

Uvod u digitalnu obradu signala

Potrebno predznanje

- Poznavanje programskog jezika C

Šta će biti naučeno tokom izrade vežbe

- Upoznavanje sa Texas Instruments TMS320C5535 eZdsp razvojnom pločom
- Upotreba Code Composer™ Studio programskog okruženja za razvoj programske podrške za DSP procesore
 - Rad sa projektima (otvaranje i formiranje, osvrt na bitne opcije)
 - Dodavanje postojeće i pravljenje nove datoteke sa izvornim kodom
 - Popravljanje grešaka prilikom prevođenja i uvezivanja
 - Pokretanje programa na TMS320C5535 simulatoru
 - Kontrolisano izvršavanje programa – upotreba alata za pomoć prilikom otklanjanja grešaka tokom izvršenja (Prikaz memorije, Prikaz izraza, Tačke prekida)
- Upotreba AD/DA konvertora za obradu signala u realnom vremenu na TMS320C5535 eZdsp razvojnoj ploči

Motivacija

Različite fizičke pojave za obradu, kao što su zvuk, slika ili merni podaci, se pomoću senzora, kao što su mikrofoni, kamera ili merači, pretvaraju u električne signale (struja, napon, snaga). Zatim se ti električni signali pomoću jednog spreznog elementa digitalizuju – pretvaraju u format koji se može obraditi na procesoru. Digitalni signali se obrađuju na procesoru na osnovu isprogramiranih algoritama digitalne obrade signala. Dobijeni rezultat se iz digitalnog formata opet pretvara u električni signal pomoću drugog spreznog elementa. Na kraju takav električni signal se na displeju, na primer zvučnici, ekran ili pokazivači, prikazuje u formi koja je razumljiva korisniku, koji može biti čovek ili opet mašina.

1 Fizička arhitektura

Za potrebe demonstracije i implementacije algoritama digitalne obrade signala, na laboratorijskim vežbama opisanim u ovom udžbeniku koristiće se digitalni signal procesor TMS320C5535 kompanije Texas Instruments.

1.1 Texas Instruments TMS320C5535

Procesor TMS320C5535 pripada porodici C5000 procesora za digitalnu obradu signala (DSP) kompanije *Texas Instruments*. Namenjen je aplikacijama sa malom potrošnjom energije. Temelji se na C55X arhitekturi, koja postiže visoke performanse uz vrlo malu potrošnju energije. Karakteristike TMS320C5535 procesora date su u tabeli 1.

Tabela 1 - Karakteristike TMS320C5535 procesora

Jezgro:	<ul style="list-style-type: none">• Aritmetika u nepokretnom zarezu (<i>fixed-point</i>)• Takt procesora 50-100 MHz• Izvršenje jedne ili dve instrukcije po ciklusu• Dvostruka MAC (pomnoži i saberi) jedinica• Dvostruka aritmetičko-logička jedinica (ALUs)• Tri unutrašnje magistrale za čitanje podataka i dve unutrašnje magistrale za pisanje podataka• 320KB RAM memorije, podeljene na:<ul style="list-style-type: none">○ 64KB RAM memorije sa dvostrukim pristupom (DARAM), 8 blokova od 4K x 16-Bit○ 256KB RAM memorije sa jednostrukim pristupom (SARAM), 32 bloka od 4K x 16-Bit○ 128KB ROM (4 bloka od 16K x 16-Bit)• Hardverska podrška za izračunavanje FFT transformacije
Sprega sa perifernim jedinicama:	<ul style="list-style-type: none">• Kontroler za direktan pristup memoriji (DMA)<ul style="list-style-type: none">○ 4 DMA kontrolera sa po 4 kanala• Tri 32-bitna brojača opšte namene (GP)• Dvostruka eMMC/SD sprega• UART• SPI sa podrškom za 4 uređaja• I²C magistrala• Četiri I²S magistrale• USB Port sa podrškom za USB 2.0 <i>Full Speed</i> i <i>High Speed</i> uređaje• 10-bitni ADC sa sukcesivnom aproksimacijom (SAR) sa 4 ulaza• JTAG (IEEE-1149.1)• 32 Ulazno/Izlazna prolaza opšte namene (GPIO)<ul style="list-style-type: none">○ Podrška za konfiguraciju do 20 GPIO prolaza istovremeno

Najčešća primena TMS320C5535 procesora jeste u uređajima koji ne podrazumevaju zahtevne algoritme obrade, a pritom zahtevaju nisku potrošnju energije, kao što su:

- Bežični audio uređaji (npr. bežične slušalice, mikrofoni)
- Sprega sa senzorima u industrijskim pogonima

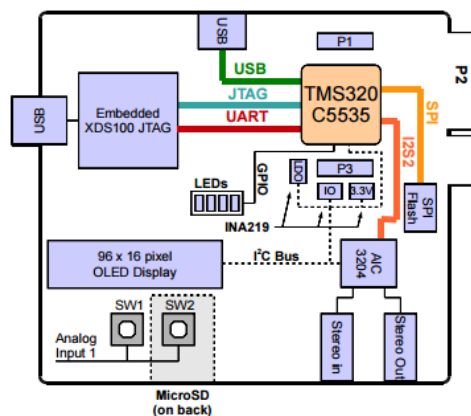
- Prenosivi medicinski uređaji
- Uređaji za biometrijsko očitavanje podataka
- Uređaji za automatizaciju pametnih kuća

	Software Compatibility	Max Freq of part (MHz)	Peripherals Differences	Memory	Co-Processor	Integrated Power Mgmt
12x12x0.8 BGA144 Pin to Pin Compatible	C5532	C55x DSP 50 MHz @ 1.05V 100 MHz @ 1.3 V		64KB		1 LDO
	C5533	C55x DSP 50 MHz @ 1.05V 100 MHz @ 1.3 V	USB	128KB		2 LDOs
	C5534	C55x DSP 50 MHz @ 1.05V 100 MHz @ 1.3 V	USB	256KB		3 LDOs
	C5535	C55x DSP 50 MHz @ 1.05V 100 MHz @ 1.3 V	USB, LCD SAR ADC	320KB	FFT	3 LDOs
10x10x0.65 BGA196 Pin to Pin Compatible	C5504	C55x DSP 100 MHz @ 1.3V 120 MHz @ 1.3 V 150 MHz @ 1.4V	USB	256KB EMIF		1 LDO
	C5505	C55x DSP 100 MHz @ 1.3V 120 MHz @ 1.3 V 150 MHz @ 1.4V	USB, LCD SAR ADC	320KB EMIF	FFT	1 LDO
	C5514	C55x DSP 100 MHz @ 1.3V 120 MHz @ 1.3 V	USB	256KB EMIF		3 LDOs
	C5515	C55x DSP 100 MHz @ 1.3V 120 MHz @ 1.3 V	USB, LCD SAR ADC	320KB EMIF	FFT	3 LDOs

Slika 1 - Uporedni prikaz odlika procesora C55x familije

1.2 TMS320C5535 eZdsp Development Kit

TMS320C5535 eZdsp Development Kit predstavlja paket alata za razvoj softvera za digitalnu obradu signala upotrebom TMS320C5535 procesora. Ovaj razvojni paket sadrži razvojnu ploču malih dimenzija, male potrošnje koja je napajana preko USB-a.



Slika 2 - TMS320C5535 eZdsp Development Kit i blok diagram

Na ploči se pored TMS320C5535 procesora nalaze:

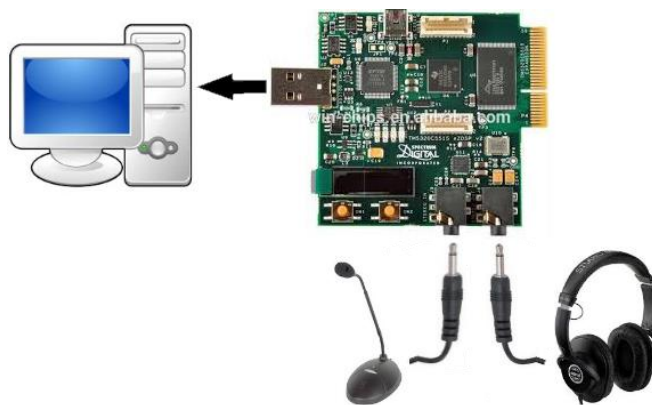
- XDS100 emulator –omogućava kontrolisano izvršenje programa (*debug*) i određene funkcije razvojnog okruženja.
- Flash memorija kapaciteta 8-MB – omogućava trajno smeštanje programskog koda

- TLV320AIC3204 programabilni audio kodek – omogućava konverziju signala iz analognog u digitalni domen i obrnuto.
- USB 2.0 port – služi za povezivanje sa računarom
- Micro SD card slot – omogućava korišćenje SD kartice kao dodatnog memorijskog prostora
- Linijski 3.5mm audio ulazni i izlazni port – za povezivanje spoljnih audio uređaja sa kodekom
- 60-pinski prolaz za povezivanje proširenja
- OLED displej rezolucije 96 x 16 piksela – omogućava prikazivanje informacija u toku izvršavanja koda
- 2 dugmeta - omogućavaju dodavanje spoljne kontrole u toku izvršavanja koda
- 4 led diode – omogućavaju kontrolnu signalizaciju u toku izvršavanja koda

Pored razvojne ploče, skup alata uključuje razvojno okruženje Code Composer™ Studio.

1.2.1 Povezivanje sa računarom

Pre početka rada sa razvojnom pločom potrebno je istu povezati sa PC računarom. Način povezivanja prikazan je na slici 4. Za povezivanje sa računarom koristi se USB 2.0 konektor. Za povezivanje spoljnih audio uređaja koriste se 3.5mm konektori. Na *stereo in* se mogu povezati različiti izvori zvuka poput mikrofona, Line-out izlaza na PC-u, telefon, mp4 player itd. Za preslušavanje izlaznog signala u toku izrade vežbi najčešće će se koristiti slušalice koje je potrebno povezati na *stereo out* konektor. Umesto slušalice moguće je povezati i *Line in* ulaz na računaru za potrebe snimanja izlaznog signala.



Slika 3 - Povezivanje TMS320C5535 eZdsp razvojne ploče

2 Programski jezik C

Svi pimeri implementacije algoritama opisanih u ovom udžbeniku dati su u programskom jeziku C, i svaka od vežbi podrazumeva kao obavezno predznanje poznavanje ovog programskog jezika. U ovom poglavlju dat je pregled osnovnih elemenata i koncepata u programskom jeziku C kao i proširenja i specifičnosti ovog jezika koja se tiču TMS320C5535 procesora. Cilj ovog poglavlja nije opis svih elemenata programskog jezika C, već pregled onih elemenata neophodnih za praćenje udžbenika i izradu zadataka. Određene osobine i pojmovi koji nisu dati u ovom poglavlju, a koji su neophodni za izradu nekog od zadataka ili razumevanje određenog primera, objašnjeni su u tekstu koji sadrži opis tog zadatka ili primera.

Primena algoritama za obradu signala najčešće podrazumeva izvršavanje tih algoritama nad nekim signalima u realnom vremenu. Kako bi se omogućilo što brže i efikasnije izvršavanje tih algoritama izdvojila se posebna klasa namenskih procesora prilagođena ovoj vrsti obrade, koji se nazivaju digitalni signal procesori. Bilo da se algoritmi izvršavaju na digitalnim signal procesorima ili drugoj vrsti namenskih procesora, ograničenja koja postavlja obrada u realnom vremenu čine programske jezike višeg ili visokog nivoa nepogodnim za njihovu implementaciju. Kako bi se postigla što veća efikasnost, u praksi se za implementaciju najčešće koriste asemblerski jezik ili programski jezik C. Iako najčešće manje efikasan po pitanju vremena izvršavanja, programski jezik C donosi određene prednosti, a to su:

- programiranje je bliže domenu problema: moguće je programirati bez znanja o resursima fizičke arhitekture. Samim tim nema grešaka usled neodgovarajuće raspodele resursa, kao što su registri, memorija i sl,
- kod je pregledan, lakše se čita i razume,
- kod je lako prenosiv: uz minimalne modifikacije i odgovarajućeg prevodioca dobija se izvršni kod za drugu platformu.

