenjuje ih samim makroima.

Pri korišćenju makroa treba biti oprezan, jer je moguća pojava neželjenih propratnih efekata[2], na primer ukoliko je naziv makroa koji je definisan uključen u naziv neke od varijabli ili konstanti. Generalna preporuka je da se u slučaju kada je potrebno definisati konstantu koristi ključna reč **const**, umesto **#define**.

Sintaksa:

```
#define constantName value
```

Primeri:

```
#define ledPin 3 // kompajler ce svaku pojavu ledPin u kodu zameniti vrednoscu 3.
#define bajt unsigned char
```

Savet:

Treba izbegavati terminator tačka-zarez nakon **#define** izraza, pošto će u tom slučaju kompajler ubaciti i terminator u makro, što može rezultovati čudnim greškama pri kompajliranju. Iz istog razloga, u **#define** izraz ne treba ubacivati znak jednakosti.

```
#define ledPin 3; // ovo je greska
#define ledPin = 3 // takodje greska
```

Pogledati još:
const

Nazad na jezičke reference

[2] Oindikacijamameramaoprezaineželjenimreakcijamaposavetovatisesalekaromilifarmaceutom.

#include

Opis:

#include se koristi kada je potrebno uključiti eksterne biblioteke u program. Ovim se programeru omogućava pristup velikoj grupi standardnih C biblioteka, kao i biblioteka funkcija koje su namenski pisane za Arduino platformu.

Primer:

```
#include <math.h>
```

U ovom primeru u program je uključena biblioteka koja sadrži standardne matematičke funkcije.

Nazad na jezičke reference

= (operator dodele vrednosti)

Opis:

Ovaj operator vrednost izraza desno od znaka jednakosti smešta u promenljivu levo od znaka jednakosti. Pojedinačan znak "=" se u C-u naziva operatorom dodele vrednosti.

Primer:

```
int sensVal;                // definicija celobrojne promenljive sensVal
sensVal = analogRead(0);    // ocitavanje digitalnog ekvivalenta
                            // napona na analognom ulazu 0
                            // smesta se u promenljivu SensVal
```

Korisni saveti:

Promenljiva sa leve strane operatora dodele (odnosno znaka jednakosti) treba da bude odgovarajućeg tipa koji ima dovoljan opseg da može u sebe da primi vrednost izraza. Ukoliko to nije slučaj, vrednost koja se smešta u promenljivu neće biti korektna.

Takođe, neophodno je voditi računa o razlici između operatora dodele "=" i operatora poređenja "==".

Pogledati još:
if

Nazad na jezičke reference

Sabiranje, oduzimanje, množenje i deljenje

Opis:

Ovi operatori vraćaju zbir, razliku, proizvod, odnosno količnik 2 operanda. Operacija se obavlja imajući u vidu tip operanada, pa npr. izraz 9/4 daje rezultat 2, pošto su 9 i 4 celi brojevi. Ovo takođe znači da može doći do prekoračenja opsega, ukoliko je rezultat veći od maksimalne vrednosti koja može biti smeštena u promenljivu tog tipa (npr. dodavanjem vrednosti 1 na promenljivu tipa *int* čja vrednost je 32767 daje rezultat -32768). Ako su operandi različitog tipa, tip rezultata će biti "veći" od ta dva tipa. Ako je jedan od operanada tipa *float* ili *double*, pri izračunavanju će biti korišćena aritmetika u pokretnom zarezu.

Primeri:

$y = y + 3;$
$x = x - 7;$
$i = j * 6;$
$r = r/5;$

Sintaksa:

$rezultat = vrednost1 + vrednost2;$
$rezultat = vrednost1 - vrednost2;$
$rezultat = vrednost1 * vrednost2;$
$rezultat = vrednost1/vrednost2;$

Parametri:

- vrednost1, vrednost2: bilo koja promenljiva ili konstanta

Korisni saveti:

- Kod celobrojnih izračunavanja, podrazumevani (default) tip je *int*, što pri pojedinim izračunavanjima dovodi do prekoračenja opsega (npr. 60 * 1000 će dati negativan rezultat).
- Potrebno je izabrati tip varijable koji je dovoljno velikog opsega da može da primi najveći mogući rezultat za dato izračunavanje.
- Preporučljivo je za različite tipove podataka znati pri kojim vrednostima dolazi do premašaja.
- Kod izračunavanja u kojima figurišu realni brojevi izraženi u više decimala, koriste se promenljive tipa *float*, ali pri tome treba voditi računa o njihovim manama: veće zauzeće memorije, sporija izračunavanja.
- Korišćenjem operatora kastovanja (konverzije), npr. (int)myFloat, moguće je vršiti konverziju između različitih tipova "u letu".

Nazad na jezičke reference

% (ostatak pri deljenju)

Opis:

Izračunava ostatak pri deljenju dva cela broja.

Sintaksa:

ostatak = deljenik % delilac

Parametri:

- deljenik: broj koji se deli
- delilac: broj kojim se deli

Povratna vrednost:

ostatak pri deljenju

Primeri:

```
x = 7 % 5;   // x dobija vrednost 2
x = 9 % 5;   // x dobija vrednost 4
x = 5 % 5;   // x dobija vrednost 0
x = 4 % 5;   // x dobija vrednost 4
```

```
/* azurira jedan clan niza u svakoj iteraciji */
int values[10];
int i = 0;

void setup() {}

void loop()
{
    values[i] = analogRead(0);
    i = (i + 1) % 10;   // kada indeks dodje do kraja, vraca se na pocetak
}
```

Napomena:
Ovaj operator ne radi sa realnim brojevima (u pokretnom zarezu).
Pogledati još:
/ (deljenje)

Nazad na jezičke reference

Logički operatori

Najčešće se koriste unutar logičkog uslova u okviru **if** izraza.

&& (logičko I)

Vrednost izraza je istinita jedino ako oba operanda imaju vrednost TAČNO.

Primer:

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) {
    // ocitavaju se stanja dva pekidaca
    // ovde je kod koji se izvrsava samo ako su
    // oba prekicaca pritisnuta istovremeno
}
```

|| (logičko ILI)

Vrednost izraza je istinita jedino ako bar jedan od operanada ima vrednost TAČNO.

Primer:

```
if (x > 0 || y > 0) {
    // ovde je kod koji se izvrsava
    // ako bar jedna od promenljivih ima vrednost vecu od 0

    // ekivalentan uslov moze da se napise kao if(x || y)
    // posto se vrednost promenljive tumaci kao TACNO
    // ako je veca od 0
}
```

! (logičko NE)

Vrednost izraza je istinita ako operand ima vrednost NETAČNO.

Primer:

```
if (!x) {
    // kod se izvrsava ako je vrednost promenljive x NETACNO,
    // odnosno ako je x <= 0
}
```

Upozorenje:

Neophodno je obratiti pažnju na razliku između operatora && **(logičko I)** i & **(bitsko I)**, budući da se radi o fundamentalno različitim operacijama. Slično, ne treba brkati || **(logičko ILI)** i | **(bitsko ILI)**, kao i **!** **(logičko NE)** i ~(bitsko invertovanje).

Pogledati još:
& (bitsko I)
| (bitsko ILI
~(bitsko invertovanje)
if

Nazad na jezičke reference

Bitski operatori

Bitski operatori obavljaju svoje operacije nad varijablama na nivou parova bita na odgovarajućim pozicijama. U nastavku su navedene definicije i sintaksa vezana za ovakve operatore.

& (bitsko I)

Operator **bitsko I** koji je u C-u predstavljen simbolom &, povezuje dva celobrojna izraza. I operacija se obavlja simultano i nezavisno na svakom paru bita koji odgovaraju istoj poziciji u binarnom zapisu, u skladu sa pravilom: ako su oba ulazna bita 1, izlaz je takođe jednak 1, u suprotnom je 0. Drugi način za izražavanje istog pravila je:

0 0 1 1 *operand_1*
0 1 0 1 *operand_2*
———
0 0 0 1 (*operand_1* & *operand_2*) - rezultat operacije

Kod Arduina, celobrojni tip *int* predstavlja 16-bitnu vrednost, tako da korišćenje operatora & izaziva istovremeno obavljanje 16 I operacija nad parovima bita na odgovarajućim pozicijama. Na primer, u programskoj sekvenci:

```
int a =  92;    // binarno:  0000000001011100
int b = 101;    // binarno:  0000000001100101
int c = a & b;  // rezultat: 0000000001000100, odnosno 68 decimalno
```

svaki od 16 parova bita brojeva a i b učestvuje u bitskoj I operaciji, a 16 rezultujućih bita se smeštaju u promenljivu c, čime se dobija vrednost 00000000 01000100 binarno, odnosno 68 decimalno. Jedna od uobičajenih primena bitske I operacije je selekcija određenih bita u okviru celobrojne vrednosti, što se naziva *maskiranjem* i ilustrovano je primerom prikazanim u nastavku.

| (bitsko ILI)

Operator **bitsko ILI** predstavljen ja simbolom |. Slično & operatoru, operacija se vrši nezavisno na svakom paru bita celobrojnih vrednosti koje učestvuju u izrazu. Bitska ILI operacija izvršena nad parom bita, daje rezultat 1 ukoliko je bar jedan od ulaznih bita 1, u suprotnom je 0. Drugim rečima:

0 0 1 1 *operand_1*
0 1 0 1 *operand_2*
———
0 1 1 1 (*operand_1* | *operand_2*) - rezultat operacije

Primer programske sekvence u kojoj se koristi bitsko ILI:

```
int a =  92;     // binarno:  0000000001011100
int b = 101;     // binarno:  0000000001100101
int c = a | b;   // rezultat: 0000000001111101, odnosno 125 decimalno
```

Primer programa:

Bitski I i ILI operatori se vrlo često koriste u Očitaj-Izmeni-Upiši (engl. Rad-Modify-Write) operacijama na portovima. Kod mikrokontrolera, port je 8-bitni broj koji predstavlja stanja pinova. Upis vrednosti u port uzrokuje istovremeno ažuriranje logičih vrednosti na svim pinovima.

PORTD je promenljiva koja je povezana sa 8-bitnim registrom stanja digitalnih pinova 0,1,2,3,4,5,6,7. Ako je na poziciji određenog bita logička jedinica, napon na pinu je visok i obratno (prethodno, pin mora da bude podešen kao izlaz, korišćenjem pinMode() komande). Dakle, dodelom vrednosti PORTD = B10001100; pinovi 2, 3 i 7 se postavljaju na visoko stanje. Ovde potencijalno postoji problem što se takođe utiče na stanja pinova 0 i 1 koji se koriste za serijsku komunikaciju, što može dovesti do greške u komunikaciji.

Program implementira sledeći algoritam:

- Očita se stanje PORTD i pomoću bitskog I, selektovani biti se postavljaju na 0, bez promene stanja ostalih bita.
- Modifikovana vrednost PORTD se kombinuje sa vrednostima bita koje treba postaviti na 1, korišćenjem bitske ILI operacije.

```
int i;  // brojacka promenljiva
int j;

void setup(){
    DDRD = DDRD | B11111100;// biti 2 do 7 postaju izlazni
                            // bez promene funkcija pina 0 i 1 (xx | 00 == xx)

                            // isto se postize uzastopnim pozivanjem
                            // pinMode(pin, OUTPUT) za pinove 2 do 7
    Serial.begin(9600);
}

void loop(){
    for (i=0; i<64; i++){
        PORTD = PORTD & B00000011;  // pinovi 2 - 7 se postavljaju na 0,
                                    // pinovi 0 i 1 se ne menjaju (xx & 11 ==
                                       xx)
        j = (i << 2);               // pomeranje varijable udesno za 2 bita
        PORTD = PORTD | j;          // kombinovanjem trenutnog stanja porta
                                    // sa novim stanjima LED dioda
        Serial.println(PORTD, BIN); // prikaz stanja PORTD (zbog debagovanja)
        delay(100);
    }
}
```

^ (bitsko EKS-ILI)

Operator **bitsko EKS-ILI** predstavljen ja simbolom ^. Ovaj operator funkcioniše na sličan način kao i prethodno opisani bitski operatori, sprovođenjem logičke operacije na svakom paru bita ponaosob. Rezultat operacije je 1 ako su vrednosti ulaznih bita međusobno različite, a 0 ako su iste:

0 0 1 1 *operand_1*
0 1 0 1 *operand_2*
———
0 1 1 0 (*operand_1* ^ *operand_2*) - rezultat operacije

Primer programske sekvence u kojoj se koristi bitsko EKS-ILI:

```
int x = 12;      // binarno: 1100
int y = 10;      // binarno: 1010
int z = x ^ y;   // binarno: 0110, odnosno decimalno 6
```

Operator ^se često koristi u svrhu invertovanja (tj. promene sa 0 na 1, ili sa 1 na 0) pojedinačnih bita u okviru celobrojne vrednosti. U ovom slučaju, na mestima gde se u masci nalaze jedinice, ti biti se invertuju, dok ostali ostaju nepromenjeni. U nastavku je prikazan program koji kontroliše treptanje LED diode priključene na pin 5, tako što joj menja stanje korišćenjem bitske EKS-ILI operacije:

```
// treptanje LED diode na pinu 5
// demonstrira upotrebu bitskog EKS-ILI operatora
void setup(){
    DDRD = DDRD | B00100000; // digitalni pin 5 je izlaz
    Serial.begin(9600);
}

void loop(){
    PORTD = PORTD ^ B00100000;  // invertuj stanje bita 5, bez promene stanja
        ostalih
    delay(100);
}
```

Pogledati još:
&& (logičko I)
| (logičko ILI)

Nazad na jezičke reference

~ (bitsko invertovanje)

Bitski NE operator je predstavljen simbolom ~. Za razliku od operatora & i |, operator invertovanja se primenjuje na jedan operand, koji se navodi sa desne strane. Bitsko NE menja vrednost svakog bita u onu koja joj je suprotna u logičkom smislu (0 postaje 1, a 1 postaje 0). Primer:

0 1 *operand*

---

1 0 (~*operand*) - rezultat operacije