Programski jezik C definisan je međunarodnim standardom. Međutim, s obzirom da se programski jezik C može koristiti za pisanje softvera za različite platforme postoje određene karakteristike koje se razlikuju od platforme do platforme odnosno od prevodioca (kompajlera) do prevodioca, kao što su podržani tipovi i veličine tipova, određene direktive, atributi funkcija i sl.

2.1 Tipovi podataka

U tabeli 2 prikazani su svi tipovi podataka podržani od strane kompajlera za TMS320C55x arhitekturu. Prva kolona predstavlja ključnu reč kojom je tip označen, koja ujedno predstavlja i ime tipa. Druga kolona predstavlja veličinu tipa u bitima. Treća kolona predstavlja format tipa, a poslednja kolona brojevni opseg.

Tabela 2 – Tipovi podataka kod kompajlera za TMS320C55x

Tip	Veličina tipa	Format	Opseg
Char	16 bita	ASCII karakter	[-32768, 32767]
unsigned char	16 bita	ASCII karakter	[0, 65535]
Short	16 bita	Drugi komplement dvojke	[-32768, 32767]
unsigned short	16 bita	Binarni broj	[0, 65535]
Int	16 bita	Drugi komplement	[-32768, 32767]
unsigned int	16 bita	Binarni broj	[0, 65535]
Long	32 bita	Drugi komplement	[-2167483648, 2147483647]
unsigned long	32 bita	Binarni broj	[0, 4294967295]
Float	32 bita	IEEE 32-bit	[1.175494*10 ⁻³⁸ , 340282346*10 ³⁸]
Double	32 bita	IEEE 32-bit	[1.175494*10 ⁻³⁸ , 340282346*10 ³⁸]
long double	32 bita	IEEE 32-bit	[1.175494*10 ⁻³⁸ , 340282346*10 ³⁸]
Enum	16 bita	Drugi komplement	[-32768, 32767]
Pokazivači	16 bita	Adresa u memoriji	[0x0, 0xFFFF]

Kao što se može zaključiti iz pomenute tabele, kod pomenute arhitekture ne postoji osmobićni tip podataka. Najamanji tip podataka je veličine 16-bita. Takođe bitna razlika u odnosu na većinu opštenamenskih procesora jeste u tome što je veličina *int* tipa 16 bita. Kako bi se izbegli problemi uzrokovani pogrešnom upotrebom tipova, najčešće se koriste oznake za celobrojne tipove definisane u okviru standardnog zaglavlja *stdint.h*. Nazivi tipova u ovom zaglavlju uključuju i veličinu tipa pa smanjuju rizik od pomenutih grešaka. S obzirom da je reć o standardnom zaglavlju, ono je definisano i za druge arhitekture. Zbog toga korišćenje ovog zaglavlja olakšava prevođenje napisanog C koda različitim prevodiocima i razlićite arhitekture. Na primer, *int16_t* na svim arhitekturama predstavlja celobrojni oznaćeni tip veličine 16 bita. Spisak tipova iz *stdint.h* zaglavlja za TMS320C55x procesore dat je u tabeli 3.

Tabela 3 - Tipovi definisani u zaglavlju *stdint.h* za TMS320C55x

Definisani simbol	Tip kome odgovara
<code>int16_t</code>	<code>int</code>
<code>uint16_t</code>	<code>unsigned int</code>
<code>int32_t</code>	<code>long</code>
<code>uint32_t</code>	<code>unsigned long</code>

Za izraćunavanje veličine tipa u toku izvršavanja može se koristiti operator *sizeof*. U pitanju je unarni operator koji sraćunava velićinu tipa i izraćava je u bajtima. Može se koristiti nad promenljivom ili nad tipom. U slućaju da se koristi nad promenljivom, povratna vrednost je velićina te promenljive tj velićina tipa kog je ta promenljiva. Kada radi nad tipom vraća njegovu velićinu. Ovde je potrebno napomenuti da je velićina bajta kod TMS320C5535 arhitekture jednaka 16 bita.

Po definiciji: `sizeof(char) = 1`

Kada je reć o tipovima u aritmetici pokrećnog zareza (*float*, *double*, *long double*), iako su definisani C standardom, ovi tipovi nisu podržani od strane svih arhitektura. Kod namenskih procesora sa redukovanim instrukcijskim setom (RISC) ovi tipovi mogu biti hardverski podržani, softverski podržani ili nepodržani. Hardverska podrška podrazumeva postojanje jedinice za izvršenje instrukcija nad brojevima u ovoj aritmetici, i kod procesora sa ovakvom podrškom preporućuje se korišćenje ovih tipova tokom implementacije. Softverska podrška podrazumeva postojanje softverskih biblioteka niskog nivoa koje emuliraju instrukcije nad tipovima u aritmetici pokrećnog zareza. Kod procesora sa softverskom podrškom ovi tipovi se mogu koristiti u ranoj fazi razvoja za potvrdu ispravnosti algoritma ili generisanje referentnih rezultata, dok je za obradu u realnom vremenu izvršenje emuliranih instrukcija najčešće isuviše sporo. TMS320C55x procesori imaju softversku podršku za tipove u aritmetici sa pokrećnim zarezom.

Tabela 4 – Vrste podršle aritmetike sa pokrećnim zarezom

Vrsta podrške aritmetike sa pokrećnim zarezom	Preporućeno korišćenje tipova
Nema podrške	Nemoguće korišćenje
Softverska podrška	Rani razvoj, provera ispravnosti algortima, obrada koja nije u realnom vremenu, nezahtevne obrade
Hardverska podrška	Kada god je potrebno

Pored svih navedenih tipova, u programskom jeziku C postoji i specijalni tip *void* koji označava da nema tipa. Promenljivu tipa *void* nije moguće definisati. Ovaj tip se koristi za definiciju pokazivača na nepoznatu vrednost ili definiciju funkcije koja nema povratnu vrednost, o čemu će biti više reči u daljem tekstu.

2.2 Promenljive

Bilo koji program predstavlja izvršavanje operacija nad nekim podacima. Za predstavljanje podataka u programskom jeziku C koriste se promenljive. Pre nego što se neka promenljiva može koristiti u okviru programskog jezika C potrebno je definisati tu promenljivu. Definicija promenljive sastoji se iz tipa promenljive i identifikatora koji predstavlja naziv te promenljive. Na primer

```
int16_t promenljiva1
```

Identifikator koji predstavlja naziv promenljive čini niz karaktera. Svaki identifikator mora započeti slovom ili „_“. Svi ostali karakteri mogu biti slova, cifre ili karakter „_“. Identifikatori ne smeju da budu jednaki nekoj od ključnih reči koje se koriste u programskom jeziku C (npr. *if*, *while*, *struct* i sl.). U okviru imena promenljivih razlikuju se mala i velika slova.

Po vidljivosti promenljive mogu biti lokalne i globalne. Lokalne promenljive su one definisane u okviru bloka ograničenog sa { } (npr. telo petlje ili funkcije). Ove promenljive su vidljive samo unutar tog bloka. Globalne promenljive su definisane van svih blokova, vidljiva je unutar čitave datoteke, a može biti i vidljiva iz drugih datoteka. Vidljivost promenljive iz druge datoteke (zatvorenost) određuje se pridruživanjem ključne reči *static*. Ukoliko je prilikom definicije pridružena pomenuta ključna reč, promenljiva je vidljiva samo u okviru datoteke, a ukoliko je nema promenljiva je vidljiva spolja. Da bi se nekoj globalnoj promenljivoj moglo pristupiti iz neke druge datoteke, potrebno je u okviru te datoteke napisati deklaraciju iste te promenljive sa pridodatom ključnom reči *extern*.

Primer	Promenljiva	Vidljivost
<pre>int prom1; static int prom2 void foo() { int prom3; ... }</pre>	prom1	Globalna promenljiva, vidljiva u čitavoj datoteci i iz spoljnih datoteka
	prom2	Globalna promenljiva, vidljiva samo u datoteci u kojoj je definisana
	prom3	Lokalna promenljiva, vidljiva samo unutar funkcije foo.

Dat je primer korišćenja globalne promenljive definisane u datoci Datoteka1.c iz datoteke Datoteka2.c:

Datoteka1.c <pre>int x; void foo1() { ... }</pre>	Datoteka2.c <pre>extern int x; void foo2() { x=2; ... }</pre>
---	---

Prilikom deklaracije promenljivoj je moguće dodeliti i ključnu reč *const*. Promenljive kojima je pridružena ova ključna reč ne mogu menjati svoju vrednost tokom izvršenja programa.

2.3 Operacije nad promenljivima

Programski jezik C omogućava određene operacije nad promenljivama. U tabeli 5 dat je pregled najčešće korišćenih aritmetičkih operacija. U poslednjoj koloni primer dat je primer korišćenja operatora. U zagradi je dat rezultat izraza za vrednosti promenljivih $A = 10$ i $B = 20$.

Tabela 5 - Aritmetički operatori u programskom jeziku C

Operator	Objašnjenje	Primer
+	Sabiranje dva operanda	$C = A + B$ ($C = 30$)
-	Oduzimanje vrednosti drugog operanda od prvog	$C = A - B$ ($C = -10$)
*	Množenje dve promenljive	$C = A * B$ ($C = 200$)
/	Deljenje dve promenljive	$C = B / A$ ($C = 2$)
%	Deljenje po modulu. Rezultat je ostatak pri celobrojnem deljenju prvog operanda drugim	$C = B \% A$ ($C = 0$)
++	Uvećanje celobrojnog operanda za 1. Ukoliko se piše pre operanda operacija će se izvršiti pre pristupa vrednosti, a ukoliko se piše posle izvršiće se posle pristupa vrednosti.	$A++$ ($A = 11$) $C = A++$ ($C = 10, A = 11$) $C = ++A$ ($C = 11, A = 11$)
--	Umanjenje celobrojnog operanda za 1. Važi isto pravilo kao i za prethodnu operaciju.	$C = A--$ ($C = 10, A = 9$) $C = --A$ ($C = 9, A = 9$)

Logičke operacija najčešće se koriste za ispitivanje relacije dve promenljive. Povratna vrednost logičkih izraza tačnost iskaza.

Tabela 6 – Logički operatori u programskom jeziku C

Operator	Objašnjenje	Primer
==	Provera da li su vrednosti jednake	$A == B$ (netačno)
!=	Provera da li su vrednosti različite	$A != B$ (tačno)
>	Provera da li je prva vrednost veća od druge	$A > B$ (netačno)
<	Provera da li je prva vrednost manja od druge	$A < B$ (tačno)
>=	rovera da li je prva vrednost veća ili jednaka drugoj	$A >= B$ (netačno).
<=	Provera da li je prva vrednost manja ili jednaka drugoj	$A <= B$ (tačno).
&&	Logičko "I". Provera da li su obe vrednosti različite od 0.	$A \&\& B$ (tačno)
	Logičko "ILI". Provera da li je bar jedna vrednost različita od 0.	$A B$ (tačno)
!	Negacija. Ukoliko je trenutna logička vrednost izraza tačna, postaje netačna i suprotno.	$!(A \&\& B)$ (netačno).

Pored logičkih i aritmetičkih u programskom jeziku C definisane su i tzv. bitski operatori. Ovi operatori izvršavaju se nad bitskom predstavom brojeva tako što se operacija predstavljena operatorom izvršava

nad svakim pojedinačnim bitom. Spisak bitskih operacija dat je u tabeli 7. Za date primere pretpostavljene su osmobitne promenljive čije su vrednosti $A = 60$ (0011 1100) i $B = 13$ (0000 1101).

Tabela 7 – Bitski operatori u programskom jeziku C

Operator	Objašnjenje	Primer
&	Logičko „I“. Vrednost svakog bita rezultujuće vrednosti sa pozicijom n predstavlja rezultat logičke operacije „I“ između n -tog bita prvog i n -tog bita drugog operanda.	$C = A \& B$ ($C = 12$ ili 0000 1100)
 	Logičko „ILI“ između odgovarajućih bita dva operanda.	$C = (A B)$ ($C = 61$ ili 0011 1101)
^	Logičko „Ekskluzivno ILI“. Ako su n -ti bit prvog i drugog operanda različiti, n -ti bit rezultata je 1, u suprotnom 0.	$C = (A \wedge B)$ ($C = 49$, ili 0011 0001)
~	Negacija nad bitima. Vrednost svakog bita je promenjena.	$C = (\sim A)$ ($C = -61$ ili 1100 0011)
<<	Pomeraj u levo. Rezultat je vrednost levog operanda čija je bitska predstava pomerenjena u levo za N mesta, gde je N vrednost drugog operanda.	$C = A \ll 2$ ($C = 240$ ili 1111 0000)
>>	Pomeraj u desno. Rezultat je vrednost levog operanda čija je bitska predstava pomerenjena u desno za N mesta, gde je N vrednost drugog operanda.	$C = A \gg 2$ ($C = 15$ ili 0000 1111)

Najčešće korišćen operator u programskom jeziku C je operator dodele „=“. Ovaj operator vrši dodelu vrednosti izraza sa desne strane resursu predstavljenom izrazom sa leve strane. Pored osnovnog operatora dodele postoje i izvedeni operatori koji kombinuju aritmetički ili bitski operator (koji nije unarni) i operator dodele. Kod ovakvih operatora izračunava se vrednost operacije označene sa operatorom koji stoji uz „=“ tako što se kao operandi uzmu vrednosti sa leve i desne strane operatora, i potom se vrednost smesti u resurs predstavljen izrazom sa leve strane. Npr.

```
A += B  <=>  A = A + B
A *= B  <=>  A = A * B
A >>= B  <=>  A = A >> B
```

2.4 Nizovi

Niz u programskom jeziku C predstavlja blok memorije koji se tumači kao da se u njemu, jedan za drugim, nalaze elementi iste veličine. Elementi niza su kontinualne memorijske lokacije. Niz se u programskom jeziku C definiše u sledećem formatu:

```
<tip elementa> <ime_niza> [< broj elemenata>] = {<elementi niza>;
```

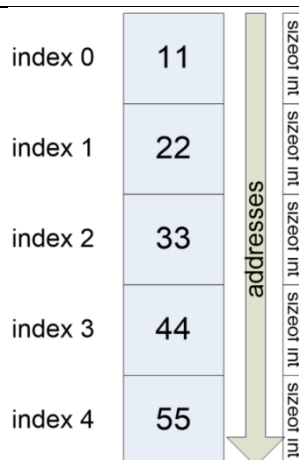
Kao i kod regularnih promenljivih, ako nema eksplicitne inicijalizacije, nizovi statičke trajnosti postavljaju se na nulu (svi njihovi elementi), a automatske neće (ostaće nedefinisane). Za eksplicitnu inicijalizaciju koristi se lista elemenata u vitičastim zagradama. Lista ne sme imati više elemenata od dimenzije niza. Ako ima manje, preostali elementi se popunjavaju nulama. Ako postoji inicijalizacija niza, dimenzija može biti izostavljena. Tada će dimenzija biti određena brojem elemenata u listi za inicijalizaciju. Dat je primer

definicije i inicijalizacije nizova. Ukoliko je *array1* lokalni niz vrednosti elemenata niza će biti nedefinisane {ND, ND, ND, ND, ND}, ukoliko je globalni vrednosti su {0, 0, 0, 0, 0}. Vrednosti niza *array2* su {1, 3, 5, 0, 0}.

```
int array1[5];  
int array2[5] = {1,3,5};
```

Elementima niza pristupa se koristeći operator indeksiranja []. Dat je primer definicije celobrojnog niza dužine 5 elemenata, čiji su elementi redom 11, 22, 33, 44, 55. Nakon toga sledi čitanje trećeg elementa niza (44) i smeštanje u promenljivu a.

```
int array[5] = {11, 22, 33, 44, 55};  
a = array[3];
```



Slika 4 – Prikaz memorije koja sadrži niz *array* definisan u prethodnom primeru.

2.5 Pokazivači

Pokazivači su promenljive čije su vrednosti adrese nekih podataka u memoriji. Pokazivači se mogu odnositi (mogu pokazivati) na promenljive ili funkcije. Deklaracija pokazivača data je u fromi:

<tip> * <naziv>;

gde *tip* može predstavljati tip podatka na koji pokazivač pokazuje, ne i tip pokazivača (tip pokazivača je pokazivač na *tip*), a naziv predstavlja naziv nove promenljive odnosno pokazivača. Specijalni karakter „*” može da stoji uz ime tipa, ime promenljive ili odvojen od oba. Primer:

```
int* ptr1;  
int * ptr2;  
int *ptr3;
```

Sva tri primera predstavljaju pokazivače istog tipa.

Dodeljivanje vrednosti pokazivaču se vrši koristeći operator dodele, s tim da je pokazivaču potrebno dodeliti adresu neke promenljive ili funkcije. Adresi promenljive ili funkcije se pristupa koristeći operator &. Dat je primer dodele adrese jedne promenljive pokazivaču. U ovom primeru *ptr* je pokazivač, a *value* promenljiva na koju nakon izvršenog koda *ptr* pokazuje.

```
int *ptr;  
int value = 5;  
ptr = &value;
```

Pokazivač na neki tip može uzeti samo adresu tog istog tipa, u suprotnom kompajler prijavljuje upozorenje ili grešku. Jedini izuzetak je pokazivač na void tip koji može uzeti vrednost bilo kog tipa.

```
int var;  
float* fptr;  
void* vptr;  
  
fptr = &var;          /* Greška */  
vptr = &var;          /* OK */
```

Pristup vrednosti na koju pokazivač pokazuje naziva se dereferenciranje pokazivača. Dereferenciranje se vrši koristeći operator „*“. Primer dereferenciranja pokazivača prilikom čitanja vrednosti memorijske lokacije na koju pokazivač pokazuje dat je sa:

```
int *ptr;  
int value = 5;  
ptr = &value;  
int a = *ptr;          /* a = value */
```

Tip i veličina podatka koji je pročitana iz memorije određen je tipom pokazivača. Primer upisa u memorijsku lokaciju na koju pokazivač pokazuje dat je sa:

```
*ptr = 7;              /* value = 7 */
```

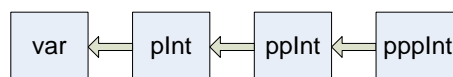
Operacije sabiranja i oduzimanja celog broja i pokazivača izvršavaju se tako što se pokazivač poveća za dati broj pomnožen veličinom tipa na koji pokazivač pokazuje. Isto važi i za unarne operatore ++ i --.

```
data_type* ptr;  
ptr ± n <=> ptr ± sizeof (data_type)
```

Oduzimanje dva pokazivača je moguće samo ako su pokazivači istog tipa. Sama operacija oduzimanja jedino ima smisla ako pokazivači pokazuju na adrese unutar istog parčeta memorije, u suprotnom rezultat će biti nedefinisan. Sa druge strane sabiranje dva pokazivača nije dozvoljeno po standardu te će kompajler u tom slučaju prijaviti grešku. Slično je i za poređenje dva pokazivača odnosno poređenje ima smisla samo ako pokazivači ukazuju na isto parče memorije.

Pokazivač može pokazivati na bilo koji tip, pa tako i na tip pokazivača. Primer pokazivača, i pokazivača na pokazivač dat je ispod.

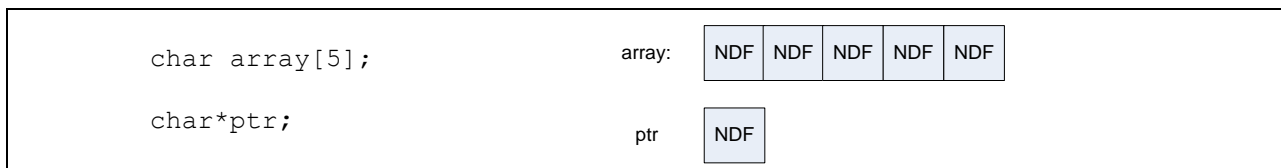
```
int var;  
int* pInt = &var;  
int** ppInt = &pInt;  
int*** pppInt = &ppInt;
```



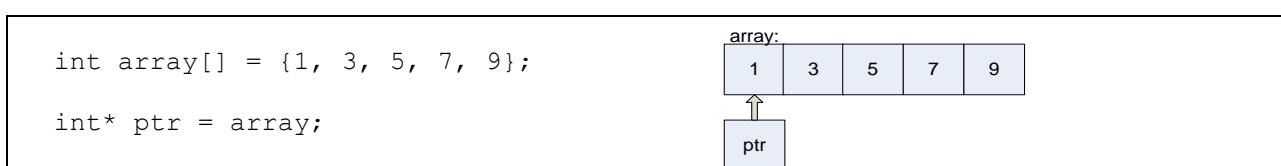
Višestruki pokazivači su potrebni u dva scenarija. Prvi je, kada je u programu neophodno proslediti pokazivač po referenci, a drugi, u slučaju rada sa višedimenzionalnim nizovima.

2.5.1 Odnos nizova i pokazivača

Iako je već rečeno da su nizovi i pokazivači vrlo slični, ipak postoje razlike. Kada se definiše niz, zauzima se memorija za sve njegove elemente. Kada su u pitanju pokazivači to nije slučaj, jer se memorija zauzima samo za taj pokazivač. Primer:



Niz, odnosno ime niza, predstavlja njegovu adresu, adresu početka niza (prvog elementa). Pokazivač je sa druge strane samo promenljiva čija je vrednost neka adresa. U tom smislu se niz može posmatrati kao pokazivačka konstanta.



Pored prethodno navedenih razlika, razlika nizova i pokazivača se ogleda i u rezultatu *sizeof* operacije:

- za niz vraća broj memorijskih reči koje su zauzete za ceo niz
- za pokazivač vraća koliko memorijskih reči je zauzeto za adresu

```
char array[15];
char* ptr;
printf("%d\n", sizeof(array));    /* rezultat je 15 */
printf("%d\n", sizeof(ptr));     /* rezultat je 1 */
```

Razlika u & operatoru:

- Za niz vraća adresu prvog elementa (isto što i &array[0])
- Za pokazivač vraća adresu pokazivača

```
char array[] = {'a', 'b', 'c', 'd', 'e'};
char* ptr = array;

if (array[0] == ptr[0])          /* ispis je "jednako" */
    printf("jednako");
else
    printf("nije jednako");

if (&array == &ptr)              /* ispis je "nije jednako" */
    printf("jednako");
else
    printf("nije jednako");
•
```

2.6 Zauzimanje memorije

Prilikom definicije nizova programski jezik C podržava dva načina zauzimanja memorije kroz definisanje samih promenljivih:

- Statičko zauzimanje – koje se primenjuje za sve globalne i statički definisane promenljive. Svaka promenljiva određuje jedan blok memorije, nepromeljive veličine. Memorija se zauzima jednom (prilikom pokretanja programa) i nikad se ne oslobađa.
- Automatsko zauzimanje – koje se primenjuje za automatske promenljive tj za argumente funkcija ili lokalno definisane promenljive. Memorijski prostor za ovakve promeljive se zauzima svaki put kad se u toku izvršavanja programa uđe u programski blok koji sadrži definiciju promeljive, a oslobađa se na kraju tog programskog bloka.