```
int a = 103;     // binarno:  0000000001100111
int b = ~a;      // binarno:  1111111110011000 = -104
```

Kao što se vidi u prethodnom primeru, invertovanjem pozitivnog broja je dobijena negativna vrednost. To nije iznenađujuće, ako se ima u vidu način reprezentovanja negativnih brojeva u komplementu dvojke: invertovanjem svakog bita proizvoljne celobrojne vrednosti x dobija se vrednost -x-1, što je vrednost koju treba sabrati sa 1 da bi se dobilo -x.

Pogledati još:
! (logičko NE)

Nazad na jezičke reference

<< (pomeranje ulevo) i >> (pomeranje udesno)

Opis:

U C-u postoje dva operatora pomeranja (engl. Shift): operator pomeranja ulevo << i operator pomeranja udesno >>. Ovi operatori uzrokuju da svi biti levog operanda budu pomereni ulevo ili udesno za onoliko mesta koliko iznosi vrednost desnog operanda.

Sintaksa:

promenljiva << broj_bita
promenljiva >> broj_bita

Parametri:

- promenljiva - celobrojna promenljiva (byte, int, long)
- broj_bita - ceo broj ≤ 32

Primer:

```
int a = 5;         // binarno: 0000000000000101
int b = a << 3;    // binarno: 0000000000101000, ili 40 decimalno
int c = b >> 3;    // binarno: 0000000000000101, nazad na pocetnu vrednost
```

Pri pomeranju vrednosti x ulevo za y bita (x << y), krajnjih y bita sa leve strane se gube (tj. "ispadaju napolje"), dok se na mesto krajnjih y bita sa desne strane upisuju nule:

```
int a = 5;          // binarno: 0000000000000101
int b = a << 14;    // binarno: 0100000000000000 - prva jedinica u 101
                    //                             je "ispala napolje"
```

Alternativni način shvatanja operatora pomeranja ulevo je množenje vrednosti prvog operanda sa brojem 2 dignutim na stepen koji odgovara vrednosti drugog operanda. Drugim rečima, $a << b$ odgovara vrednosti $a \cdot 2^b$. Na primer, stepeni dvojke mogu biti dobijeni na sledeći način:

$1 << 0 == 1$
$1 << 1 == 2$
$1 << 2 == 4$
$1 << 3 == 8$
...
$1 << 8 == 256$
$1 << 9 == 512$
$1 << 10 == 1024$
...

Pri pomeranju vrednosti x udesno za y bita (x >> y), ako je najviši bit broja x jednak jedinici, rezultat zavisi od toga kog tipa je x. Ako je x označeni tip (recimo *int*), u tom slučaju najviši bit ima značenje predznaka (0 ako je broj pozitivan, a 1 ako je negativan). U tom slučaju, najviši bit se kopira u y nižih bita pri pomeranju. Time se postiže da pomeranje udesno za y mesta u aritmetičkom smislu predstavlja deljenje sa 2^y, bez obzira na to da li se radi o pozitivnom ili negativnom broju:

```
int x = -16;      // binarno: 1111111111110000
int y = x >> 3;   // binarno: 1111111111111110
```

Ovakvo ponašanje, koje se naziva još ekstenzijom predznaka, ne mora uvek biti poželjno. U tom slučaju, ako je potrebno da pri pomeranju vrednosti kojima se broj popunjava obavezno budu nule, može se upotrebiti operator konverzije, kojim se postiže da vrednost bude tretirana kao neoznačen ceo broj:

```
int x = -16;                    // binarno: 1111111111110000
int y = (unsigned int)x >> 3;   // binarno: 0001111111111110
```

Nazad na jezičke reference

++ (inkrement) i −− (dekrement)

Opis:

Inkrement, ili dekrement promenljive.

Sintaksa:

x++; // povećava x za 1 i vraća staru vrednost x (postinkrement)
++x; // povećava x za 1 i vraća novu vrednost x (preinkremet)
x– ; // smanjuje x za 1 i vraća staru vrednost x (postdekrement)
–x ; // smanjuje x za 1 i vraća novu vrednost x (predekrement)

Parametri:

- x: celobrojna promenljiva

Povratna vrednost:

Stara, ili nova inkrementirana/dekrementirana vrednost promenljive.

Primeri:

```
x = 2;
y = ++x;      // x dobija vrednost 3, y takodje postaje 3
y = x--;      // x se vraca na 2, y je i dalje 3
```

Pogledati još:
+=
-=

Nazad na jezičke reference

+= , -= , *= , /=

Opis:

Aritmetičke operacije se obavljaju nad operandima sa leve i desne strane operatora, nakon čega se rezultat smešta u levi operand. Ovo je kompaktniji način za izražavanje operacija koje su navedene u nastavku.

Sintaksa:

x += y; // ekvivalent izraza x = x + y; x -= y; // ekvivalent izraza x = x - y; x *= y; // ekvivalent izraza x = x * y; x /= y; // ekvivalent izraza x = x / y;

Parametri:

- x: promenljiva proizvoljnog tipa
- y: promenljiva ili konstanta proizvoljnog tipa

Primeri:

```
x = 2;
x += 4;      // x sada ima vrednost 6
x -= 3;      // x sada ima vrednost 3
x *= 10;     // x sada ima vrednost 30
x /= 2;      // x sada ima vrednost 15
```

Nazad na jezičke reference

&= (složeno bitsko I)

Opis:

Složeni bitski I operator (& =) se najčešće koristi kako bi se biti promenljive na određenim pozicijama postavili na nisko stanje (resetovali).

Sintaksa:

x &= y; // ekvivalent izraza x = x & y;

Parametri:

- x: celobrojna promenljiva
- y: celobrojna konstanta ili promenljiva

Primer:

Ako je na primer potrebno resetovati (postaviti na 0) bite na pozicijama 0 i 1 neke osmobitne varijable, a da pri tome vrednosti ostalih bita ne promene, koristi se operator &= u kombinaciji sa konstantom B11111100. Ova konstanta se u programerskom žargonu naziva *maskom*.

1 0 1 0 1 0 1 0 promenljiva
1 1 1 1 1 1 0 0 maska
———————-
1 0 1 0 1 0 0 0 rezultat

Evo ponovo iste operacije, s tim što su biti promenljive zamenjeni simbolom x:

x x x x x x x x promenljiva
1 1 1 1 1 1 0 0 maska
———————-
x x x x x x 0 0 rezultat

Prema tome, ako je:
myByte = 10101010;
Tada važi:
myByte &= B1111100 == B10101000;

Pogledati još:
| =
& (bitsko I)

Nazad na jezičke reference

| = (složeno bitsko ILI)

Opis:

Složeni bitski ILI operator (| =) se najčešće koristi kako bi se biti promenljive na određenim pozicijama postavili na visoko stanje (setovali).

Sintaksa:

x | = y; // ekvivalent izraza x = x | y;

Parametri:

- x: celobrojna promenljiva
- y: celobrojna konstanta ili promenljiva

Primer:

Ako je na primer potrebno setovati (postaviti na 1) bite na pozicijama 0 i 1 neke osmobitne varijable, a da pri tome vrednosti ostalih bita ne promene, koristi se operator | = u kombinaciji sa konstantom B00000011. Ova konstanta se u programerskom žargonu naziva *maskom*.

1 0 1 0 1 0 1 0 promenljiva
0 0 0 0 0 0 1 1 maska
———————-
1 0 1 0 1 0 1 1 rezultat

Evo ponovo iste operacije, s tim što su biti promenljive zamenjeni simbolom x:

x x x x x x x x promenljiva
0 0 0 0 0 0 1 1 maska
———————-
x x x x x x 1 1 rezultat

Prema tome, ako je:
myByte = 10101010;
Tada važi:
myByte | = B00000011 == B10101011;

Pogledati još:
&=
| (bitsko ILI)

Nazad na jezičke reference

Konstante

Ovde će biti reči o predefinisanim konstantama u Arduino jeziku, koje se koriste kako bi programski kod bio čitljiviji i lakše razumljiv. Postoji više podgrupa Arduino konstanti.

Predstavljanje logičkih nivoa (true i false)

Dve konstante reprezentuju istinitosne vrednosti u Arduino jeziku: **true** i **false**.

false
Od ove dve konstante, **false** je jednostavnija za definisanje, pošto se se definiše kao 0 (nula).

true
Za konstantu **true** se često smatra da ima vrednost 1, što jeste korektno, ali **true** ima širu definiciju. U logičkom smislu, svaki ceo broj različit od nule se u logičkom smislu tumači kao tačno. Prema tome, -1, 2 i -200 predstavljaju tačne logičke vrednosti.

Konstante **true** i **false** se pišu malim slovima, za razliku od konstanti **HIGH**, **LOW**, **INPUT** i **OUTPUT**.

Definisanje nivoa na pinovima (HIGH i LOW)

Pri očitavanju ili upisivanju vrednosti digitalnog pina, postoje dva moguća stanja u kojima pin može da se nalazi, odnosno koja mogu da budu podešena kao izlazna vrednost: visoko stanje koje predstavlja logičku jedinicu (**HIGH**) i nisko stanje koje predstavlja logičku nulu (**LOW**).

HIGH

Značenje konstante **HIGH** se razlikuje u zavisnosti od toga da li je pin podešen da bude ulaz (**INPUT**) ili izlaz (**OUTPUT**). Kada se pomoću funkcije **pinMode** funkcija pina podesi na **INPUT**, pri čitanju stanja funkcija **digitalRead** vraća **HIGH**, ako je napon na pinu veći od 3V.

Ako je pin podešen na **INPUT** pomoću funkcije **pinMode**, a nakon toga se u njega upiše vrednost **HIGH** pomoću funkcije **digitalWrite**, ovim se uključuje interni pull-up otpornik vrednosti 20K, čija svrha je da postavi stanje pina na logičku jedinicu, ali dozvoljava obaranje stanja na logičku nulu od strane eksternih komponenti. Ujedno, ovo je način na koji radi **INTERNAL_PULLUP**.

Kada je pin konfigurisan na **OUTPUT** i stanje mu je podešeno na **HIGH** funkcijom **digitalWrite**, na njemu se postavlja napon od 5V. U ovom stanju, pin može da "daje" struju npr. za uključenje LED diode koja je vezana na red sa otpornikom između napajanja i pina.

LOW

Konstanta **LOW** takođe ima različito značenje u zavisnosti od toga da li je pin podešen kao **INPUT** ili **OUTPUT**. Kada je pin podešen kao **INPUT** korišćenjem funkcije **pinMode**, pri čitanju stanja funkcija **digitalRead** vraća **LOW**, ako je napon na pinu manji od 2V.

Kada je pin konfigurisan na **OUTPUT** i stanje mu je podešeno na **LOW** funkcijom **digitalWrite**, na njemu se postavlja napon od 0V. U ovom stanju, pin može da "vuče" struju npr. za uključenje LED diode koja je vezana na red sa otpornikom između pina i mase.

Definisanje funkcionalnosti digitalnih pinova (INPUT, INPUT_PULLUP i OUTPUT)

Digitalni pinovi mogu biti konfigurisani na tri načina, kao ulaz **INPUT**, **INPUT_PULLUP** ili izlaz **OUTPUT**. Promena funkcionalnosti pina pomoću funkcije **pinMode** utiče na električno ponašanje pina.

Pinovi konfigurisani kao ulazi (INPUT)

Arduino (Atmega) pinovi konfigurisani kao ulazi (**INPUT**) pomoću funkcije **pinMode()** se nalaze u stanju visoke impedanse i kao takvi se odlikuju izuzetno velikom ulaznom otpornošću reda veličine 100 Megaoma. To čini ovaj režim rada pogodnim npr. za očitavanje senzora i drugih ulaznih uređaja.

Pinovi konfigurisani kao (INPUT_PULLUP)

Atmega čip u okviru Arduina poseduje interne pull-up otpornike (otpornike koje je moguće interno spojiti na napon napajanja). Ova opcija se aktivira prosleđivanjem argumenta **INPUT_PULLUP** funkciji **pinMode**.

Pinovi konfigurisani kao izlazi (OUTPUT)

Za Arduino (Atmega) pinove konfigurisane kao izlazi (**OUTPUT**) pomoću funkcije **pinMode()** se kaže da su u stanju niske impedanse. To znači da su u stanju da izađu na kraj sa značajnom strujom koja se prosleđuje, ili dolazi od strane eksternog kola. Atmega pinovi mogu da "daju" , ili "gutaju" struju jačine do 40mA, što je recimo sasvim dovoljno za upravljanje LED diodama, ali ne i za upravljanje potrošačima koji zahtevaju veću struju kao što su elektromagnetni releji i motori (u tom slučaju su potrebne dodatne eksterne komponente kao što su prekidački tranzistori). Takođe, treba voditi računa o tome da pinovi konfigurisani kao izlazi mogu da budu oštećeni ili uništeni ako se kratko spoje na napajanje u trenutku kada je izlazna vrednost 0 (0V), odnosno na masu u trenutku kada je izlazna vrednost 1 (5V).

LED_BUILTIN

Većina Arduino ploča imaju pin koji je povezan sa LED diodom koja je ugrađena na samoj ploči, u kombinaciji sa rednim otpornikom. **LED_BUILTIN** služi kao zamena za "ručno" podešavanje ovog pina kao logičke promenljive. Kod većine ploča, LED dioda je spojena na digitalni pin 13.

Pogledati još:
pinMode()
Celobrojne konstante

Logičke promenljive

Nazad na jezičke reference

Celobrojne konstante

Celobrojne konstante su brojevi koji se direktno koriste u programu, kao npr. 123. Podrazumeva se da se celi brojevi tretiraju kao tip **int**, ali to se može primeniti upotrebom modifikatora U ili L (videti dole). Uobičajeno je da se celobrojne konstante tretiraju kao decimalni brojevi (tj. u sistemu sa osnovom 10), ali specijani prefiksi omogućavaju unos brojeva u drugim brojnim sistemima.

Osnova sistema	Primer	Prefiks	Validni karakteri
10 (decimalni)	123	ne postoji	0-9
2 (binarni)	B1111011	B	0-1
8 (oktalni)	0173	0	0-7
16 (heksadecimalni)	0x7B	0x	0-9, A-F, a-f

Decimalni sistem je sistem sa osnovom 10, koji se uobičajeno koristi u svakodnevnoj praksi. Za konstante bez frefiksa se podrazumeva da su u decimalnoj formi.

Primer:

101 // decimalni broj 101 $(1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0)$

Binarni je sistem sa osnovom 2. Samo cifre 0 i 1 su dozvoljene.

Primer:

B101 // isto sto i 5 decimalno $(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$

Binarni prefiks radi samo sa vrednostima ograničenim na 8 bita, između 0 (B0) i 255 (B11111111). Ako je potrebno uneti 16-bitnu binarnu vrednost (u slučaju da je promenljiva tipa *int*), to se može postići u dva koraka, na sledeći način:

myInt = (B11001100 * 256) + B10101010; // B11001100 je viši bajt

Oktalni je sistem sa osnovom 8. Validne su cifre 0-7. Oktalne vrednosti imaju prefiks 0.

Primer:

0101 // isto sto i 65 decimalno $(1 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0)$

Upozorenje:

Postoji potencijalna opasnost od nenamernog generisanja greške koja je teška za otkrivanje, ako se slučajno ispred decimalne konstante navede vodeća nula, što dovodi do toga da kompajler tumači konstantu kao oktalnu vrednost.

Heksadecimalni je sistem sa osnovom 16. Validne cifre su 0-9 i slova A-F (odnosno a-f). A ima vrednost 10, B je 11, sve do F koje ima vrednost 15. Heksadecimalne vrednosti imaju prefiks "0x".

Primer:

0x101 // isto sto i 257 decimalno $(1 \cdot 16^2 + 0 \cdot 16^1 + 1 \cdot 16^0)$

Modifikatori U i L

Pri upotrebi celobrojnih konstanti, podrazumeva se da one pripadaju opsegu koji nameće tip **int**. Međutim, korišćenjem modifikatora moguće je kreirati konstante koje pripadaju drugim celobrojnim tipovima i to:

- 'u' ili 'U' pretvaraju konstantu u neoznačeni celobrojni tip. Primer: 33u
- 'l' ili 'L' pretvaraju konstantu u 32-bitni (long) format. Primer: 100000L
- 'ul' ili 'UL' pretvaraju konstantu u 32-bitni neoznačeni (unsigned long) format. Primer: 32767ul

Pogledati još:
#define
byte
int
int
unsigned int
long
unsigned long

Nazad na jezičke reference

Realne konstante (u pokretnom zarezu)

Slično celobrojnim konstantama, realne konstante se koriste u svrhu poboljšanja čitljivosti koda. Realne konstante se tokom kompajliranja zamenjuju vrednostima u koje se preračunavaju.

Primer:

float f = .005;

Konstante u pokretnom zarezu se mogu izraziti u više ekvivalentnih notacija. 'E' i 'e' oba predstavljaju validne eksponencijalne indikatore.

Realna konstanta	Preračunava se u	Takođe se preračunava u
10.0	10	
$2.34E5$	$2.34 * 10^5$	234000
$67e - 12$	$67.0 * 10^{-12}$	.000000000067

Nazad na jezičke reference

void

Ključna reč **void** se koristi isključivo u deklaracijama funkcija i označava da funkcija ne vraća nikakvu povratnu informaciju.

Primer:

```
// funkcije "setup" and "loop" obavljaju odredjene operacije
// ali ne vracaju nikakvu informaciju "glavnom" programu iz kojeg su pozvane
void setup()
{
    // ...
}

void loop()
{
    // ...
}
```

Pogledati još:
Deklaracije funkcija

Nazad na jezičke reference

boolean

Tip **boolean** se odnosi na logičke promenljive i može imati jednu od dve moguće vrednosti (true ili false). Promenljiva tipa **boolean** zauzima 1 bajt u memoriji.

Primer:

```
int LEDpin = 5;        // LED na pinu 5
int switchPin = 13;    // prekidac je spojen izmedju pina 13 i mase

boolean running = false;

void setup()
{
    pinMode(LEDpin, OUTPUT);
    pinMode(switchPin, INPUT);
    digitalWrite(switchPin, HIGH);  // ukljucuje pull-up otpornik
}

void loop()
{
    if (digitalRead(switchPin) == LOW)
    {   // prekidac je pritisnut - u suprotnom pull-up otpornik diktira stanje
        1 na pinu
        delay(100);                     // kasnjenje, kako bi se ustalilo
            stanje pina
                                        // posle "zvecanja" tastera
        running = !running;             // promena stanja promenljive running
        digitalWrite(LEDpin, running)   // indikacija preko LED diode
    }
}
```

Pogledati još:
Konstante
Logički operatori
Promenljive

Nazad na jezičke reference

char

Opis:

Tip podataka koji zauzima 1 bajt memorije i u koji je smećten znakovni karakter. Karakteri se navode korišćenjem apostrofa, npr. 'A' (za grupe više karaktera - stringove koriste se navodnici, npr "ABC").

Karakteri su smešteni u memoriju kao brojevi koji predstavljaju ASCII[3] kodove pojedinih karaktera. To znači da je moguće obavljati aritmetičke operacije sa promenljivama ovog tipa (npr. 'A' + 1 ima vrednost 66, pošto je ASCII kod slova 'A' 65).

Tip **char** je označeni 8-bitni tip, što znači da uzima vrednosti od -128 do 127. Za neoznačene 8-bitne brojeve, koristi se tip **byte**.

Primer:

```
char myChar = 'A';
char myChar = 65; // obe deklaracije su ekvivalentne
```

Pogledati još:
byte
int
nizovi
Serial.println

Nazad na jezičke reference

[3] ASCII (American Standard Code for Information Interchange) je standardni kod za predstavljanje alfanumeričkih i specijalnih karaktera u kojem je svakom karakteru pridružen jedinstven numerički kod u rasponu 0-127.

unsigned char

Opis:

Ovo je neoznačeni celobrojni tip koji zauzima 1 bajt memorije. Ekvivalentan je tipu byte. Promenljiva tipa **unsigned char** uzima vrednosti iz opsega 0 do 255.

Zbog konzistentnosti sa Arduino stilom programiranja, preporučuje se upotreba tipa **byte**.

Primer:

```
unsigned char myChar = 240;
```

Pogledati još:
byte
int
nizovi
Serial.println

Nazad na jezičke reference

byte

Opis:

Promenljiva ovog tipa sadrži 8-bitni neoznačen broj, od 0 do 255.

Primer:

byte b = B10010; // "B" je binarni prefiks (B10010 = 18 decimalno)
See also

Pogledati još:
word
byte()
Deklaracije promenljivih

Nazad na jezičke reference

int

Opis:

Ovo je primarni tip koji se koristi za definisanje celobrojnih promenljivih. Na Arduino Uno (i ostalim pločama baziranim na ATMega kontrolerima) int zauzima 16 bita, odnosno 2 bajta u memoriji. To znači da je dozvoljeni opseg vrednosti od -2147483648 do 2147483647 (odnosno od -2^{31} do $2^{31}-1$). Na Arduino Due ploči, **int** je 32-bitni (4-bajtni). To znači da je dozvoljeni opseg vrednosti od -32768 do 32767 (odnosno od -2^{15} do $2^{15}-1$).

Promenljive tipa **int** su označene. Za predstavljanje negativnih vrednosti se koristi aritmetika komplementa dvojke. Pogodnost vezana za ovu tehniku je što aritmetičke operacije same po sebi rade transparentno i na očekivan način, dajući na kraju ispravnu vrednost koja može biti pozitivna ili negativna. Međutim, potrebno je obratiti pažnju na potencijalne komplikacije koje mogu nastati upotrebom operatora pomeranja udesno (>>).

Primer:

```
int ledPin = 13;
```

Sintaksa:

```
int var = val;
```

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Upozorenje:

Pri operacijama sa celobrojnim vrednostima, ako dođe do prekoračenja, vrednost se naglo ”prevaljuje” sa maksimalne na minimalnu vrednost za dati opseg. Ovaj fenomen se manifestuje u oba smera. Primer za 16-bitnu **int** promenljivu:

```
int x;
x = -32768;
x = x - 1; // x sada ima vrednost 32767
x = 32767;
x = x + 1; // x sada ima vrednost -32768
```

Pogledati još:
unsigned int
Celobrojne konstante
Deklaracije promenljivih

Nazad na jezičke reference

unsigned int

Opis:

Na UNO i ostalim Arduino pločama sa ATMEGA kontrolerima, **unsigned int** je 16-bitni ceo broj. Razlika između ovog tipa i **int** je što se u ovom slučaju sve vrednosti interpretiraju kao neoznačeni (pozitivni) brojevi, što rezultuje opsegom od 0 do 65535, odnosno od 0 do $2^{16} - 1$. Kod Due ploča, ovaj tip je 32-bitni (4-bajtni), sa vrednostima u opsegu od 0 do $2^32 - 1 = 4294967295$.

Generalno, razlika između označenih i neoznačenih tipova je u načinu interpretiranja najvišeg bita, koji se stoga još naziva i "bit znaka".Kod tipa **int** koji je označen, ako je najviši bit 1, broj je negativan i interpretira se u aritmetici komplementa dvojke.

Primer:

unsigned int ledPin = 13;

Sintaksa:

unsigned int var = val;

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Upozorenje:

Kada promenljiva premaši maksimalnu vrednost, "prevaljuje" se ponovo na minimalnu vrednost, što je u ovom slučaju 0. Ovaj fenomen se manifestuje u oba smera.

unsigned int x;
x = 0;
x = x - 1; // x sada ima vrednost 65535
x = x + 1; // x sada ima vrednost 0
See Also

Pogledati još:
int
long
unsigned long
Deklaracije promenljivih

Nazad na jezičke reference

word

Opis:

Tip **word** označava 16-bitni neoznačeni broj u opsegu od 0 do 65535. Ekvivalentan je tipu **unsigned int**.

Primer:

```
word w = 10000;
```

Pogledati još:
byte
word()

Nazad na jezičke reference

long

Opis:

Promenljive tipa **long** koriste se za smeštanje velikih celobrojnih vrednosti. Koriste se 32 bita (4 bajta) za smeštanje brojeva u opsegu od -2147483648 do 2147483647. Ako se obavljaju matematičke operacije sa celobrojnim konstantama, bar jedna od vrednosti mora biti praćena slovom L (čime se eksplicitno naglašava da se radi o vrednosti tipa **long**), inače će se cela operacija vršiti u 16-bitnoj **int** aritmetici, što može dovesti do greške usled prekoračenja opsega.

Primer:

long speedOfLight = 300000L;

Sintaksa:

long var = val;

Parametri:

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Pogledati još:
byte
int
unsigned int
unsigned long
Celobrojne konstante
Deklaracije promenljivih

Nazad na jezičke reference

unsigned long

Opis:

Tip **Unsigned long** je 32-bitni (4-bajtni) neoznačeni celobrojni tip. Za razliku od običnog **long** tipa, u ovom slučaju u obzir dolaze samo pozitivne vrednosti, od 0 do 4294967295 ($2^{32} - 1$).

Primer:

```
unsigned long time;

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

void loop()
{
    Serial.print("Vreme: ");
    time = millis();
    //ispisuje broj milisekundi od pocetka programa
    Serial.println(time);
    // ceka da prodje sekund, da bi se izbeglo slanje ogromnih kolicina
       podataka
    delay(1000);
}
```

Sintaksa:

unsigned long var = val;

Parametri:

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Pogledati još:
byte
int
unsigned int
long
Celobrojne konstante
Deklaracije promenljivih

Nazad na jezičke reference

short

Opis:

Na svim Arduino platformama, **short** označava 16-bitni (2-bajtni) označeni celobrojni tip. Opseg vrednosti je od -32768 do 32767 (odnosno od -2^{15} do $2^{15} - 1$).

Primer:

short ledPin = 13;

Sintaksa:

short var = val;

Parametri:

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Pogledati još:
byte
int
unsigned int
long
Celobrojne konstante
Deklaracije promenljivih

Nazad na jezičke reference

float

Opis:

Tip podataka za realne brojeve, odnosno brojeve sa decimalnom tačkom. Ovakvi brojevi se često koriste za aproksimaciju analognih i kontnualnih vrednosti, budući da imaju veću rezoluciju od celih brojeva. Brojevi u pokretnom zarezu mogu uzimati vrednosti između -3.4028235E+38 i 3.4028235E+38. Promenljive tipa **float** zauzimaju 4 bajta u memoriji.

Promenljiva ovog tipa je obično u stanju da reprezentuje 6-7 značajnih decimalnih cifara. Za razliku od drugih platformi, gde se preciznost povećava korišćenjem tipa **double** (čak i do 15 cifara), na Arduino ATMEGA kontrolerima **double** i **float** tip su međusobno ekvivalentni.

Brojevi u pokretnom zarezu se ne mogu smatrati tačnim nego približnim vrednostima i pri poređenju mogu produkovati čudne rezultate. Na primer, 6.0 / 3.0 ne mora da iznosi tačno 2.0. Umesto poređenja koje podrazumeva da je dobijena tačna vrednost, bolje je proveriti da li je razlika između vrednosti manja od neke unapred zadate male vrednosti.

Aritmetika u pokretnom zarezu je mnogo sporija od celobrojne i u principu je treba izbegavati, ako npr. iteracija petlje treba da se izvrši pre određenog kritičnog vremenskog trenutka. Stoga programeri obično ulažu dodatni napor da svedu računicu na celobrojnu aritmetiku.

Da bi bila korišćena aritmetika pokretnog zareza, bar jedna od brojnih vrednosti mora sadržati decimalnu tačku, u suprotnom će kompajler odraditi račun korišćenjem celobrojne aritmetike.

Primeri:

float myfloat; float sensorCalbrate = 1.117;

Sintaksa:

float var = val;

Parametri:

- var - naziv promenljive
- val - vrednost koja se dodeljuje promenljivoj

Primer kodne sekvence:

```
int x;
int y;
float z;

x = 1;
y = x / 2;            // y ima vrednost 0, int odbacuje sve iza decimalnog
    zareza
z = (float)x / 2.0;   // z ima vrednost .5 (mora se navesti vrednost 2.0,
    umesto 2)
```

Pogledati još:
int
double
Deklaracije promenljivih

Nazad na jezičke reference

double

Opis:

Realan broj u pokretnom zarezu, dvostruke preciznosti. Na Uno i ostalim ATMEGA pločama zauzima 4 bajta, što znači da je implementiran na isti način kao i **float**, bez dobitka na preciznosti. Na Arduono Due pločama, **double** ima 8-bajtnu (64-bitnu) reprezentaciju.

Savet:

Korisnicima koji "portuju" (prebacuju) kod sa druge platforme se preporučuje da provere kod kako bi utvrdili da li se podrazumevana preciznost razlikuje od one koju obezbeđuje Arduino sistem baziran na ATMEGA kontroleru.

Pogledati još:
float

Nazad na jezičke reference

string - niz karaktera

Opis:

Tekstualni stringovi mogu biti reprezentovani na 2 načina. Može se koristiti tip **String**, ili se implementirati na klasičan "C-ovski" način kao niz karaktera sa nul-terminatorom. Ovde je opisan drugi od dva spomenuta načina. Više detalja o String objektu dato je u posebnom odeljku.

Primeri:

U nastavku je navedeno nekoliko različitih validnih deklaracija stringova.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Načini deklarisanja stringova:

1. Deklarisanje niza karaktera, bez inicijalizacije, kao u Str1.
2. Deklarisanje niza karaktera (sa jednim dodatnim karakterom), pri čemu će kompajler sam dodati nul-karakter na kraju, kao u Str2.
3. Isto kao prethodni slučaj, uz eksplicitno dodavanje nul-karaktera, kao u Str3.
4. Inicijalizacija string konstantom unutar znakova navoda; kompajler će automatski rezervisati niz čija veličina je jednaka dužini stringa, uz automatsko dodavanje nul-karaktera, kao u Str4.
5. Isto kao prethodni slučaj, uz eksplicitno zadavanje dužine niza, kao u Str5.
6. Inicijalizacija stringa, uz rezervisanje dodatnog prostora kako bi u njega kasnije mogao da bude smešten duži string, kao u Str6.

Nul-terminacija

Generalno, stringovi završavaju nul-karakterom (ASCII kod 0). Ovo Omogućava funkcijama kao što je **Serial.print()** da znaju gde se nalazi kraj stringa. U suprotnom, ovakve funkcije bi nastavile dalje da čitaju naredne vrednosti iz memorije (koje ne pripadaju stringu), sve dok eventualno ne naiđu na nul-karakter. Ovo predstavlja neželjeno ponašanje i može da dovede do "pucanja" programa, tako da uvek treba obratiti pažnju na terminaciju.

Ovo ujedno znači da niz u kojem se smešta string treba da bude bar za jedan bajt duži nego što je dužina teksta. Zbog toga su Str2 i Str5 dužine 8, iako "arduino" sadrži 7 karaktera - poslednja pozicija se automatski popunjava nul-karakterom. Str4 će automatski biti dimenzionisan na 8 karaktera.

Jednostruki ili dvostruki navodnici?

Stringovi se uvek definišu unutar dvostrukih znakova navoda (”Abc”), dok se pojedinačni karakteri definišu unutar jednostrukih navodnika (’A’).

Dugački stringovi mogu se prelomiti u više redova na sledeći način:

char myString[] = ”Ovo je prva linija”
” Ovo je druga linija”
” blabla truc truc”;

Nizovi stringova

Često je zgodno, kada se radi sa većim količinama teksta, kao što je recimo projekat na kojem se za ispis teksta koristi LCD displej, definisati niz stringova. Budući da su stringovi nizovi sami po sebi, ovo zapravo predstavlja primer dvodimenzionalnog niza (matrice). U dole navedenom primeru, simbol zvezdice iza naziva tipa char (”char*”) označava da se radi o nizu ”pokazivača”. Zapravo, svi nazivi nizova su ništa drugo do pokazivači na prvi član niza, tako da se na ovaj način definiše niz nizova karaktera. Pokazivači predstavljaju jedan od ”mračnijih” koncepata jezika C za razumevanje od strane programera početnika, ali u ovom slučaju nije neophodno njihovo duboko poznavanje da bi mogli da budu uspešno upotrebljeni.

Primer:

```
char* myStrings[]={
"Ovo je string 1",
"Ovo je string 2",
"Ovo je string 3",
"Ovo je string 4",
"Ovo je string 5",
"Ovo je string 6"};

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

void loop(){
    for (int i = 0; i < 6; i++){
        Serial.println(myStrings[i]);
        delay(500);
    }
}
```

Pogledati još:

nizovi
String objekat
Nazad na jezičke reference

String - objekat

Opis:

Klasa **String** omogućava manipulaciju tekstualnim stringovima na kompleksnije načine u odnosu na obične nizove karaktera. Između ostalog, podržava konkatenaciju (nadovezivanje) Stringova, pretragu i zamenu podstringova itd. Zauzima više memorije nego niz karaktera, ali sa druge strane pruža više mogućnosti. Da bi se izbegla konfuzija, nizovi karaktera se nazivaju string (sa malim 's'), a instance String klase se nazivaju String sa velikim 'S'. String konstante, navedene između dvostrukih navodnika se tretiraju kao obični nizovi karaktera, a ne kao objekti String klase.

Obilje primera i korisnih referenci vezanih za String klasu (na engleskom jeziku) nalazi se u odgovarajućoj sekciji zvanične Arduino web stranice:

http://arduino.cc/en/Reference/StringObject

Nazad na jezičke reference

nizovi

Niz je kolekcija promenljivih kojima se pristupa preko indeksa, koji predstavlja poziciju promenljive u okviru niza.

Kreiranje (deklarisanje) niza

U nastavku su prikazane validne metode za kreiranje (deklarisanje) niza

```
int myInts[6];
int myPins[] = 2, 4, 8, 3, 6;
int mySensVals[6] = 2, 4, -8, 3, 2;
char message[6] = "hello";
```

Niz je moguće deklarisati bez inicijalizacije, kao kod *myInts*.

U primeru *myPins*, niz se deklariše bez eksplicitnog navođenja dužine. Kompajler automatski prebrojava elemente i kreira niz odgovarajuše dužine. KOnačno, moguće je istovremeno inicijalizovati niz i zadati mu dužinu, kao kod *mySensVals*. Kada se deklariše niz tipa **char**, potrebno je da niz sadrži jedan dodatni element, za smeštanje nul-karaktera.

Pristup elementima niza

Indeksi u nizovima **uvek** počinju od nule, odnosno prvi element niza obavezno ima indeks 0. Prema tome:

mySensVals[0] == 2, mySensVals[1] == 4, i tako redom.

To takođe znači da u nizu od 10 elemenata, poslednji element je onaj sa indeksom 9:

```
int myArray[10]=9,3,2,4,3,2,7,8,9,11;
// myArray[9] ima vrednost 11
// myArray[10] ne pripada nizu, nego se nalazi u "tuđoj" memoriji
```

Iz ovih razloga, treba biti oprezan kada se pristupa nizovima. Pristup iza poslednjeg elementa (korišćenjem indeksa većeg od deklarisane veličine niza umanjene za 1) dovodi do čitanja iz memorije koja je rezervisana u druge svrhe. U gorem slučaju, ako se upisuje vrednost u lokacije koje su indeksirane izvan opsega rezervisanog za niz, može doći do ozbiljnijih grešaka koje dovode do "pucanja" programa. Ovakve greške su obično teške za uočiti, budući da prolaze kompajliranje (C kompajler ne vrši proveru da li je indeks niza izvan opsega, za razliku od nekih drugih jezika kao što su BASIC ili Java).

Dodela vrednosti elementima niza:

mySensVals[0] = 10;

Čitanje vrednosti elemenata niza:

x = mySensVals[4];

Nizovi i **for** petlje:

Manipulacije nad nizovima se često vrše unutar **for** petlji, gde se brojačka promenljiva koristi za indeksiranje elemenata niza. U sledećem primeru, elementi niza se korišćenjem petlje ispisuju preko serijskog porta:

```
int i;
for (i = 0; i < 5; i++) {
    Serial.println(myPins[i]);
}
```

Pogledati još:
Deklaracije promenljivih

Nazad na jezičke reference

char()

Opis:

Konvertuje vrednost u tip **char**.

Sintaksa:

char(x)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa

Vraća vrednost tipa:

char

Pogledati još:
char

Nazad na jezičke reference

byte()

Opis:

Konvertuje vrednost u tip **byte**.

Sintaksa:

byte(x)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa

Vraća vrednost tipa:

byte

Pogledati još:
byte

Nazad na jezičke reference

int()

Opis:

Konvertuje vrednost u tip **int**.

Sintaksa:

int(x)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa

Vraća vrednost tipa:

int

Pogledati još:
int

Nazad na jezičke reference

word()

Opis:

Konvertuje vrednost u tip **word**, ili kreira word od 2 bajta.

Sintaksa:

word(x)
word(h,l)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa
- h: viši (značajniji) bajt
- l: niži (manje značajan) bajt

Vraća vrednost tipa:

word

Pogledati još:
word

Nazad na jezičke reference

long()

Opis:

Konvertuje vrednost u tip **long**.

Sintaksa:

long(x)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa

Vraća vrednost tipa:

long

Pogledati još:
long

Nazad na jezičke reference

float()

Opis:

Konvertuje vrednost u tip **float**.

Sintaksa:

float(x)

Parametri:

- x: promenljiva ili konstanta proizvoljnog tipa

Vraća vrednost tipa:

float

Pogledati još:
float

Nazad na jezičke reference

Doseg promenljivih

Promenljive u varijanti jezika C u kojem se programira Arduino imaju svojstvo zvano **doseg**:

- **Globalna promenljiva** je promenljiva koja je "vidljiva" iz svake funkcije u programu.
- **Lokalna promenljiva** je vidljiva samo u funkciji u kojoj je deklarisana.

U Arduino okruženju, promenljive koje su deklarisane izvan funkcija (npr. *setup()*, *loop()* i sl.) su globalne promenljive.

Kada program raste i postaje sve kompleksniji, lokalne promenljive predstavljaju koristan način da se obezbedi da svaka funkcija ima pristup samo svojim promenljivama. Ovim se sprečavaju greške koje bi mogle nastati ako jedna funkcija promeni vrednost promenljive koja pripada drugoj funkciji.

Takođe, ponekad je zgodno deklarisati i inicijalizovati promenljivu unutar **for** petlje. Ovim se kreira promenljiva kojoj je moguće pristupiti samo unutar vitičastih zagrada koje obuhvataju telo petlje.

Primer:

```
int gPWMval;  // svaka funkcija "vidi" ovu promenljivu

void setup()
{
    // ...
}

void loop()
{
    int i;    // "i" je jedino "vidljiva" unutar funkcije "loop"
    float f;  // "f" je jedino "vidljiva" unutar funkcije "loop"
    // ...

    for (int j = 0; j <100; j++){
        // promenljivoj "j" je moguce pristupiti samo unutar
        // viticastih zagrada koje obuhvataju for petlju
    }

}
```

Nazad na jezičke reference

static

Ključna reč **static** se koristi za kreiranje promenljivih koje su "vidljive" samo u svojoj funkciji. Međutim, za razliku od lokalnih promenljivih koje se kreiraju i uništavaju pri svakom pozivu funkcije, statičke promenljive zadržavaju vrednost i između dva poziva funkcije.

Promenljive koje su deklarisane kao **static** se kreiraju i inicijalizuju samo pri prvom pozivu funkcije.

Primer:

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

void loop()
{
    static int i = 0;        // i se kreira samo pri prvom pozivu,
                             // nakon cega zadrzava vrednost
                             // po izlasku iz funkcije

    Serial.println(i++);     // i se inkrementira pri svakom pozivu loop()
    delay(1000);
}
```

Nazad na jezičke reference

volatile

volatile je ključna reč poznata kao *kvalifikator promenljive*, obično korišćen ispred naziva tipa promenljive, kako bi modifikovao način na koji kompajler, a zatim i sam program tretira promenljivu.

Proglašenje promenljive za **volatile** je kompajlerska direktiva. Kompajler je program koji prevodi C/C++ kod u mašinski kod, koji se fizički izvršava na Atmega čipu Arduino sistema.