Zajedničko za statičko i automatsko zauzimanje memorije je da veličina koju je potrebno zauzeti mora biti konstantna vrednost.

- ```
• static int var = 5;
• int array1[10];
• int array2[var]; /* greška */
```

Treći način zauzimanja memorije - dinamičko zauzimanje nije podržano preko C-e promenljivih, ali je dostupno preko standardnih biblioteka (stdlib) i preko C++ operatora.

- Dinamičko zauzimanje – tehnika koja se primenjuje u slučajevima kada informacije o veličini i trajanju promenljivih nisu dostupne pre početka izvršavanja programa. Predstavlja akciju gde programer eksplicitno zatraži zauzimanje memorije. Sa obzirom da zauzimanjem upravlja programer, dužan je i da zauzetu memoriju na ispravan način oslobodi. U suprotnom može doći do grešaka koje je teško uočiti i otkloniti (curenje memorije).

Kod programa koji su namenjeni izvršavanju na uređajima sa ograničenim resursima ovaj pristup se ređe koristi. Pregled funkcija za dinamičko zauzimanje i oslobađanje memorije dat je u tabeli 8.

Tabela 8 - Funkcije za dinamičko zauzimanje i oslobađanje memorije

| Funkcija                                            | Opis                                                                                     |
|-----------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>void* malloc (size_t size)</code>             | Omogućava zauzimanje memorije . Parametar predstavlja veličinu u bajtima                 |
| <code>void* realloc (void* ptr, size_t size)</code> | Promena veličine zauzetog memorijskog bloka.                                             |
| <code>void* calloc (size_t num, size_t size)</code> | Zauzimanje memorije, i inicijalizacija svih memorijskih lokacija na prosleđenu vrednost. |
| <code>void free (void* ptr)</code>                  | Oslobađanje zauzetog memorijskog bloka                                                   |

## 2.7 Kontrola toka izvršenja

Programski jezik C sadrži kontrolne direktive koje služe sa upravljanje tokom izvršavanja programa. Ove direktive omogućavaju uslovno ili ponovljeno izvršavanje određenog dela koda.

Uslovno izvršavanje određenog dela koda u zavisnosti od tačnosti nekog izraza postiže se korišćenjem IF-ELSE strukture. Format ove strukture je dat sa:

```
If (<uslovni izraz>)
 <Kod koji se izvršava ukoliko je izraz tačan>
else
 <Kod koji se izvršava ukoliko izraz nije tačan>
```

Grana ELSE može biti i izostavljena. U koliko kod koji se izvršava sadrži više od jednog iskaza potrebno ga je predstaviti blokom ograničenim sa „{“ i „}”. Primer korišćenja IF-ELSE iskaza dat je ispod.

```
if(a < 20)
{
 printf("a je manje od 20\n");
}
else
{
 printf("a nije manje od 20\n");
}
```

U slučaju kada je potrebno odabrati između nekoliko različitih segmenata koda, korišćenje IF-ELSE konstrukcije može postati previse složeno, nepregledno ili u nekim slučajevima neefikasno. Druga mogućnost za implementaciju ovakve konstrukcije jeste SWITCH iskaz. SWITCH omogućava odabir jednog segmenta koda koji će biti izvršen. Format SWITCH iskaza je dat sa:

```
switch (<celobrojni izraz>)
{
 case <konstantna vrednost 1>:
 <kod>
 break;
 case <konstantna vrednost 2>:
 <kod>
 break;
 ...
 default:
 <kod>
}
```

Kod ovakvog iskaza, celobrojni izraz dat u zagradi se evaluira i vrši se skok na CASE izraz sa dobijenom vrednosti i izvršava se kod koji sledi. Ukoliko takva vrednost ne postoji, skače se na izraz DEFAULT. Iskaz *break* služi za prekidanje daljeg izvršenja i izlazak iz SWITCH konstrukcije. U slučaju da je *break* izostavljen, nastavlja se sa daljim izvršavanjem koda nakon sledećeg CASE izraza. Dat je primer jednog SWITCH iskaza.

```
switch (x)
{
 case 1:
 printf("Vrednost x je 1 \n");
 break;
 case 2:
```

```
 printf("Vrednost x je 2 \n");
 break;
default:
 printf("Vrednost x nije ni 1 ni 2 \n");
 break;
}
```

Pored uslovnih iskaza programski jezik C podržava konstrukciju petlji koje omogućavaju ponovljeno izvršavanje određenog dela koda. Postoje tri tipa petlji, to su WHILE; DO-WHILE i FOR.

WHILE petlja podrazumeva ponovljeno izvršavanje koda dokle god je zadati uslov ispunjen odnosno različit od 0. Pre izvršavanja svake iteracije vrši se ponovno izračunavanje izraza koji predstavlja uslov. Ukoliko uslov pre prve iteracije nije zadovoljen, telo petlje nikada neće biti izvršeno. Format WHILE konstrukcije dat je sa:

```
while (<izraz>)
{
 <kod>
}
```

DO-WHILE petlja podrazumeva proveru uslova nakon izvršenja tela petlje. Petlja se takodje ponavlja sve dok je zadati uslov zadovoljen. Ovakav pristup se koristi najčešće kada upit da li je potrebno ponoviti izvršavanje koda zavisi od rezultata obrade unutar tela petlje. U slučaju DO-WHILE telo petlje će se uvek izvršiti makar jednom.

```
do
{
 <kod>
} while (<izraz>)
```

Za razliku prethodnih, FOR petlja uz ključnu reč sadrži 3 izraza. Prvi izraz predstavlja izraz za inicijalizaciju i izvršava se samo jednom pre početka izvršenja petlje. Potom sledi izraz koji predstavlja uslov kraja petlje kao kod WHILE konstrukcije. Treći izraz jeste akcija koja se izvršava u svakoj iteraciji nakon izvršenja tela petlje. Izrazi su odvojeni karakterom „;“.

```
for (<izraz za inicijalizaciju>; <uslov>; <akcija>)
{
 <kod>
}
```

Sva tri pomenuta izraza u okviru FOR petlje su opciona, odnosno ne moraju postojati. Petlja bez ijednog izraza čini beskonačnu FOR petlju: *for(;;)*.

Dat je primer jedne petlje napisane koristeći tri različite konstrukcije koje programski jezik C podržava. Petlja vrši ispis prvih 10 prirodnih brojeva.



| WHILE                                                                       | DO-WHILE                                                           | FOR                                                                             |
|-----------------------------------------------------------------------------|--------------------------------------------------------------------|---------------------------------------------------------------------------------|
| <pre>x = 0; while ( x &lt; 10 ) {     printf( "%d\n", x );     x++; }</pre> | <pre>for (x=0; x &lt; 10, x++ ) {     printf( "%d\n", x ); }</pre> | <pre>x = 0; do {     printf( "%d\n", x );     x++; } while ( x &lt;= 10 )</pre> |

Bilo koja petlja može se prekinuti koristeći ključnu reč *break*. U slučaju ugnjeđenih petlji, *break* se odnosi samo na unutrašnju (ugnježdenu) petlju. Pored prekida čitave petlje, moguće je u okviru petlje prekinuti samo trenutnu iteraciju, i skočiti na proveru uslova. To se vrši koristeći ključnu reč *continue*.

|                                                                                                                                       |                                                |
|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| <pre>for ( x = 0; x &lt; 10; x++ ) {     if (x == 2)         continue;     if(x == 7)         break;     printf( "%d\n", x ); }</pre> | <b>Rezultat:</b><br>0<br>1<br>3<br>4<br>5<br>6 |
|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|

## 2.8 Funkcije

Program u pisan koristeći programski jezik C sastoji se od jedne ili više funkcija. Ukoliko nije drugačije specificirano početna funkcija od koje počinje izvršavanje programa jeste funkcija sa nazivom *main*. Funkcije mogu da pozivaju jedna drugu.

U programskom jeziku C definicija funkcije se sastoji iz tipa povratne vrednosti, identifikatora koji predstavlja ime funkcije, liste parametara i tela funkcije. Format definicije dat je sa:

|                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>&lt;tip povratne vrednosti&gt; &lt;ime funkcije&gt; ( &lt;argument 1&gt;, &lt;argument 2&gt;...) {     &lt;telo funkcije&gt; }</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------|

Dat je primer definicije funkcije *prosek*, koji računa prosečnu vrednost dva broja. Prva linija koda datog u prethodnom primeru predstavlja deklaraciju funkcije. Povratna vrednost funkcije je *int*. Funkcija sadrži dva parametra, *x* i *y* koji su oba tipa *int*. U okviru tela petlje, nalazi se kod koji će se izvršiti svaki put kada funkcija bude pozvana. Telo petlje završava iskazom *return* koji označava izlazak iz funkcije. Izraz koji stoji uz *return* čini vrednost koja će biti vraćena pozivaocu funkcije. Sve funkcije osim onih koje imaju tip povratne vrednosti *void*, moraju sadržati *return* iskaz.

|                                                                             |
|-----------------------------------------------------------------------------|
| <pre>int prosek( int x, int y) {     int z = (x+y)/2;     return z; }</pre> |
|-----------------------------------------------------------------------------|

Parametri funkcije se koriste za prosleđivanje vrednosti funkciji od strane dela programa koji je poziva. Prilikom poziva funkcije formalni parametri se zamenjuju stvarnim. Na primer:

```
void main()
{
 int a = 2, b = 3;
 prosek(a, b);
}
```

S obzirom da se parametri funkcije prenose po vrednosti, za svaki parametar se prilikom poziva pravi lokalna kopija. Svaka izmena nad parametrima unutar tela funkcije ne utiče na promenu vrednosti koje su originalno prosleđene. U prethodnom primeru u slučaju da je unutar funkcije *prosek*, promenljivoj *x* promenjena vrednost, promenljiva *a* u funkciji *main* bi i dalje ostala ne promenjena. Da bi se neka promenljiva mogla menjati unutar tela funkcije, i da ta promena bude vidljiva i van funkcije potrebno je proslediti adresu te promenljive, odnosno pokazivač na nju.

```
void uvecaj(int* x)
{
 *x = *x + 1;
}

void main()
{
 int a = 2;
 uvecaj(&a); //nakon poziva vrednost a je 3.
}
```

Da bi funkcija mogla biti pozvana ona mora biti definisana. Međutim programski jezik C omogućava poziv funkcije koja je definisana u drugoj datoteci, ili istoj datoteci ali nakon mesta gde je pozvana. Da bi se to omogućilo potrebno je napisati prototip funkcije pre njenog poziva. Prototip funkcije podrazumeva potpunu deklaraciju funkcije završenu sa karakterom „;“. Neki programski prevodioci omogućavaju poziv funkcije koja nije deklarasiрана, ali tu praksu treba izbegavati. Primer:

```
int foo(int x);

void main()
{
 int a = 2;
 uvecaj(&a); //nakon poziva vrednost a je 3.
}

int foo(int x)
{
 X++;
 return x;
}
```

Ako je potrebno deklarirati funkciju koja je definisana u spoljnoj datoteci, prilikom deklaracije funkciji se dodaje kvalifikator *extern* pre deklaracije. Ukoliko je potrebno ograničiti vidljivost promenljive samo na datoteku u kojoj je definisana, tada se koristi ključna reč *static* kao i kod promenljivih.

## 2.9 Pretprocesorske direktive

Pretprocesorske direktive omogućavaju izvršavanje određenih izmena na kodu pre samog prevođenja programa, odnosno u fazi pretprocesiranja. Pretprocesorske direktive omogućavaju zamenu određenog teksta u kodu drugim pre samog prevodjenja, uključivanje koda iz drugih datoteka, definisanje određenih simbola i uslovno prevođenje delova koda. Svaka pretprocesorska direktiva započinje znakom „#“.