Konkretno, ovim se nalaže kompajleru da učita vrednost promenljive iz RAM memorije, umesto iz internog registra, koji predstavlja lokaciju gde se promenljiva privremeno smešta, kako bi procesor mogao da njome manipuliše. Pod određenim okolnostima, vrednost promenljive smeštene u registru može biti nestalna.

Promenljiva treba da bude proglašena kao **volatile** kadgod njena vrednost može biti izmenjena od strane nečega što je izvan kontrole dela koda u kom se pojavljuje, kao što je recimo nit koja se paralelno izvršava. Kod Arduina, jedino mesto gde je moguće da se tako nešto desi je deo koda povezan sa prekidima, kao što je potprogram za obradu prekida.

Primer:

```
// menja stanje LED diode, na promenu stanja pina za eksterni prekid

int pin = 13;
volatile int state = LOW;

void setup()
{
    pinMode(pin, OUTPUT);
    attachInterrupt(0, blink, CHANGE);
}

void loop()
{
    digitalWrite(pin, state);
}

void blink()
{
    state = !state;
}
```

Pogledati još:
AttachInterrupt

Nazad na jezičke reference

const

Ključna reč **const** označava konstantu. To je kvalifikator koji modifikuje ponašanje promenljive, na taj način da je prevodi u stanje gde ju je moguće samo očitati (eng. *Read only*). To znači da je promenljivu moguće koristiti isto kao svaku drugu promenljivu tog tipa, osim što joj nije moguće promeniti vrednost. Pri pokušaju dodele vrednosti **const** promenljivoj, kompajler će prijaviti grešku. Konstante definisane na ovaj način se pokoravaju ostalim pravilima vezanim za doseg promenljivih. Ovo i potencijalni problemi pri korišćenju **#define** čine ključnu reč **const** superiornim metodom koji se preporučuje za definisanje konstanti.

Primer:

```
const float pi = 3.14;
float x;

// ....
x = pi / 2;     // konstante se regularno koriste u matematickim izrazima

pi = 7;         // GRESKA - nemoguce je modifikovati konstantu
```

Pogledati još:
#define

Nazad na jezičke reference

sizeof()

Opis:

Operator **sizeof** vraća veličinu (u bajtima) varijable, ili broj bajta koje u memoriji zauzima niz.

Sintaksa:

sizeof(variable)

Parametri:

- variable: promenljiva bilo kog tipa (npr. int, float, byte), ili niz

Primer:

Operator **sizeof** je koristan pri radu sa nizovima (kao što su stringovi), gde je zgodno imati mogućnost promene dužine niza, bez negativnog uticaja na ostale delove programa.
Sledeći primer štampa tekstualni string, karakter po karakter.

```
char myStr[] = "ovo je test";
int i;

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

void loop() {
    for (i = 0; i < sizeof(myStr) - 1; i++){
        Serial.print(i, DEC);
        Serial.print(" = ");
        Serial.write(myStr[i]);
        Serial.println();
    }
    delay(5000);
}
```

Nazad na jezičke reference

setup()

Funkcija **setup()** se poziva na početku programa. Koristi se za inicijalizaciju promenljivih, za definisanje funkcija pinova i sl. Ova funkcija se izvršava samo jednom, nakon dovođenja napajanja na ploču ili nakon pritiska reset tastera.

Primer:

```
int buttonPin = 3;              //taster je spojen na pin 3

void setup()
{
    Serial.begin(9600);         //inicijalizacija serijskog porta; brzina =
        9600
    pinMode(buttonPin, INPUT);  //inicijalizacija pina 3 kao digitalnog ulaza
}

void loop()
{
    // ...
}
```

Pogledati još:
loop

Nazad na jezičke reference

loop()

Nakon kreiranja **setup()** funkcije koja postavlja inicijalne vrednosti, funkcija **loop()** se poziva i izvršava u beskonačnoj petlji, čime se programu daje mogućnost promene vrednosti promenljivih, u cilju reagovanja na događaje u okruženju.

Primer:

```
const int buttonPin = 3;

// setup inicijalizuje serijski port i ulazni pin
void setup()
{
    Serial.begin(9600);
    pinMode(buttonPin, INPUT);
}

// loop proverava stanje pina svake sekunde
// i salje obavestenje o njegovom stanju preko serijskog potra
void loop()
{
    if (digitalRead(buttonPin) == HIGH)
        Serial.write('H');
    else
        Serial.write('L');
    delay(1000);
}
```

Pogledati još:
setup

Nazad na jezičke reference

pinMode()

Opis:

Konfiguriše izabrani pin da se ponaša kao ulaz, ili kao izlaz. Za više informacija, pogledati referancu o konstantama INPUT, OUTPUT i INPUT_PULLUP za definisanje funkcionalnosti digitalnih pinova. Od verzije Arduino 1.0.1, moguće je uključiti interne pull-up otpornike korišćenjem opcije INPUT_PULLUP. Uz to, opcija INPUT eksplicitno isključuje interne pull-up otpornike.

Sintaksa:

pinMode(pin, mode)

Parametri:

- pin: broj pina koji se konfiguriše
- mode: INPUT, OUTPUT, ili INPUT_PULLUP

Povratna vrednost:

Nema.

Primer:

```
int ledPin = 13;                // LED je spojena na pin 13

void setup()
{
    pinMode(ledPin, OUTPUT);    // podesava pin kao izlaz
}

void loop()
{
    digitalWrite(ledPin, HIGH); // ukljucuje LED
    delay(1000);                // ceka sekund
    digitalWrite(ledPin, LOW);  // iskljucuje LED
    delay(1000);                // ceka sekund
}
```

Napomena:

Analogni ulazni pinovi takođe mogu biti korišćeni kao digitalni pinovi, kojima se pristupa kao A0, A1, itd.

Pogledati još:

Konstante
digitalWrite()
digitalRead()
Nazad na jezičke reference

digitalWrite()

Opis:

Postavlja digitalni pin na visoku (**HIGH**), ili nisku (**LOW**) vrednost.

Ako je pin pomoću **pinMode()** funkcije konfigurisan kao izlaz, napon na njemu će biti postavljen na 5V ako je stanje pina **HIGH** (ili 3.3V, u zavisnosti od toga koji napon napajanja koristi konkretna ploča), odnosno na 0V, ako je stanje pina **LOW**.

Ako je pin konfigurisan kao ulaz (**INPUT**), funkcija **digitalWrite(HIGH)** uključuje interni pull-up, a **digitalWrite(LOW)** ga isključuje.

NAPOMENA: Ako se **pinMode()** ne podesi kao **OUTPUT**, u slučaju kada je LED dioda priključena na pin, poziv **digitalWrite(HIGH)** će uključiti LED diodu, koja će u tom slučaju svetleti vrlo slabo. To će se desiti zato što je na ovaj način uključen pull-up otpornik koji je vezan na red sa LED diodom, a struja će biti male vrednosti zbog velike otpornosti pull-up otpornika.

Sintaksa:

digitalWrite(pin, value)

Parametri:

- pin: broj pina
- value: vrednost HIGH ili LOW

Povratna vrednost:

Nema.

Primer:

```
int ledPin = 13;                // LED je spojena na pin 13

void setup()
{
    pinMode(ledPin, OUTPUT);    // podesava pin kao izlaz
}

void loop()
{
    digitalWrite(ledPin, HIGH); // ukljucuje LED
    delay(1000);                // ceka sekund
    digitalWrite(ledPin, LOW);  // iskljucuje LED
    delay(1000);                // ceka sekund
}
```

Napomena:

Analogni ulazni pinovi takođe mogu biti korišćeni kao digitalni pinovi, kojima se pristupa kao A0, A1, itd.

Pogledati još:
pinMode()
digitalRead()

Nazad na jezičke reference

digitalRead()

Opis:

Očitava vrednost naznačenog digitalnog pina, koja može biti visoka (**HIGH**), ili niska (**LOW**).

Sintaksa:

digitalRead(pin)

Parametri:

- pin: broj digitalnog pina čije stanje se očitava

Povratna vrednost:

HIGH ili LOW.

Primer:

```
int ledPin = 13;     // LED je spojena na digitalni pin 13
int inPin = 7;       // taster je spojen na digitalni pin 7
int val = 0;         // promenljiva u koju se smesta ocitana vrednost

void setup()
{
    pinMode(ledPin, OUTPUT);     // postavlja pin 13 kao izlaz
    pinMode(inPin, INPUT);       // postavlja pin 7 kao ulaz
}

void loop()
{
    val = digitalRead(inPin);    // ocitava ulazni pin
    digitalWrite(ledPin, val);   // postavlja stanje LED diode
                                 // da odgovara stanju tastera
}
```

Ako pin "visi", tj. nije spojen ni na kakav ulazni uređaj, funkcija **digitalRead()** vraća **HIGH** ili **LOW**, pri čemu se ne može sa sigurnošću znati koja od ove dve vrednosti će biti vraćena (pošto se stanje pina pod dejstvom šuma može menjati na slučajan način).

Analogni ulazni pinovi takođe mogu biti korišćeni kao digitalni pinovi, kojima se pristupa kao A0, A1, itd.

Pogledati još:
pinMode()
digitalWrite()

Nazad na jezičke reference

analogReference()

Opis:

Ova funkcija konfiguriše referentni napon koji se koristi pri A/D konverziji (odnosno vrednost napona koja određuje maksimum ulaznog opsega). Opcije su:

- DEFAULT: uobičajena naponska referenca od 5V (na Arduino pločama koje se napajaju na 5V), odnosno 3.3V (na Arduino pločama koje se napajaju na 3.3V)
- INTERNAL: ugrađena referenca, koja iznosi 1.1V na ATMega168 i ATMega 328, odnosno 2.56V na ATMega8.
- INTERNAL1V1: ugrađena referenca od 1.1V (samo na Arduino Mega pločama)
- INTERNAL2V56: ugrađena referenca od 2.56V (samo na Arduino Mega pločama)
- EXTERNAL: kao referenca se koristi napon doveden na AREF pin (od 0 do 5V)

Sintaksa:

analogReference(type)

Parametri:

- type: tip naponske reference koja se koristi (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, ili EXTERNAL).

Povratna vrednost:

Nema.

Napomena:

Nakon promene analogne reference, prvih nekoliko očitavanja funkcijom **analogRead()** može biti neprecizno.
After changing the analog reference, the first few readings from analogRead() may not be accurate.

Upozorenje:

Nije preporučljivo dovoditi manje od 0V ili više od 5V na AREF pin! Ako se koristi eksterna naponska referenca na AREF pinu, potrebno je prvo podesiti analognu referencu na EXTERNAL, pre poziva funkcije **analogRead()**. U suprotnom, doći će do kratkog spoja između aktivne naponske reference (interno generisane) i AREF pina, što može dovesti do fizičkog oštećenja mikrokontrolera na Arduino ploči.

Alernativno, eksterna referenca se spaja na AREF pin preko otpornika od 5K, čime se obezbeđuje "bezbolna" promena između interne i eksterne reference tokom rada. Pri tome, treba voditi računa da otpornik utiče na vrednost referentnog napona, usled postojanja internog otpornika od 32K na AREF pinu. Ova dva otpornika čine naponski razdelnik, pa npr. eksterni napon od 2.5V u kombinaciji sa naponskim razdelikom daje referencu od $2.5V \cdot \frac{32K}{32K+5K} \approx 2.2V$ na AREF pinu.

Pogledati još:
analogRead()

Nazad na jezičke reference

analogRead()

Opis:

Funkcija očitava vrednost izabranog analognog pina. Arduino UNO ploča sadrži 6 kanala (Mini i Nano 8, a Mega čak 16), koji se dovode na ulaz 10-bitnog A/D konvertora. To znači da se ulani naponi u opsegu 0-5V predstavljaju celim brojevima od 0 do $2^{10}-1 = 1023$. Prema tome, rezolucija iznosi 5V / 1024 jedinica = 4.9mV / jedinici. Ulazni opseg i rezolucija mogu biti promenjeni upotrebom funkcije **analogReference()**.

Tipično trajanje konverzije, odnosno očitavanja analognog ulaza je oko 100 mikrosekundi, što znači da je moguće postići maksimalnu frekvenciju od oko 10000 očitavanja u sekundi.

Sintaksa:

analogRead(pin)

Parametri:

- pin: broj analognog ulaznog pina (0 do 5 na većini ploča, 0 do 7 na Mini i Nano, 0 do 15 na Mega)

Povratna vrednost:

int (0 do 1023)

Primer:

```
int analogPin = 3;  // srednji izvod potenciometra vezanog izmedju Vcc i GND
                    // povezan je na analogni pin 3
int val = 0;        // promenljiva u koju se smesta ocitana vrednost

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

void loop()
{
    val = analogRead(analogPin);    // ocitava analogni ulaz pin
    Serial.println(val);            // salje ocitanu vrednost preko serijskog
       porta
}
```

Pogledati još:
analogReference()

Nazad na jezičke reference

analogWrite() - PWM

Opis:

Funkcija zapravo ne postavlja analognu vrednost na izlazni pin, nego uključuje PWM signal na njemu (PWM - engl. Pulse Width Modulation = Impulsno-Širinska Modulacija). Obično se koristi za kontrolu intenziteta svetljenja LED diode, ili za kontrolu brzine DC motora. Nakon poziva **analogWrite()**, na pinu će biti generisana stabilna povorka pravouganih impulsa zadatog faktora ispune, sve do sledećeg poziva neke od funkcija koje služe za upis ili čitanje stanja na istom pinu (npr. **analogWrite**, **digitalRead()**, ili **digitalWrite()**). Frekvencija PWM signala na većini pinova je oko 490Hz. Na UNO i sličnim pločama, frekvencija PWM na pinovima 5 i 6 je dvostruko veća, oko 980Hz. Na većini Arduino ploča (onima sa ATMega 168 ili ATMega 328), ova funkcija radi na pinovima 3, 5, 6, 9, 10 i 11.

Nije neophodno pozvati funkciju **pinMode()** pre poziva funkcije **analogWrite()**.

Funkcija **analogWrite()** nema nikakve veze sa analognim pinovima, niti sa **analogRead()** funkcijom.

Sintaksa:

analogWrite(pin, value)

Parametri:

- pin: pin koji se koristi za PWM izlaz
- value: vrednost koja određuje faktor ispune, između 0 (uvek isključeno) i 255 (uvek uključeno)

Povratna vrednost:

Nema.

Napomene i poznati problemi:

PWM izlazi generisani na pinovima 5 i 6 će imati nešto veći faktor ispune od očekivanog. To se dešava usled interakcije između funkcija **millis()** i **delay()**, koje dele isti interni tajmer. Ovo je pogotovo uočljivo pri manjim faktorima ispune (npr. 0-10) i može da uzrokuje da vrednost 0 nikada ne isključuje izlaze na pinovima 5 i 6 u potpunosti.

Primer:

```
// program podesava LED diodu, proporcionalno polozaju potenciometra
int ledPin = 9;      // LED je spojena na digitalni pin 9
int analogPin = 3;   // potenciometar je spojen na analogni pin 3
int val = 0;         // promenljiva koja pamti ocitanu vrednost

void setup()
{
}
void loop()
{
  val = analogRead(analogPin);  // citanje ulaznog pina
  analogWrite(ledPin, val / 4); // analogRead vraca vrednost od 0 to 1023,
                                // a analogWrite uzima vrednost od 0 to 255
}
```

Pogledati još:
analogRead()

Nazad na jezičke reference

millis()

Opis:

Vraća broj milisekundi od kako je Arduino počeo da izvršava program. Ova vrednost će prekoračiti opseg (odnosno vratiti se na nulu) nakon približno 50 dana.

Parametri:

Nema.

Povratna vrednost:

Broj milisekundi od početka programa (unsigned long)

Primer:

```
unsigned long time;

void setup(){
    Serial.begin(9600);
}
void loop(){
    Serial.print("Vreme: ");
    time = millis() / 1000;
    //prikazuje broj sekundi od pocetka programa
    Serial.println(time);
    delay(1000);
}
```

Savet:

Pošto funkcija vraća vrednost tipa **unsigned long**, može doći do greške pri kombinovanju sa drugim tipovima promenljivih kao što je **int**.

Pogledati još:
micros()
delay()
delayMicroseconds()

Nazad na jezičke reference

micros()

Opis:

Vraća broj mikrosekundi od kako je Arduino počeo da izvršava program. Ova vrednost će prekoračiti opseg (odnosno vratiti se na nulu) nakon približno 70 minuta. Na Arduino pločama koje rade na 16MHz (npr. Duemilanove i Nano), ova funkcija ima rezoluciju od 4 mikrosekunde (odnosno, povratna vrednost je uvek deljiva sa 4). Na pločama koje rade na 8Mhz (npr. LilyPad), ova funkcija ima rezoluciju od 8 mikrosekundi.

Parametri:

Nema.

Povratna vrednost:

Broj mikrosekundi od početka programa (unsigned long)

Primer:

```
unsigned long time;

void setup(){
    Serial.begin(9600);
}
void loop(){
    Serial.print("Vreme: ");
    time = micros();
    //prikazuje broj mikrosekundi od pocetka programa
    Serial.println(time);
    delay(1000);
}
```

Pogledati još:
millis()
delay()
delayMicroseconds()

Nazad na jezičke reference

delay()

Opis:

Pauza u programu, čije trajanje (u milisekundama) se prosleđuje kao parametar.

Sintaksa:

delay(ms)

Parametri:

- ms: trajanje pauze u milisekundama (unsigned long)

Povratna vrednost:

Nema.

Primer:

```
int ledPin = 13;                // LED je spojena na pin 13

void setup()
{
    pinMode(ledPin, OUTPUT);    // podesava pin kao izlaz
}

void loop()
{
    digitalWrite(ledPin, HIGH); // ukljucuje LED
    delay(1000);                // ceka sekund
    digitalWrite(ledPin, LOW);  // iskljucuje LED
    delay(1000);                // ceka sekund
}
```

Prethodni primer pokazuje da je jednostavno postići treptanje LED diode korišćenjem funkcije **delay()**. Takođe, ova funkcija se često koristi za zadatke kao što je zaštita od "zvečanja" tastera. Međutim korišćenje funkcije **delay** u programu može biti skopčano sa izvesnim nedostacima. Naime, tokom trajanja ove funkcije, kontroler nije u mogućnosti da očitava senzore, obavlja matematičke operacije, postavlja stanja pinova i sl. Prema tome, poziv ove funkcije efektivno dovodi do zastoja u većini ostalih poslova koje kontroler obavlja. Dole navedeni primer prikazuje alternativni način kontrolisanja vremena, gde se umesto funkcije **delay()** kontinualno vrši očitavanje vremena funkcijom **millis()**.

```
const int ledPin =  13;      // LED je na pinu 13

int ledState = LOW;          // promenljiva koja pamti stanje LED diode
unsigned long t0 = 0;            // promenljiva koja pamti pocetak vremnskog
    intervala
long interval = 1000;        // perioda treptanja (u milisekundama)

void setup() {
    // inicijalizacija izlaznog pina:
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    unsigned long t = millis();

    if(t - t0 > interval)
    {
        // pamti pocetni trenutak novog intervala:
        t0 = t;

        // promena stanja LED diode:
        if (ledState == LOW)
            ledState = HIGH;
        else
            ledState = LOW;
        digitalWrite(ledPin, ledState);
    }
}
```

Pojedine stvari se ipak dešavaju na ATMega čipu u paraleli sa izvršenjem **delay()** funkcije, pošto ova funkcija ne vrši zabranu prekida. Serijska komunikacija koja stiže preko RX pina se smešta u prijemni bafer, PWM (**analogWrite**) impulsi se generišu, a prekidi se redovno opslužuju.

Pogledati još:
millis()
micros()
delayMicroseconds()

Nazad na jezičke reference

delayMicroseconds()

Opis:

Pauza u programu, čije trajanje (u mikrosekundama) se prosleđuje kao parametar.
Trenutno, najveća vrednost parametra za koju se dobija precizno kašnjenje je 16383. Ovo će verovatno biti podložno izmenama u budućim verzijama Arduina. Za kašnjenja duža od nekoliko hiljada mikrosekundi, preporučljivo je koristiti funkciju **delay()**.

Sintaksa:

delayMicroseconds(ms)

Parametri:

- us: trajanje pauze u mikrosekundama (unsigned long)

Povratna vrednost:

Nema.

Primer:
Sledeći program na pinu 8 generiše povorku impulsa periode $100\mu s$ (odnosno frekvencije 10kHz).

```
int outPin = 8;

void setup()
{
    pinMode(outPin, OUTPUT);     // digitalni pin 8 je izlaz
}

void loop()
{
    digitalWrite(outPin, HIGH); // postavlja pin na 1
    delayMicroseconds(50);      // pauzira 50 mikrosekundi
    digitalWrite(outPin, LOW);  // postavlja pin na 0
    delayMicroseconds(50);      // pauzira 50 mikrosekundi
}
```

Ova funkcija radi vrlo precizno za vrednosti intervala duže od 3 mikrosekunde, ali se za kraće intervale ne može garantovati preciznost. Takođe, od verzije Arduino 0018, **delayMicroseconds()** više ne vrši zabranu prekida.

Pogledati još:
millis()
micros()
delay()
Nazad na jezičke reference

min()

Opis:

Funkcija vraća manji od dva ulazna parametra.

Sintaksa:

min (x,y)

Parametri:

- x: broj proizvoljnog tipa
- y: broj proizvoljnog tipa

Povratna vrednost:

Manji od dva broja.

Primer:

sensVal = min(sensVal, 100); // obezbedjuje da senSval nikada ne predje vrednost 100

Napomena:

Iako na prvi pogled može delovati nelogično, **max()** se često koristi za ograničavanje donjeg kraja opsega promenljive, a **min()** za ograničavanje gornjeg kraja opsega.

Upozorenje:

Zbog načina na koji je funkcija **min** realizovana, treba izbegavati istovremeno korišćenje drugih funkcija nad parametrima u zagradi, pošto na taj način može do neispravnih rezultata.

min(a++, 100); // ovakve izraze treba izbegavati

a++;
min(a, 100); // umesto toga, ostalu matematiku treba izmestiti izvan poziva funkcije

Pogledati još:
max()
constrain()

Nazad na jezičke reference

max()

Opis:

Funkcija vraća veći od dva ulazna parametra.

Sintaksa:

max (x,y)

Parametri:

- x: broj proizvoljnog tipa
- y: broj proizvoljnog tipa

Povratna vrednost:

Veći od dva broja.

Primer:

sensVal = max(sensVal, 20); // obezbedjuje da senSval nikada ne bude manje od 20

Napomena:

Iako na prvi pogled može delovati nelogično, **max()** se često koristi za ograničavanje donjeg kraja opsega promenljive, a **min()** za ograničavanje gornjeg kraja opsega.

Upozorenje:

Zbog načina na koji je funkcija **max** realizovana, treba izbegavati istovremeno korišćenje drugih funkcija nad parametrima u zagradi, pošto na taj način može do neispravnih rezultata.

max(a–, 0); // ovakve izraze treba izbegavati

a–;
max(a, 100); // umesto toga, ostalu matematiku treba izmestiti izvan poziva funkcije

Pogledati još:
min()
constrain()

Nazad na jezičke reference

abs()

Opis:

Funkcija vraća apsolutnu vrednost ulaznog parametra.

Sintaksa:

abs (x)

Parametri:

- x: broj proizvoljnog tipa

Povratna vrednost:

x, ako je $x \geq 0$
$-x$, ako je $x < 0$

Upozorenje:

Zbog načina na koji je funkcija **abs** realizovana, treba izbegavati istovremeno korišćenje drugih funkcija nad parametrima u zagradi, pošto na taj način može do neispravnih rezultata.

```
abs(a++); // ovakve izraze treba izbegavati

a++;
abs(a); // umesto toga, ostalu matematiku treba izmestiti izvan poziva funkcije
```

Nazad na jezičke reference

constrain()

Opis:

Funkcija ograničava broj unutar zadatog intervala.

Sintaksa:

constrain (x, a, b)

Parametri:

- x: broj proizvoljnog tipa koji je potrebno ograničiti
- a: donji kraj (minimalna vrednost) opsega
- b: gornji kraj (maksimalna vrednost) opsega

Povratna vrednost:

x, ako je $x \in [a, b]$
a, ako je $x < a$
b, ako je $x > b$

Primer:

sensVal = constrain(sensVal, 10, 150); // ogranicava opseg vrednosti sensVal izmedju 10 i 150

Pogledati još:
min()
max()

Nazad na jezičke reference

map()

Opis:

Funkcija koristi linearnu funkciju koja preslikava broj iz jednog intervala u drugi. To znači da se vrednost parametra **fromLow** preslikava u **toLow**, vrednost parametra **fromHigh** u **toHigh**, a ostale vrednosti se preslikavaju linearno.

Ova funkcija ne postavlja ograničenje da vrednost koja se preslikava mora biti unutar intervala, što u pojedinim slučajevima može biti korisno i poželjno. Ako se ipak želi postaviti takvo ograničenje, pre poziva **map()** potrebno je prvo pozvati **constrain()**.

Takođe, nije neophodno da "donje granice" budu manje od "gornjih granica", pa prema tome ova funkcija može biti korišćena za obrtanje opsega brojeva, npr:

y = map(x, 1, 50, 50, 1);

Funkcija takođe radi korektno i sa negativnim brojevima, prema tome validan je i sledeći primer:

y = map(x, 1, 50, 50, -100);

Funkcija **map** koristi celobrojnu matematiku, pa stoga ignoriše eventualne razlomljene vrednosti (tj. sve što se nalazi iza decimalnog zareza).

Sintaksa:

map (value, fromLow, fromHigh, toLow, toHigh)

Parametri:

- value: broj koji se preslikava
- fromLow: "donji" kraj polaznog intervala
- fromHigh: "gornji" kraj polaznog intervala
- toLow: "donji" kraj odredišnog intervala
- toHigh: "gornji" kraj odredišnog intervala

Povratna vrednost:

Ulazna vrednost preslikana u odredišni interval.

Primer:

```
/* mapiranje analognog ulaza u 8-bitni opseg (0 do 255) */
void setup() {}

void loop()
{
    int val = analogRead(0);
    val = map(val, 0, 1023, 0, 255);
    analogWrite(9, val);
}
```

Dodatak - implementacija funkcije:

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Pogledati još:
constrain()

Nazad na jezičke reference

pow()

Opis:

Funkcija računa vrednost broja dignutu na zadati stepen. Može biti korišćena i za dizanje broja na stepen koji nije ceo broj, što je korisno kada je potrebno eksponencijalno mapiranje vrednosti.

Sintaksa:

pow(base, exponent)

Parametri:

- base: broj (float)
- exponent: stepen na koji se diže vrednost base (float)

Povratna vrednost:

$base^{exponent}$ (float)

Pogledati još:
sqrt()
float

Nazad na jezičke reference

sqrt()

Opis:

Funkcija vraća kvadratni koren broja.

Sintaksa:

sqrt(x)

Parametri:

- x: broj proizvoljnog tipa

Povratna vrednost:

$\sqrt{x}$ (float)

Pogledati još:
pow()

Nazad na jezičke reference

sin()

Opis:

Funkcija vraća sinus ugla (u radijanima). Rezultat je u opsegu od -1 do 1.