Tabela 9 sadrži pregled pretprocesorskih direktiva podržanih od strane programskog jezika C. Pored navedenih određeni programski prevodioci podržavaju dodatne pretprocesorske direktive koje omogućavaju podešavanje načina prevođenja. Te direktive nisu opsiane C standardom već predstavljaju proširenja specifična za dati prevodilac.

Tabela 9 – Pregled pretprocesorskih direktiva u programskom jeziku C

| Direktiva                              | Opis                                                                                                                                                              |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#define SIMBOL VREDNOST</b>         | Gde god se SIMBOL pojavi u kodu, biće zamenjen sa VREDNOST                                                                                                        |
| <b>#include "naziv datoteke"</b>       | Na mesto ove direktive biće uključen celokupan sadržaj datoteke čiji je naziv naveden pod navodnicima.                                                            |
| <b>#include &lt;naziv datoteke&gt;</b> | Isto kao i prethodna direktiva. Jedina razlika je što pretprocesor prvo traži datoteku sa datim nazivom u sistemskim direktorijumima..                            |
| <b>#ifdef SIMBOL</b>                   | Uslovno prevođenje dela koda ukoliko je navedeni simbol ranije definisan. Odnosi se na sav kod do naredne #else ili #endif direktive.                             |
| <b>#ifndef SIMBOL</b>                  | Uslovno prevođenje dela koda ukoliko navedeni simbol nije definisan.                                                                                              |
| <b>#else</b>                           | Uslovno prevođenje u koliko prethodna #ifdef ili #ifndef direktiva nije ispunjena                                                                                 |
| <b>#endif</b>                          | Završetak dela koda koji se uslovno prevodi                                                                                                                       |
| <b>#undef SIMBOL</b>                   | Poništava definiciju nekog simbola. Od pozicije ove direktive do kraja datoteke ili ponovne definicije simbola pretprocesor će smatrati da simbol nije definisan. |

Najčešće korišćena direktiva u programskom jeziku C jeste `#include` direktiva. Ona omogućava uključivanje sadržaja drugih datoteka u trenutnu datoteku koja se prevodi. Prilikom navođenja naziva datoteke koja se uključuje navodi se putanja do datoteke uokvirena navodnicima ("datoteka") ili znakovima manje i veće (<datoteka>). Razlika između ova dva načina jeste u tome što se u prvom slučaju datoteka pretražuje prvo na putanjama specifikiranim od strane korisnika. U drugom slučaju prvo se pretražuju sistemske putanje. Druga opcija se najčešće koristi za zaglavlja standardnih biblioteka.

Definicija simbola može da sadrži vrednost na koju je simbol definisan. U tom slučaju svaka pojava definisanog simbola u kodu biće zamenjena tom vrednošću u fazi pretprocesiranja. Ukoliko definicija ne sadrži vrednost, taj simbol služi samo za logičke provere koristeći direktive za uslovno prevođenje koda. Posebna vrsta simbola su makro definicije. Makro predstavlja definiciju simbola sa parametrima. U slučaju makro definicije svi prilikom zamene simbola vrednošću, svi specifikirani parametri se zamenjuju

parametrima koji se nalaze u zagradi nakon simbola. Dat je primer makro definicije u okviru koje se simbol DO menja FOR petljom. Simbolu DO se prosleđuju 3 parametra. Ispod primera je dat rezultat pretprocesiranja.

```
#define DO(var, beg, end) for(var = beg; var <= end; var++);

void foo()
{
 DO(i, 1, 10)
 printf("Hello world!\n");
}

Rezultat pretprocesiranja:
void foo()
{
 for(i = 0; i <= 10; i++)
 printf("Hello world!\n");
}
```

U slučaju poziva makro simbola potrebno je izbegavati prosleđivanje izraza, s obzirom da pretprocesor ne vrši evaluaciju izraza, već samo zamen teksta. Dat je primer česte greške koja nastaje kao posledica prosleđivanja izraza kao parametara prilikom korišćenja makro simbola. Primer predstavlja korišćenje DO makroa definisanog u prethodnom primeru.

```
DO(i++, 1, 10)
 printf("Hello world!\n");

Rezultat pretprocesiranja:

for(i++ = 0; i++ <= 10; i++++) //GREŠKA
 printf("Hello world!\n");
```

Upotreba direktiva za uslovno prevođenje koda prikazana je na sledećem primeru.

```
#define DEBUG
void foo()
{
#ifdef DEBUG
 printf("DEBUG simbol je definisan!\n");
#else
 printf("DEBUG simbol je definisan!\n");
#endif
}

Rezultat pretprocesiranja:
void foo()
{
 printf("DEBUG simbol je definisan!\n");
}
```

### 3 Razvojno okruženje Code Composer™ Studio

*Code Composer™ Studio* (CCS) predstavlja integrisano razvojno okruženje (IDE) za *Texas Instruments* mikrokontrolere i namenske procesore. CCS nudi set alata koji omogućavaju razvoj i analizu namenske programske podrške. Pomenuti set alata uključuje C/C++ programski prevodilac, assembler, povezič, prozore za uređivanje izvornog koda, alat za kontrolisano izvršenje koda (eng. *debugger*), alat za profilisanje koda (eng. *profiler*), i mnoge druge.

CCS razvojno okruženje zasnovano je na *Eclipse* platformi. *Eclipse* predstavlja široko upotrebljivan kao osnov mnogih razvojnih okruženja. Otvorenog je koda i nudi mnoštvo korisnih alata za razvoj programske podrške. CCS proširenja u *Eclipse*-u odnose se na omogućivanje upotrebe posebnih odlika namenskih uređaja proizvedenih od strane *Texas Instruments*-a.

U toku ovog kursa koristićemo Code Composer™ Studio 6.

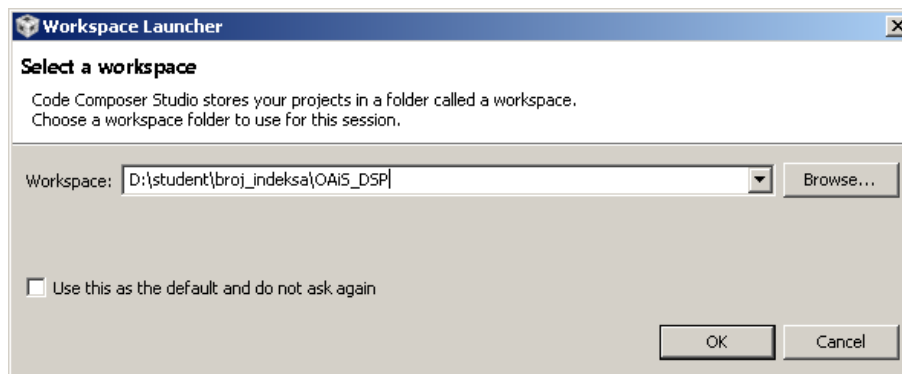
## 4 Upoznavanje sa CCS razvojnim okruženjem

### 4.1 Pokretanje CCS razvojnog okruženja

- **Zadatak:**
  - Pokrenuti Code Composer Studio razvojno okruženje.

Nakon pokretanja CCS razvojnog okruženja, od korisnika se zahteva da izabere putanju do radnog prostora (*workspace*). *Workspace* jeste direktorijum na disku koji će biti korišćen za čuvanje informacija tokom rada sa CCS okruženjem (aktivni projekti, podešavanja izgleda okruženja, log...). Ujedno predstavlja i podrazumevanu putanju pilikom pravljenja novih projekata.

Tokom rada sa CCS okruženjem moguće je imati aktivan samo jedan *workspace*. Dve instance CCS-a ne mogu deliti isti *workspace*.



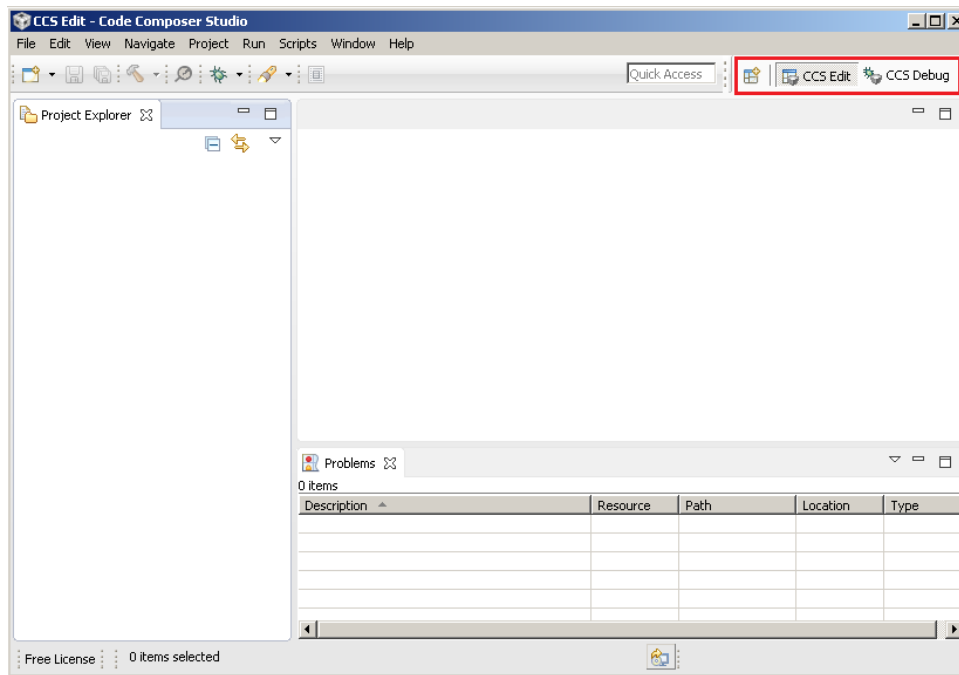
Slika 5 - Odabir putanje do radnog prostora

- **Zadatak:**
  - Odabrati željenu putanju do radnog prostora
  - Ukoliko je obeležena opcija "Use this as the default...", navedeni direktorijum će biti smatran za podrazumevani radni prostor i ovaj se više neće pojavljivati prilikom pokretanja CCS.

Radni direktorijum moguće je naknadno promeniti iz menija *File -> Switch Workspace*.