Sintaksa:

sin(x)

Parametri:

- x: broj (float)

Povratna vrednost:

Sinus ugla (float).

Pogledati još:
cos()
tan()
float

Nazad na jezičke reference

cos()

Opis:

Funkcija vraća kosinus ugla (u radijanima). Rezultat je u opsegu od -1 do 1.

Sintaksa:

cos(x)

Parametri:

- x: broj (float)

Povratna vrednost:

Kosinus ugla (float).

Pogledati još:
sin()
tan()
float

Nazad na jezičke reference

tan()

Opis:

Funkcija vraća tangens ugla (u radijanima). Rezultat je u opsegu od $-\inf$ do $\inf$.

Sintaksa:

tan(x)

Parametri:

- x: broj (float)

Povratna vrednost:

Tangens ugla (float).

Pogledati još:
sin()
cos()
float

Nazad na jezičke reference

randomSeed()

Opis:

Funkcija randomSeed() inicijalizuje generator pseudo-slučajnih brojeva, čime se postiže da generisanje krene od proizvoljnog člana u okviru pseudoslučajne sekvence. Ova sekvenca, iako vrlo dugačka, uvek je ista.

Ako je važno da se sekvenca vrednosti generisana funkcijom **random()** razlikuje tokom različitih slučajeva izvršenja programa, preporučljivo je inicijalizovati generator veličinom koja je stvarno slučajna, kao što je stanje na nepovezanom analognom ulazu. Međutim, nekada može biti koristan i suprotan slučaj, a to je da se pseudo-slučajna sekvenca uvek ponavlja na isti način. U tom slučaju, potrebno je pozvati **randomSeed()** sa fiksnim parametrom, pre početka generisanja sekvence.

Sintaksa:

randomSeed(seed)

Parametri:

- seed: "seme" koje se koristi za inicijalizaciju generatora (int ili long)

Povratna vrednost:

Nema.

Primer:

```
long randNumber;

void setup(){
    Serial.begin(9600);
    randomSeed(analogRead(0));
}

void loop(){
    randNumber = random(300);
    Serial.println(randNumber);

    delay(50);
}
```

Pogledati još:
random()

Nazad na jezičke reference

random()

Opis:

Funkcija generiše pseudo-slučajne brojeve.

Sintaksa:

random(max)
random(min, max)

Parametri:

- min - donja granica slučajne vrednosti (opciono)
- max - gornja granica slučajne vrednosti (obavezno)

Povratna vrednost:

Slučajan broj između min i max-1 (long).

Napomena:

Ako je važno da se sekvenca vrednosti generisana funkcijom **random()** razlikuje tokom različitih slučajeva izvršenja programa, preporučljivo je inicijalizovati generator veličinom koja je stvarno slučajna, kao što je stanje na nepovezanom analognom ulazu. Međutim, nekada može biti koristan i suprotan slučaj, a to je da se pseudo-slučajna sekvenca uvek ponavlja na isti način. U tom slučaju, potrebno je pozvati **randomSeed()** sa fiksnim parametrom, pre početka generisanja sekvence.

Primer:

```
void setup(){
    Serial.begin(9600);
    // ako je analogni ulaz 0 nepovezan, analogni sum ce uzrokovati
    // da randomSeed() generise razlicite sekvence pri svakom izvrsenju
       programa
    randomSeed(analogRead(0));
}

void loop() {
    Serial.println(random(300));    // prikazuje slucajan broj od 0 do 299
    Serial.println(random(10, 20)); // prikazuje slucajan broj od 10 do 19

    delay(50);
}
```

Pogledati još:
randomSeed()
Nazad na jezičke reference

Klasa Serial

Klasa **Serial** se koristi za komunikaciju između Arduino ploče i računara, ili drugih uređaja. Sve Arduino ploče imaju bar jedan serijski port, koji je kontrolisan od strane hardverskog modula poznatog kao UART (ili USART). Komunikacija se obavlja preko digitalnih pinova 0 (RX) i 1 (TX) na Arduino, odnosno preko USB porta na računaru. Korišćenjem serijskog monitora ugrađenog u Arduino IDE, moguće je na jednostavan način ostvariti komunikaciju sa Arduino pločom.

Funkcije:

- available()
- begin()
- end()
- find()
- findUntil()
- flush()
- parseFloat()
- parseInt()
- peek()
- print()
- println()
- read()
- readBytes()
- readBytesUntil()
- setTimeout()
- write()
- serialEvent()

Nazad na jezičke reference

Serial.available()

Opis:

Funkcija vraća broj bajta (karaktera) primljenih preko serijskog porta. Ovo su podaci koji su već stigli i smešteni u serijski bafer, čiji kapacitet je 64 bajta.

Sintaksa:

Serial.available()

Parametri:

Nema.

Povratna vrednost:

Broj bajta spremnih za čitavanje, smeštenih u serijski bafer.

Primer:

```
char incomingByte = 0;  // ovde se smesta primljeni podatak

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

void loop() {

    // podatak se ocitava tek kada je nesto primljeno:
    if (Serial.available() > 0) {
        // ocitavanje primljenog bajta:
        incomingByte = Serial.read();

        // slanje nazad:
        Serial.print("Stigao je karakter: ");
        Serial.println(incomingByte);
    }
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.begin()

Opis:

Podešava brzinu serijske komunikacije u bitima/sekundi (engl. Baud Rate). Standardne brzine za komunikaciju s računarom su: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, ili 115200. Međutim, moguće je podesiti i druge brzine, za povezivanje sa uređajima koji koriste nestandardnu brzinu komunikacije.

Opcioni drugi argument može se koristiti za konfigurisanje broja bita podataka, bita pariteta i stop bita. Ako nema ovog argumenta, podrazumeva se 8 bita podatka, 1 stop bit, bez korišćenja bita pariteta.

Sintaksa:

Serial.begin(speed)
Serial.begin(speed, config)

Parametri:

- speed: brzina u bitima/sekundi (baud rate) - long
- config: podešavanje broja bita za podatak, stop bita i bita pariteta. Validne vrednosti su:
  - SERIAL_5N1
  - SERIAL_6N1
  - SERIAL_7N1
  - SERIAL_8N1 (podrazumevana vrednost)
  - SERIAL_5N2
  - SERIAL_6N2
  - SERIAL_7N2
  - SERIAL_8N2
  - SERIAL_5E1
  - SERIAL_6E1
  - SERIAL_7E1
  - SERIAL_8E1
  - SERIAL_5E2
  - SERIAL_6E2
  - SERIAL_7E2
  - SERIAL_8E2
  - SERIAL_5O1

- SERIAL_6O1
- SERIAL_7O1
- SERIAL_8O1
- SERIAL_5O2
- SERIAL_6O2
- SERIAL_7O2
- SERIAL_8O2

Povratna vrednost:

Nema.

Primer:

```
void setup()
{
    Serial.begin(9600); // otvara serijski port, brzina = 9600 bita/s
}

void loop()
{
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.end()

Opis:

Zatvara serijski port i omogućava da pinovi RX i TX budu korišćeni kao obični digitalni pinovi. Da bi serijska komunikacija bila ponovo omogućena, potrebno je pozvati **Serial.begin()**.

Sintaksa:

Serial.end()

Parametri:

Nema.

Povratna vrednost:

Nema.

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.find()

Opis:

Serial.find() čita podatke iz serijskog bafera, dok ne pronađe traženi string, ili dok ne istekne vreme. Ako je string pronađen, vraća **true**, a ako je isteklo vreme, vraća **false**.

Sintaksa:

Serial.find(target)

Parametri:

- target: string koji se traži (char[])

Povratna vrednost:

boolean

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.findUntil()

Opis:

Serial.findUntil() čita podatke iz serijskog bafera, dok ne pronađe traženi string, ili dok ne pronađe string-terminator. Ako je string pronađen, vraća **true**, a ako je isteklo vreme, vraća **false**.

Sintaksa:

Serial.findUntil(target, terminal)

Parametri:

- target: string koji se traži (char[])
- terminal: string-terminator (char[])

Povratna vrednost:

boolean

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.flush()

Opis:

Funkcija čeka dok se završi slanje podataka preko serijskog porta.

Sintaksa:

Serial.flush()

Parametri:

Nema.

Povratna vrednost:

Nema.

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.parseFloat()

Opis:

Funkcija traži sledeći validan realan broj u serijskom baferu. Karakteri koji nisu cifre, decimalna tačka ili znak '-' se ignorišu. Izvršenje funkcije se prekida nailaskom na prvi "neregularan" karakter.

Sintaksa:

Serial.parseFloat()

Parametri:

Nema.

Povratna vrednost:

Sledeći validan realan broj (tipa **float**).

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.parseInt()

Opis:

Funkcija traži sledeći validan ceo broj u nadolazećem nizu karaktera. Ako ne bude pronađen validan broj u roku od jedne sekunde (što se može podesiti funkcijom **Serial.setTimeout()**), funkcija vraća 0.

Sintaksa:

Serial.parseInt()

Parametri:

Nema.

Povratna vrednost:

Sledeći validan ceo broj (tipa **int**).

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.peek()

Opis:

Funkcija vraća sledeći karakter u prijemnom serijskom baferu, bez njegovog uklanjanja iz bafera (za razliku od funkcije **Serial.read()**). Posledično, sukcesivni pozivi **Serial.peek()** će vraćati isti karakter.

Sintaksa:

Serial.peek()

Parametri:

Nema.

Povratna vrednost:

Prvi bajt u serijskom baferu (-1 ako je bafer prazan). Povratna vrednost je tipa **int**.

Nazad na jezičke reference

Serial.print() i Serial.println()

Opis:

Funkcija šalje preko serijskog porta (odnosno "štampa" na serijskom terminalu), podatke u obliku ASCII teksta, koji olakšava čitanje humanoidnim organizmima čija hemija se zasniva na ugljeniku. Ova komanda može da ima više različitih formi. Celi brojevi se prikazuju slanjem ASCII karaktera za svaku cifru. Realni brojevi (float) se ispisuju na sličan način, uz prikazivanje još dve cifre iza decimalnog zareza. Bajti se šalju kao pojedinačni karakteri. Karakteri se šalju takvi kakvi jesu, u formi ASCII koda. Funkcija **println** ima isti format kao **print**, ali uz to prelazi u novi red nakon slanja podatka, šaljući specijalne karaktere ASCII 13 (CR) i ASCII 10 (LF). Primeri:

- Serial.print(78); //ispisuje "78"
- Serial.print(1.23456); //ispiuje "1.23"
- Serial.print('N'); //ispisuje "N"
- Serial.print("Hello world."); //ispisuje "Hello world."

Opcioni drugi parametar specificira bazu (format) koji će biti korišćen pri ispisu. Dozvoljene vrednosti su BIN (binarni, sa osnovom 2), OCT (oktalni, sa osnovom 8), DEC (decimalni, sa osnovom 10) i HEX (heksadecimalni, sa osnovom 16). Za brojeve u pokretnom zarezu, ovaj parametar specificira broj decimala koji se prikazuje. Primeri:

- Serial.print(78, BIN); //ispisuje "1001110"
- Serial.print(78, OCT); //ispisuje "116"
- Serial.print(78, DEC); //ispisuje "78"
- Serial.print(78, HEX); //ispisuje "4E"
- Serial.println(1.23456, 0); //ispisuje "1"
- Serial.println(1.23456, 2); //ispisuje "1.23"
- Serial.println(1.23456, 4); //ispisuje "1.2346"

U cilju štednje operativne memorije, moguće je slati stringove koji su smešteni u programsku (FLASH) memoriju, tako što se string obuhvati sa **F()**. Primer:

- Serial.print(F(Hello World));

Za slanje pojedinačnih bajta, koristi se funkcija **Serial.write()**.

Sintaksa:

Serial.print(val)
Serial.print(val, format)

Parametri:

- val: vrednost koja se prikazuje - podatak bilo kog tipa
- format: za celobrojne tipove, označava osnovu sistema, a za tipove u pokretnom zarezu ozmačava broj decimalnih mesta

Povratna vrednost:
size_t (long): **Serial.print()** vraća broj poslatih bajta; očitavanje ove vrednosti je opciono.

Primer:

```
// koriscenjem for petlje, ispisuju se brojevi u raznim formatima
char x = 0;                         // celobrojna promenljiva

void setup() {
  Serial.begin(9600);               // otvara serijski port na 9600 bita/s
}

void loop() {
    // ispis oznaka
    Serial.print("CHAR");           // ispisuje oznaku
    Serial.print("\t");             // ispisuje tab

    Serial.print("DEC");
    Serial.print("\t");

    Serial.print("HEX");
    Serial.print("\t");

    Serial.print("OCT");
    Serial.print("\t");             // prelazi u novi red ("println")

    Serial.println("BIN");

    for(x=33; x<127; x++){          // ispisuje vidljivi deo ASCII tabele
        Serial.print(x);            // ispisuje ASCII-kodiran broj (kao "DEC")
        Serial.print("\t");         // ispisuje tab

        Serial.print(x, DEC);       // ispisuje ASCII-kodiran decimalni broj
        Serial.print("\t");         // ispisuje tab

        Serial.print(x, HEX);       // ispisuje ASCII-kodiran heksadecimalni broj
        Serial.print("\t");         // ispisuje tab

        Serial.print(x, OCT);       // ispisuje ASCII-kodiran oktalni broj
        Serial.print("\t");         // ispisuje tab

        Serial.println(x, BIN);     // ispisuje ASCII-kodiran binarni broj
                                    // i prelazi u novi red ("println")
    }
    while (true);                   //vrti se u beskonacnoj petlji
}
```

Napomena:

Od verzije 1.0, serijsko slanje se obavlja asinhrono; povratak iz funkcije **Serial.print()** se dešava pre nego što se karakteri pošalju.

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.read()

Opis:

Funkcija očitava prvi od raspoloživih primljenih bajta iz serijskog bafera.

Sintaksa:

Serial.read()

Parametri:

Nema.

Povratna vrednost:

Prvi raspoloživi bajt iz serijskog bafera. Ako je bafer prazan, vraća -1. Povratna vrednost je tipa **int**.

Primer:

```
char incomingByte = 0;  // ovde se smesta primljeni podatak

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

void loop() {

    // podatak se ocitava tek kada je nesto primljeno:
    if (Serial.available() > 0) {
        // ocitavanje primljenog bajta:
        incomingByte = Serial.read();

        // slanje nazad:
        Serial.print("Stigao je karakter: ");
        Serial.println(incomingByte);
    }
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.readBytes()

Opis:

Funkcija **readBytes()** učitava zadati broj bajta primljenih preko serijskog porta u zadati niz (bafer). Funkcija se završava po očitavanju zadatog broja bajta, ili po isteku vremena (videti Serial.setTimeout()).

Sintaksa:

Serial.readBytes(buffer, length)

Parametri:

- buffer: niz (char[] ili byte[]) u koji se smeštaju podaci
- length: broj bajta koji se čitaju (int)

Povratna vrednost:

Broj očitanih bajta (0 znači da ništa nije primljeno). Povratna vrednost je tipa **byte**.

Primer:

```
char string[64];    //bafer u koji se smesta primljeni string
int duzina;

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

void loop()
{
    while(!Serial.available());             // ceka da primi nesto
    delay(100);                             // pauza, dok se okonca prijem
    duzina = Serial.available();            // odredjiivanje duzine stringa

    Serial.readBytes(string, duzina);       // ucitavanje stringa u bafer
    string[duzina] = 0;                     // postavljanje terminatora
    Serial.print("Primljen je string: ");   // ispis primljenog stringa
    Serial.println(string);
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.readBytesUntil()

Opis:

Funkcija **readBytes()** učitava zadati broj bajta primljenih preko serijskog porta u zadati niz (bafer). Funkcija se završava po nailasku na zadati karakter (terminator), po očitavanju zadatog broja bajta, ili po isteku vremena (videti Serial.setTimeout()).

Sintaksa:

Serial.readBytesUntil(character, buffer, length)

Parametri:

- character: karakter po čijem prijemu se završava čitanje (terminator)
- buffer: niz (char[] ili byte[]) u koji se smeštaju podaci
- length: broj bajta koji se čitaju (int)

Povratna vrednost:

Broj očitanih bajta (0 znači da ništa nije primljeno). Povratna vrednost je tipa **byte**.

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.setTimeout()

Opis:

Funkcija **Serial.setTimeout()** podešava dužinu vremenskog intervala (u milisekundama), tokom kojeg se čekaju podaci pri pozivu funkcije **Serial.readBytesUntil()** ili **Serial.readBytes()**. Podrazumevana vrednost je 1000 milisekundi.

Sintaksa:

Serial.setTimeout(time)

Parametri:

- time: maksimalno trajanje prijema u millisekundama (long).

Povratna vrednost:

Nema.

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.write()

Opis:

Funkcija **write()** šalje binarne podatke preko serijskog porta. Ovi podaci se šalju kao jedan bajt, ili niz bajta. Ako je cilj poslati broj ne u binarnoj formi, nego u obliku niza karaktera koji predstavljaju njegove cifre, potrebno je umesto **Serial.write()** pozvati funkciju Serial.print()

Sintaksa:

Serial.write(val)
Serial.write(str)
Serial.write(buf, len)

Parametri:

- val: vrednost koja se šalje kao pojedinačan bajt
- str: string koji se šalje kao niz bajta
- buf: niz (bafer) koji se serijski šalje
- len: dužina niza

Povratna vrednost:

Funkcija vraća broj poslatih bajta. Povratna vrednost je tipa **byte**.

Primer:

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

void loop(){
    Serial.write(45); // salje bajt vrednosti 45

    int bytesSent = Serial.write("hello"); //salje string "hello" i vraca
        duzinu stringa
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Serial.serialEvent()

Opis:

Ukoliko je implementirana, ova funkcija se poziva po prijemu podatka preko serijskog porta. Učitavanje podatka se potom vrši nekom od funkcija za čitanje, npr. **Serial.read()**.

Sintaksa:

```
void serialEvent(){
 //blok naredbi
}
```

Pogledati još:
Klasa Serial

Nazad na jezičke reference

Biblioteka LiquidCrystal

Ova biblioteka omogućava upravljanje dislejima sa tečnim kristalom (LCD) baziranim na Hitachi HD44780 (ili kompatibilnom) čipu, koji je prisutan kod većine tekstualnih LCD displeja. Biblioteka radi ili u 4-bitnom ili u 8-bitnom režimu komunikacije, uz dodatne kontrolne signale (rs, enable i opciono rw).

Funkcije:

- Klasa LiquidCrystal
- begin()
- clear()
- home()
- setCursor()
- write()
- print()
- cursor() i noCursor()
- blink() i noBlink()
- display() i noDisplay()
- scrollDisplayLeft() i scrollDisplayRight()
- autoscroll() i noAutoscroll()
- leftToRight() i rightToLeft()
- createChar()

Nazad na jezičke reference

Klasa LiquidCrystal

Opis:

Klasa LiquidCrystal sadrži potrebne funkcije za upravljanje LCD displejima. Displej se deklariše kao objekat koji je instanca ove klase. Displej može biti korišćen u 4-bitnom ili 8-bitnom režimu komunikacije. U prvom slučaju, pri instanciranju se izostavljaju brojevi pinova za linije d0..d3 i na konektoru displeja ti pinovi ostaju nepovezani. Pin RW može biti vezan na masu, umesto da bude vezan na pin Arduina; ako je to slučaj, pin za RW se takođe izostavlja pri nabrajanju parametara.

Sintaksa:

LiquidCrystal lcd(rs, enable, d4, d5, d6, d7)
LiquidCrystal lcd(rs, rw, enable, d4, d5, d6, d7)
LiquidCrystal lcd(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)
LiquidCrystal lcd(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)

Parametri:

- rs: broj Arduino pina povezanog na RS pin LCD displeja
- rw: broj Arduino pina povezanog na RW pin LCD displeja (opciono)
- enable: broj Arduino pina povezanog na ENABLE pin LCD displeja
- d0,d1,d2,d3,d4,d5,d6,d7: brojevi Arduino pinova koji su povezani na odgovarajuće linije za prenos podataka na displeju. d0,d1,d2 i d3 su opcioni; ako su izostavljeni, LCD će biti kontrolisan korišćenjem 4 linije za podatke (d4,d5,d6 i d7).

Primer:

```
#include <LiquidCrystal.h>
// zadavanje pinova koji se koriste za komunikaciju s displejom
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void setup()
{
    lcd.begin(16,2);
    lcd.print("Hello, world!");
}

void loop() {}
```

Pogledati još:
Biblioteka LiquidCrystal

Nazad na jezičke reference

begin()

Opis:

Funkcija inicijalizuje hardverski interfejs LCD displeja i specificira dimenzije (širinu i visinu) displeja u karakterima. Ova funkcija mora biti pozvana pre zvanja bilo koje druge funkcije za rad sa LCD displejem.

Sintaksa:

lcd.begin(cols, rows)

Parametri:

- lcd: instanca klase LiquidCrystal
- cols: broj kolona displeja
- rows: broj vrsta displeja

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

clear()

Opis:

Funkcija briše sadržaj LCD displeja i vraća kursor u gornji levi ugao.

Sintaksa:

lcd.clear()

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

home()

Opis:

Funkcija pozicionira kursor u gornji levi ugao LCD displeja. Za razliku od funkcije **clear()**, ova funkcija ne utiče na tekst koji je trenutno ispisan na displeju.

Sintaksa:

lcd.home()

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

setCursor()

Opis:

Funkcija pozicionira kursor na zadatu poziciju.

Sintaksa:

lcd.setCursor(col, row)

Parametri:

- col: indeks kolone u koju se postavlja kursor (indeksi počinju od 0)
- row: indeks vrste u koju se postavlja kursor (indeksi počinju od 0)

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

write()

Opis:

Funkcija ispisuje zadati karakter na trenutnoj poziciji kursora i pomera kursor za jedno mesto u desno.

Sintaksa:

lcd.write(data)

Parametri:

- lcd: instanca klase LiquidCrystal
- data: karakter koji se ispisuje na displeju

Povratna vrednost:

Funkcija vraća broj ispisanih karaktera, ali je čitanje te vrednosti opciono.

Primer:

```
#include <LiquidCrystal.h>
// zadavanje pinova koji se koriste za komunikaciju s displejom
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void setup()
{
  Serial.begin(9600);
  lcd.begin(16, 2);
}

void loop()
{
  if (Serial.available()) {
    lcd.write(Serial.read());
  }
}
```

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

print()

Opis:

Funkcija ispisuje tekst na LCD displeju

Sintaksa:

lcd.print(data)
lcd.print(data,BASE)

Parametri:

- lcd: instanca klase LiquidCrystal
- data: tekst koji se ispisuje na displeju (char, byte, int, long ili string)
- BASE (opciono): osnova sistema u kojem se ispisuje broj - BIN za binarni, DEC za decimalni, OCT za oktalni, HEX za heksadecimalni

Povratna vrednost:

Funkcija vraća broj ispisanih karaktera, ali je čitanje te vrednosti opciono.

Primer:

```
#include <LiquidCrystal.h>
// zadavanje pinova koji se koriste za komunikaciju s displejom
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void setup()
{
    lcd.begin(16,2);
    lcd.print("Hello, world!");
}

void loop() {}
```

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

cursor() i noCursor()

Opis:

Funkcija **cursor()** uključuje, a **noCursor()** isključuje prikaz stacionarnog kursora na poziciji na kojoj će biti ispisan sledeći karakter. Kursor se ispisuje u formi donje crte (underscore).

Sintaksa:

```
lcd.cursor()
lcd.noCursor()
```

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

blink() i noBlink()

Opis:

Funkcija **blink()** uključuje, a **noBlink()** isključuje prikaz trepćućeg kursora na poziciji na kojoj će biti ispisan sledeći karakter. Kursor se ispisuje u formi pravougaonika.

Sintaksa:

```
lcd.blink()
lcd.noBlink()
```

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

display() i noDisplay()

Opis:

Funkcija **display()** uključuje displej koji je prethodno bio isključen pozivom funkcije **noDisplay()**. Ovim se tekst (i kursor) na displeju vraćaju u stanje kakvo je bilo pre njegovog isključenja.

Sintaksa:

```
lcd.display()
lcd.noDisplay()
```

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

scrollDisplayLeft() i scrollDisplayRight()

Opis:

Funkcije **scrollDisplayLeft()** i **scrollDisplayRight()** pomeraju (skroluju) sadržaj displeja za jedno mesto ulevo, odnosno udesno.

Sintaksa:

```
lcd.scrollDisplayLeft()
lcd.scrollDisplayRight()
```

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

autoscroll() i noAutoscroll()

Opis:

Funkcija **autoscroll()** uključuje, a **noAutoscroll()** isključuje automatsko pomeranje (skrolovanje) teksta na displeju. Automatsko skrolovanje podrazumeva da svaki novi karakter koji se ispisuje na displeju pomera postojeće karaktere za jedno mesto. Ako je trenutni smer ispisa teksta sleva nadesno (ovo je početna postavka), sadržaj se pomera ulevo, u suprotnom se pomera udesno. Dodatni efekat ovog načina rada je da se svaki novi karakter ispisuje na istoj poziciji (levo, odnosno desno od kursora).

Sintaksa:

```
lcd.autoscroll()
lcd.noAutoscroll()
```

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

leftToRight() i rightToLeft()

Opis:

Funkcije **leftToRight()** i **rightToLeft()** postavljaju smer ispisa karaktera na LCD displeju s leva na desno, odnosno s desna na levo.

Sintaksa:

lcd.leftToRight()
lcd.rightToLeft()

Parametri:

- lcd: instanca klase LiquidCrystal

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference

createChar()

Opis:

Ova funkcija se koristi za kreiranje proizvoljnog korisničkog karaktera. Dozvoljeno je kreiranje ukupno 8 karaktera veličine 5×8 piksela, pri čemu se karakteri numeriššu od 0 do 7. Matrica kojom se definiše karakter je predstavljena nizom 8 bajta, čijih pet najmanje značajnih bita predstavljaju stanja piksela u dgovaraju'em redu matrice. Ispis korisničkog karaktera se vrši pozivom funkcije **write()**, pri čemu se kao parametar navodi broj karaktera.

Sintaksa:

lcd.createChar(num, data)

Parametri:

- lcd: instanca klase LiquidCrystal
- num: broj korisničkog karaktera (od 0 do 7)
- data: matrica kojom se definiše izgled karaktera

Primer:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

byte srce[8] = {
    B00000,
    B01010,
    B11111,
    B11111,
    B01110,
    B00100,
    B00000,
};

void setup() {
    lcd.begin(16, 2);
    lcd.createChar(0, srce);    //definisanje simbola srca
}

void loop() {
    lcd.setCursor(0, 0);
    lcd.write(byte(0));
    delay(1000);
    lcd.setCursor(0, 0);
    lcd.write(' ');
    delay(1000);
}
```

Pogledati još:
Biblioteka LiquidCrystal
Klasa LiquidCrystal

Nazad na jezičke reference