## 4.2 Perspektiva

Perspektiva definiše skup i raspored aktivnih prikaza u radnom prozoru. CCS omogućava postojanje više različitih perspektiva, međutim u određenom trenutku samo jedna može biti aktivna. Perspektive su obično definisane tako da sadrže skup prikaza i alata potrebnih za ostvarenje nekog tipa zadatka. Na primer, u okviru *CCS Edit* perspektive aktivni su najčešće korišćeni prikazi i alati koji se koriste tokom pisanja izvornog koda, kao što su uređivač koda, pretraživač projekata, prikaz problema nastalih u toku prevođenja itd. Kada je pokrenut proces kontrolisanog izvršavanja programa, CCS automatski prelazi na perspektivu za kontrolisano izvršavanje programa (eng. *Debug perspective*). Meni, trake sa alatima, raspored aktivnih prikaza i alata u okviru ove perspektive prilagođen je procesu kontrolisanog izvršavanja programa. Korisnik može ručno da prelazi iz jedne perspektive u drugu. Sve izmene nad perspektivom ostaju sačuvane, i vezane su za radni prostor. Vraćanje neke od ugrađenih perspektiva na podrazumevani izgled vrši se naredbom *Window->Reset Perspective*. Pravljenje novih perspektiva vrši se tako što se trenutna perspektiva sačuva pod drugim imenom (*Window->Save Perspective As...*)



Slika 6 - CCS Edit perspektiva (crvenom bojom su označena dugmad za promenu perspektive)

## 4.3 Projekti

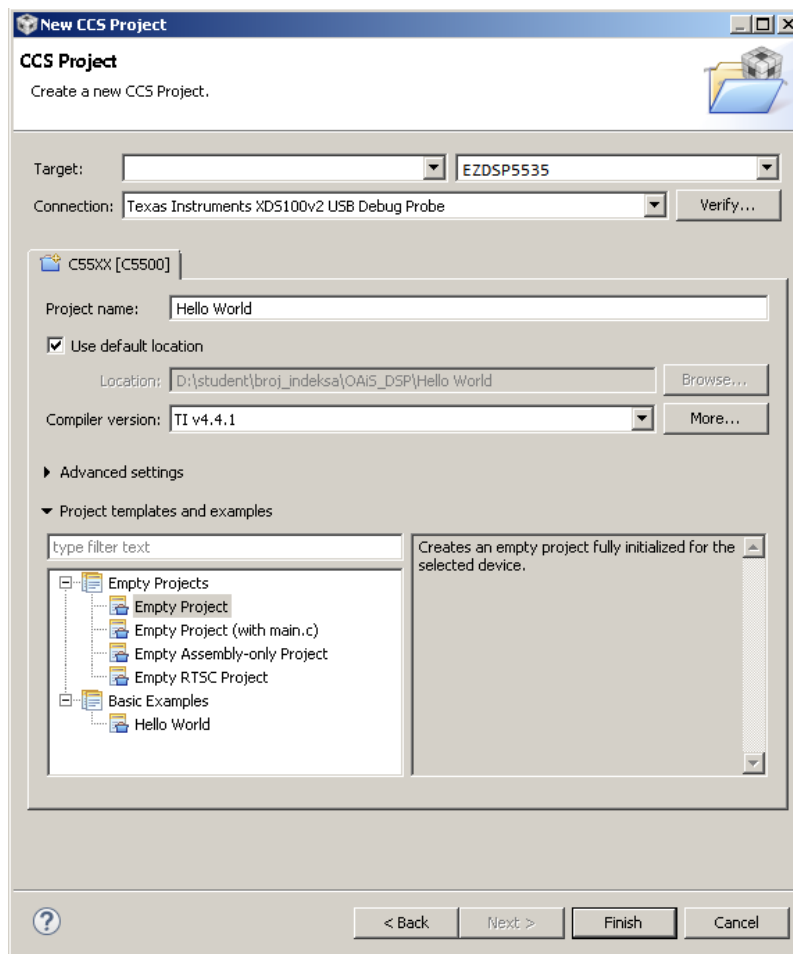
Datoteke unutar CCS radnog prostora organizovane su po projektima. Svaki projekat predstavlja skup direktorijuma i datoteka. Za pregled projekata definisanih u radnom prostoru koristi se *Project Explorer View* komponenta. Datoteke unutar projekta mapirane su na direktorijume i datoteke na disku. Svaka izmena, dodavanje ili brisanje datoteka unutar prikaza projekta u okviru *Project Explorer View* prozora, rezultiraće istom izmenom nad datotekama na disku.

#### 4.3.1 Pravljenje novog projekta

Kreiranje novog projekta vrši se odabirom opcije *File -> New -> CCS Project* ili aktiviranjem padajućeg menija desnim tasterom miša unutar *Project Explorer View*-a, pa *New -> CCS Project*.

Nakon ovog koraka otvara se čarobnjak za pravljenje novog projekta. Prilikom pravljenja novog projekta potrebno je odabrati željenu ciljnu platformu.

Pod opcijom *Target* prvi padajući meni omogućava filtriranje ponuđenih ciljnih platformi unošenjem dela imena ili odabirom familije procesora kojoj željena ciljna platforma pripada, a drugi tačan naziv procesora. Pod poljem *Connection* podrebno je izabrati spregu putem koje CCS komunicira sa datim procesorom. Ukoliko je razvojna ploča povezana na računar, moguće je izvršiti proveru ispravnosti odabranih podešavanja ciljne platforme opcijom *Verify*. Očekivani ispis na dnu konzole nakon pokretanja ove opcije prikazan je na slici 8.

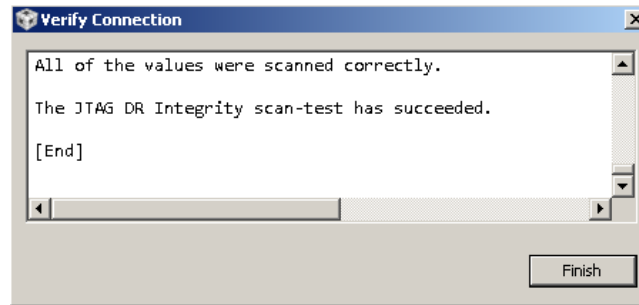


Slika 7 - Pravljenje novog projekta

Pored ciljne platforme potrebno je odabrati i naziv i putanju na kojoj se projekat nalazi (podrazumevana putanja je aktivni *workspace*).



Dodatna podešavanja novog projekta uključuju odabir verzije programskog prevodioca koji se koristi, šablona za kreiranje projekta (šabloni sadrže unapred predefinisane izvorne datoteke i podešavanja).



Slika 8 - Uspešna verifikacija odabrane ciljne platforme

- Napomena: ukoliko koristite simulator, u ovoj fazi provera odabira razvojne ploče će prikazati grešku u povezivanju.

Iako je napravljen prazan projekat, on ipak sadrži određene datoteke i to:

- *targetConfigs/ EZDSP5535.ccxml* – predstavlja konfiguracionu datoteku sa opisom ciljne platforme
- *C5535.cmd* – datoteka sa podešavanjima povezivača

Datoteka sa podešavanjima ciljne platforme (*Target Configuration File*) sadrži sve informacije potrebne za pokretanje programa i njegovo kontrolisano izvršavanje na određenom uređaju. Generisana je na osnovu odabrane ciljne platforme za projekat prilikom kreiranja novog projekta. Datoteka je generisana u XML formatu, i ima nastavak **.ccxml**. Informacije koje se nalaze unutar ove datoteke su: tip uređaja na kome će se izvršavati kod, tip sprege uređaja sa računarom i mogućnost dodavanja *GEL* datoteke. *GEL* (*General Extension Language*) jeste skripta koja služi za inicijalizaciju ciljnog uređaja.

Dvostrukim klikom na **.ccxml** datoteku otvara se grafički editor u okviru kog je moguće promeniti pomenute parametre. Takođe je moguće izvršiti proveru ispravnosti pritiskom na *Test Connection* dugme (radi isto što i *Verify* prilikom pravljenja projekta).

- **Zadatak (ukoliko koristite simulator):**
  - Otvoriti datoteku *EYDSP5535.ccxml*
  - Promeniti sadržaj polja *Connection* na *Texas Instruments Simulator*
  - Odabrati kao ciljni uređaj *C55x Rev3.0 CPU Functional Simulator*

Datoteka sa **.cmd** nastavkom označava komandnu datoteku koja se prosleđuje povezivaču u toku prevođenja projekta. U okviru ove komandne datoteke specifikovana je podela memorije na blokove (broj, nazivi i veličine memorijskih blokova) kao i označavanje pripadnosti sekcija memorijskim blokovima. Svaki projekat sme da sadrži samo jednu datoteku sa **.cmd** nastavkom.

#### 4.4 Dodavanje datoteka sa izvornim kodom u projekat

Dodavanje nove datoteke u projekat vrši se odabirom opcije File -> New -> File (s obzirom da koristimo programski jezik C, moguće je izabrati šablon za izvornu datoteku (Source File) ili zaglavlje (Header File)).

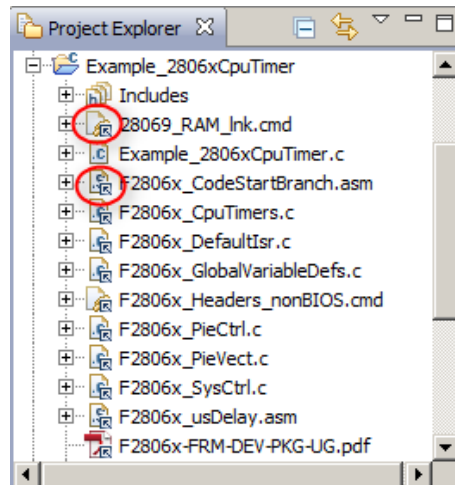
Dodavanje postojećih datoteka sa izvornim kodom može se izvršiti na dva načina:

- Dodavanjem datoteke
- Povezivanjem

Dodavanjem datoteke, datoteka se fizički nalazi u okviru projekta. Na disku je smeštena unutar projektnog direktorijuma. Dodavanje datoteke povezivanjem omogućava da određeni projekat koristi datoteku koja se ne nalazi unutar njega. Datoteka je vidljiva unutar Project Explorer View-a i označena je posebnom ikonicom (Slika 7).

Kako bi dodali postojeću datoteku u projekat potrebno je: otvoriti meni *Project->Add Files*, i odabrati željene datoteke, ili odabrati datoteke unutar *Windows Explorer*-a i prevući ih u željeni projekat unutar *Project Explorer* view-a. U oba slučaja CCS će ponuditi izbor da li želite da dodate datoteke budu smeštene u projekat, ili samo povezane sa projektom ("*Copy files*" ili "*Link to files*").

Brisanje datoteka iz projekata vrši se odabirom datoteke za brisanje, pritiskom desnog tastera miša i odabirom opcije *Delete* iz padajućeg menija.



Slika 9 - Ikonica datoteka dodatih u projekat povezivanjem

- **Zadatak:**
  - Obrisati postojeću .cmd datoteku iz projekta
  - Dodati datoteku Inkx.cmd u projekat. Prilikom dodavanja koristiti opciju *Copy File*
  - Napraviti novu izvornu datoteku pod nazivom *main.c*. U okviru napravljene datoteke napisati kratak C program koji ispisuje pozdravnu poruku "Hello World!" u CCS konzolu

```
#include <stdio.h>

void main()
{
 printf("Hello World\n");
 return;
}
```

#### 4.5 Prevođenje

Prevođenje programa vrši se odabirom opcije *Project->Build Project* ukoliko želimo da prevedimo jedan projekat ili *Project->Build All* ukoliko želimo da pokrenemo prevođenje svih projekata unutar radnog prostora.

CCS vrši inkrementalno prevođenje, što znači da se ponovo prevode samo one datoteke koje su menjane nakon prethodnog prevođenja i one koje zavise od tih datoteka. Ponekad je neophodno prisilno prevesti ceo projekat, kako bismo bili sigurni da su sve izmene uračunate u poslednjim prevedenim objektnim datotekama. Za to se koristi opcija *Project->Clean*, koja briše sve datoteke nastale u toku prevođenja jednog projekta.

CCS ima podršku za različite konfiguracije prevođenja. Podrazumevane konfiguracije svakog projekata su:

- *Debug* – prevedeni program sadrži pun skup simboličkih operacija potrebnih za kontrolisano izvršavanje programa i praćenje promenljivih sa ciljem otkrivanja grešaka (*debug* informacije). Skoro sve optimizacione tehnike nad kodom koje je kompajler u stanju da sprovodi su isključene. Optimizacija koda komplikuje otklanjanje grešaka jer se prilikom optimizacije pravi veća razlika između izvornog koda i generisanih instrukcija
- *Release* - prevedeni program ne sadrži simboličke *debug* informacije i donosi potpunu optimizaciju.

Uređivanje konfiguracija prevođenja (dodavanje, brisanje, izmena) vrši se odabirom opcije *Project -> Properties*.

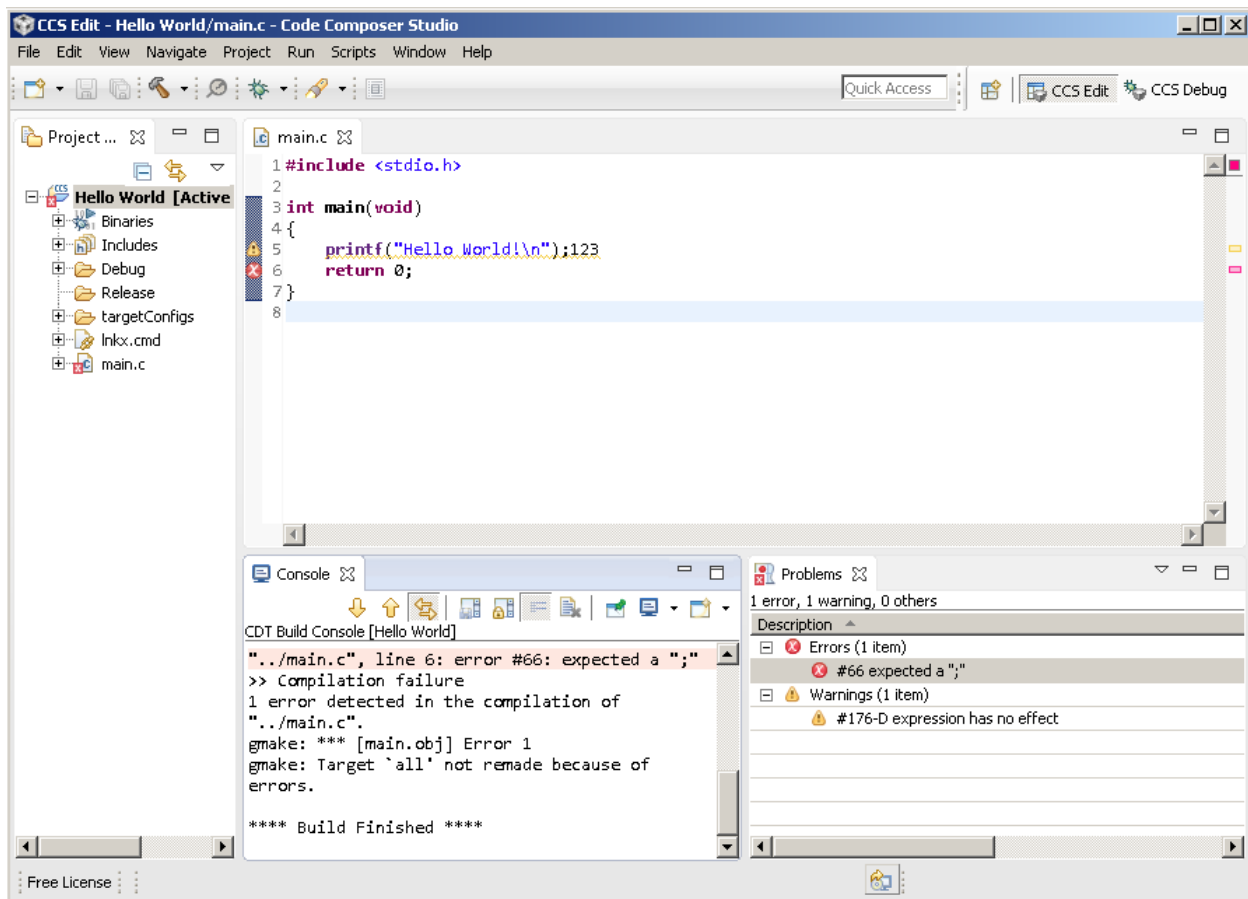
Odabir aktivne konfiguracije vrši se sa *Project -> Build Configurations -> Set Active -> [željena konfiguracija]*.

Nakon pokretanja prevođenja željenog programa, tok prevođenja (izvršene komande i ispis njihovih rezultata) možete ispratiti u CCS konzoli (Console View). Sve greške ili upozorenja, prijavljena u toku prevođenja od strane nekog od alata, biće izlistana u okviru *Problems View* prozora.

Dvoklikom na grešku ili upozorenje otvara se odgovarajuća datoteka sa kodom na mestu na koje se greška ili upozorenje odnosi.

- **Zadatak:**

- Odabrati *Debug* konfiguraciju kao aktivnu konfiguraciju prevođenja za *Hello World* projekat
- Dodati sintaksnu grešku u vaš izvorni kod. Pokrenuti prevođenje. Uočiti prijavljenu grešku.
- Ispraviti prijavljenu grešku, pokrenuti prevođenje.
- Ukoliko je datoteka uspešno prevedena, moguće je pokrenuti izvršenje programa na ciljnoj platformi.



Slika 10 - Prijavljena greška i upozorenje u toku prevođenja

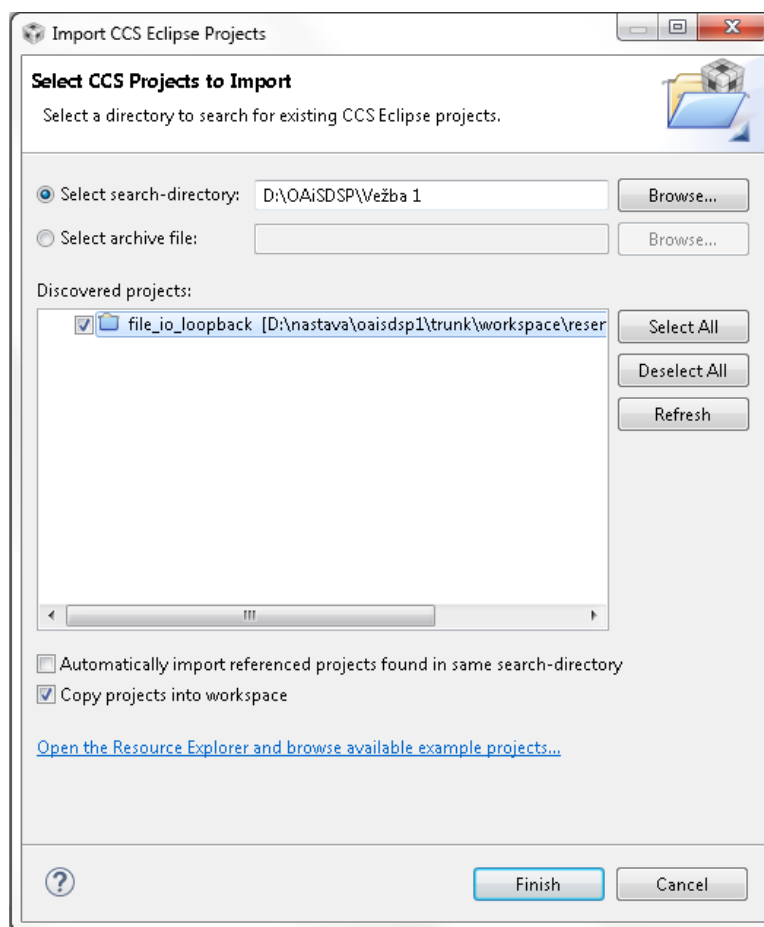
#### 4.6 Uvlačenje postojećih projekata u radni prostor

Ukoliko na disku postoji već napravljen projekat, da bi se omogućio rad nad njim potrebno je da pomenuti projekat postane deo aktivnog radnog prostora. Uvlačenje već postojećih projekata u radni prostor vrši se komandom **Project -> Import CCS Projects**.

U otvorenom prozoru potrebno je izabrati putanju do direktorijuma ili arhive koja sadrži željeni projekat. Putanja se može uneti ručno, ili pretražiti pritiskom na dugme *Browse*. Nakon odabira putanje, u okviru prozora su izlistani svi CCS projekti na datoj putanji. Potrebno je obeležiti jedan ili više projekata i pritisnuti dugme *Finish*.

Dodatne opcije koje ovaj čarobnjak pruža jesu:

- Automatsko uključivanje svih projekata koji se nalaze u listi zavisnosti svakog od projekata koji uključujemo. Lista zavisnosti omogućava da se označe projekti čije resurse određeni projekat koristi, i koji su neophodni za uspešno prevođenje i izvršavanje. Prilikom pokretanja prevođenja projekta, prvo se pokreće prevođenje svih projekata koji se nalaze u njegovoj listi zavisnosti. Lista zavisnosti se popunjava iz menija *Project->Properties->Build->Dependencies*.
- Kopiranje projekta u radni prostor. Ova opcija se odnosi na uključivanje projekata koji se ne nalaze u direktorijumu koji predstavlja aktivni radni prostor. Ukoliko se postavi na tačno, direktorijum koji sadrži projekat, sa svim datotekama biće prekopiran u direktorijum radnog prostora. Sve izmene nad projektom odnosiće se na prekopirane datoteke, ne na datoteke na putanji sa kojih je projekat originalno uključen. Ukoliko je neaktivna ova opcija, datoteke će biti uvučene u radni prostor, ali će se na disku nalaziti na drugoj lokaciji.



Slika 11 - Uključivanje postojeće datoteke u CCS radni prostor

- **Zadatak:**
  - Uključiti projekat *file\_io\_loopback* u aktivni radni prostor
  - Uključiti opciju kopiranja projekta u direktorijum radnog prostora
  - Pokrenuti prevođenje uključenog projekta.

#### 4.7 Pokretanje programa na razvojnoj ploči

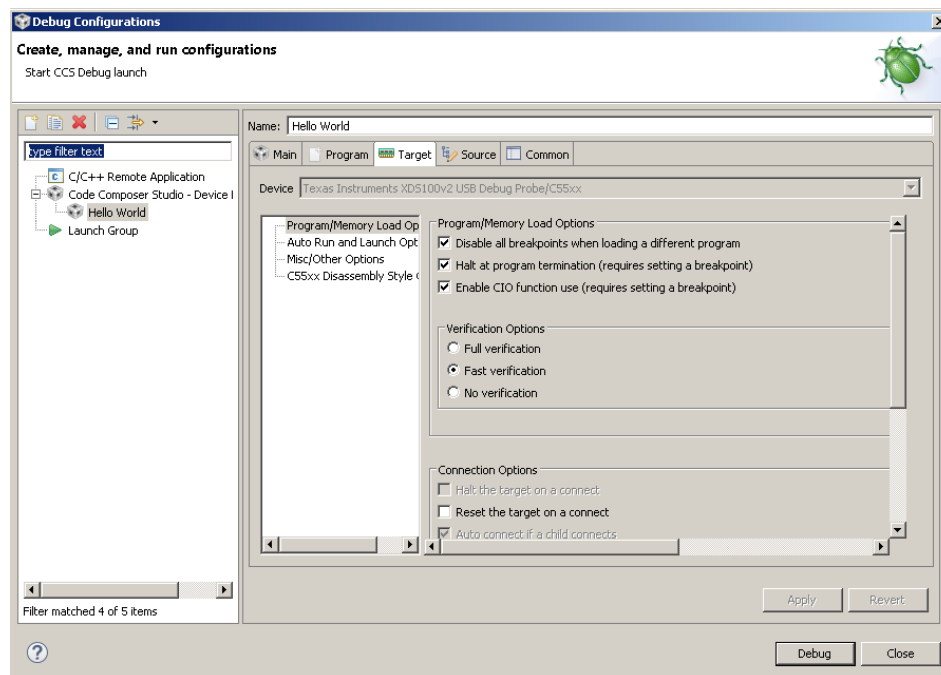
CCS razvojno okruženje omogućava učitavanje prevedenog programa u memoriju DSP uređaja, i njegovo izvršavanje. U toku izvršavanja datog programa CCS omogućava kontrolisanje toka izvršavanja, kao i dobavljanje informacija o stanju uređaja u toku izvršavanja. Pokretanje programa vrši se odabirom opcije *Run -> Debug*.

Tipičan scenario nakon odabira *Run -> Debug* akcije sastoji se iz sledećih koraka:

1. CCS učitava datoteku sa opisom ciljne platforme (*Target Configuration File*), definiše konfiguracionu datoteku *Debug procedure (Debug Configuration)*, i na osnovu informacija ove dve datoteke povezuje se na JTAG kontrolni port i započinje komunikaciju sa uređajem
2. Kada je komunikacioni kanal između CCS-a i uređaja uspostavljen, započinje se izvršenje serije naredbi za inicijalizaciju ciljnog uređaja iz *GEL* skripte ukoliko ona postoji.
3. Sadržaj objektna datoteke koja predstavlja rezultat prevođenja aktivnog projekta upisuje se u memoriju DSP uređaja putem JTAG kanala.
4. CCS automatski postavlja tačku prekida na adresu prve instrukcije unutar *main()* funkcije, resetuje se programski brojač DSP uređaja, i zaustavlja se izvršenje kada se dostigne postavljena tačka prekida.

##### 4.7.1 Podešavanja procesa kontrolisanog izvršavanja programa

Podešavanja procesa kontrolisanog izvršavanja programa (*Debug Configuration*) predstavlja standardan način za predstavljanje opisa pokretanja programa u *Eclipse* baziranim okruženjima. Ova podešavanja su vezana za radni prostor. Novo Debug podešavanje moguće je napraviti ručno izborom opcije *Run -> Debug Configurations*, ili automatski prilikom pokretanja izvršavanja programa sa *Run -> Debug*.




Slika 12 - Podešavanja kontrolisanog izvršenja programa



Opcije unutar Debug podešavanja organizovane su po tabovima:

- *Main* - osnovna podešavanja debug procedure (datoteka sa opisom ciljne platforme, skripta za inicijalizaciju, odabir aktivnih komponenti tokom procedure kontrolisanog izvršenja programa).
- *Program* – omogućava odabir izvršne datoteke koja će biti pokrenuta na ciljnoj platformi. Najčešće je to datoteka dobijena prevođenjem projekta za koji se konfiguracija definiše, ali moguće je i uključiti neku drugu izvršnu datoteku.
- *Target* – podešavanja osobina debug procedure vezana za uređaj na kome se program izvršava. Podešavanja su podeljena u četiri kategorije.
  - *Program/Memory Load Options* – podešavanja načina učitavanja izvršne datoteke ili skupa vrednosti u memoriju uređaja, podešavanja verifikacionog procesa, povezivanja, onemogućenja prekida ili reseta uređaja tokom učitavanja novih podataka.
  - *Auto Run and Launch Options* – definicija opcija koje se tiču izvršenja koda u realnom vremenu, automatskog pokretanja i učitavanja programa.
  - *Misc/Other Options* – ostala podešavanja
  - *Disassembly Style Options* – Odabir stila prilikom prikazivanja disasemblovanog izvršnog koda
- *Source* i *Common* predstavljaju standardna Eclipse-ova podešavanja koja se odnose na označavanje direktorijuma izvršnim datotekama i podešavanje prikaza i logovanja informacija primljenih od strane ciljne platforme u toku prevođenja.

#### 4.7.2 Pokretanje programa

Pored opcije *Run* -> *Debug* pokretanje programa moguće je izvršiti i iz prozora za izmene Debug podešavanja, pritiskom na dugme *Debug*. U tom slučaju biće pokrenuta konfiguracija čija podešavanja su trenutno prikazana. Treća opcija za pokretanje programa jeste pritiskom na dugme  na liniji sa alatima.

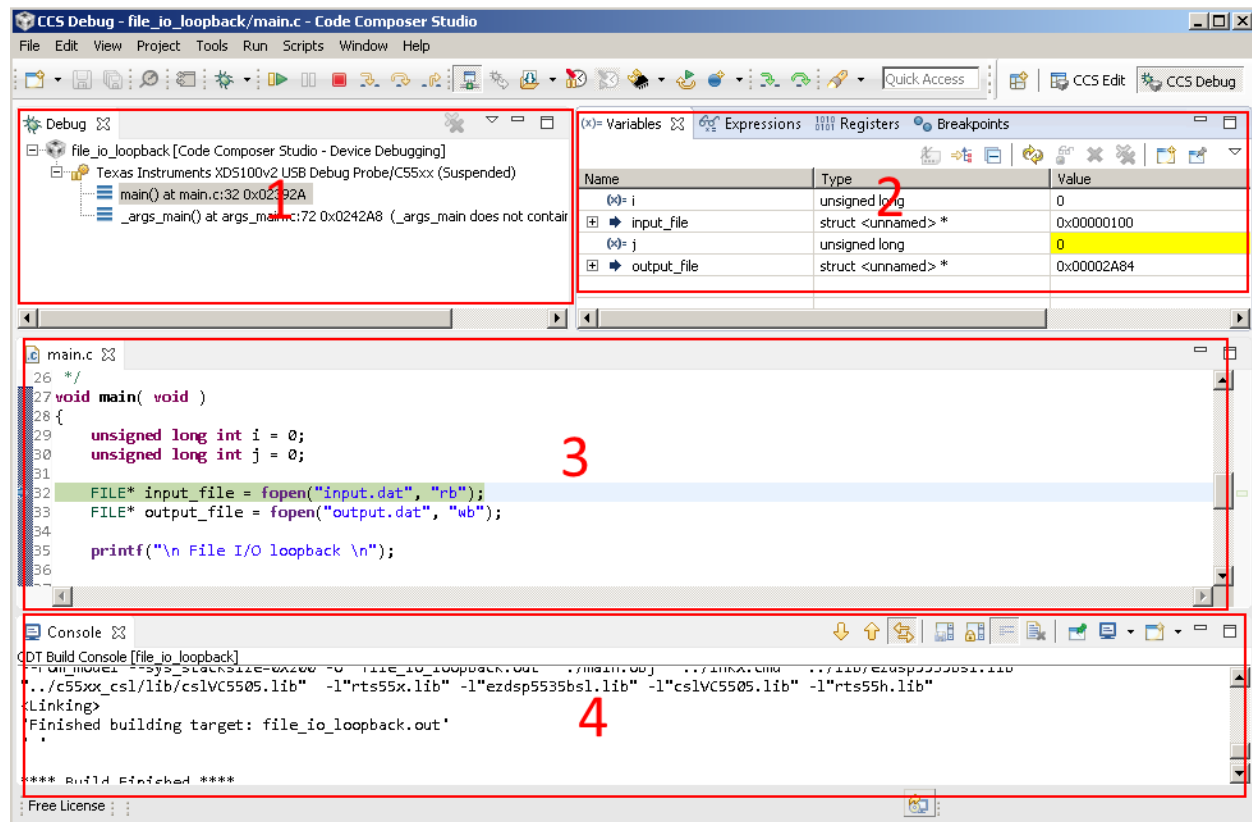
Nakon pokretanja programa CCS nudi prelazak u *Debug* perspektivu. U okviru *Debug* perspektive prikazan je set alata za kontrolisano izvršenje programa.

- **Zadatak:**
  - Pokrenuti izvršenje *Hello World* programa.
  - Nastaviti izvršenje programa do kraja pritiskom na dugme .
  - Proveriti ispis u konzoli.
  - Prekinuti izvršenje pritiskom dugmeta .

#### 4.8 Osnovni alati za kontrolisano izvršavanje programa

Debug perspektiva u okviru CCS okruženja poseduje podrazumevani skup otvorenih prikaza i njihov raspored. Svaki od prikaza moguće je ukloniti ili promeniti njegovu poziciju. Konfiguracija izgleda perspektive vezana je za radni prostor. Ponovno otvaranje uklonjenih prikaza ili otvaranje novih vrši se iz menija *Window*->*Show View*.





Slika 13 - Debug perspektiva (1-Debug View, 2-Variables View, 3-Source code View, 4-Console)

#### 4.8.1 Prikaz informacija o kontrolisanom izvršenju

Prikaz informacija o kontrolisanom izvršenju (*Debug View*) sadrži informacije o ciljnoj platformi na kojoj se program izvršava, prikazuje trenutno stanje procesorskih jezgara ciljne platforme, kao i stek poziva. Odabirom funkcije iz steka poziva, otvara se izvorni kod te funkcije na datoj lokaciji unutar prozora za prikaz izvornog koda.

#### 4.8.2 Prikaz izvornog koda

Prikaz izvornog koda (*Source code View*) sadrži prikaz izvornog koda programa koji se trenutno izvršava. Ukoliko je izvršavanje koda na ciljnoj platformi pauzirano, plavom strelicom sa leve strane označena je naredna instrukcija koja će biti izvršena.

#### 4.8.3 Alati za kontrolu toka izvršavanja programa

Traka sa alatima za kontrolu toka izvršavanja programa nalazi se iznad prikaza informacija o kontrolisanom izvršenju.



Slika 14 - Alati za kontrolu izvršavanja programa

Opcije koje nudi su redom:

- Pokreni/nastavi izvršavanje

- Pauziraj izvršavanje
- Prekini sesiju kontrolisanog izvršavanja programa
- Ulazak u funkciju iz trenutno izvršavane linije (*Step Into*)
- Izvršavanje trenutne linije bez ulaska u funkciju (*Step Over*)
- Izlazak iz funkcije i povratak na mesto poziva funkcije
  
- **Zadatak:**
  - Izmeniti primer Hello World tako što ćete dodati funkciju *hello()* koja ispisuje pozdravnu poruku na ekran. Funkciju pozvati iz funkcije *main()*.
  - Prevesti i pokrenuti program
  - Koračati kroz program do linije poziva funkcije *hello()*
  - Ući unutar funkcije *hello()*
  - Koračati do poziva funkcije *printf*
  - Pokušati ući unutar funkcije *printf*. Šta se dešava? Zbog čega?

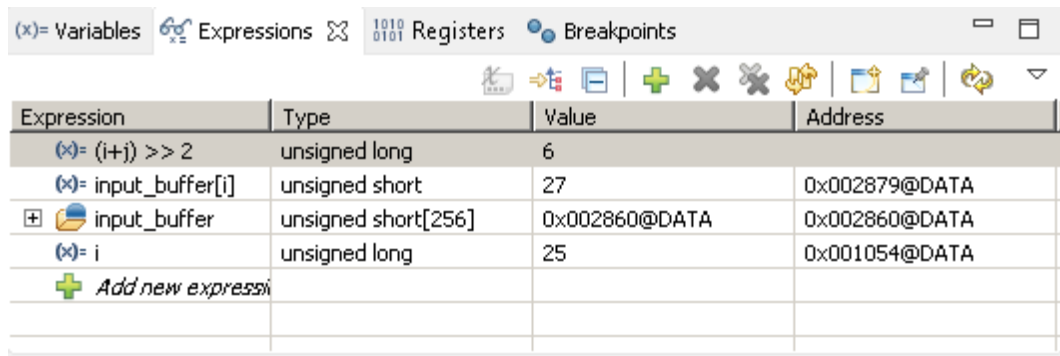
#### 4.8.4 Prikaz promenljivih

Prikaz promenljivih (*Variables View*) sadrži informacije o promenljivima koje su žive u toku izvršenja trenutne instrukcije. Prikaz sadrži naziv, tip i trenutnu vrednost promenljive.

- **Zadatak:**
  - Pokrenuti izvršenje *file\_io\_loopback* projekta
  - Ukoliko nije otvoren, otvoriti prikaz promenljivih
  - Koračati kroz kod i pratiti kako se vrednost promenljivih menja nakon izvršenih instrukcija
  - Koračati do *for* petlje koja ide po *i*.
  - Promeniti vrednost promenljive *i* (dvostruki klik na vrednost u prikazu promenljivih) tako da *i* bude veće od vrednosti *BUFFER\_LENGTH* (256).
  - Koračati dalje kroz program.

#### 4.8.5 Prikaz izraza

Prikaz izraza (*Expressions View*) predstavlja pregled izraza definisanih od strane korisnika. Izrazi mogu da uključuju promenljive i registre. Dodavanje novog izraza vrši se odabirom opcije *Add new expression* i upisom izraza ili naziva promenljive.



| Expression           | Type                | Value         | Address       |
|----------------------|---------------------|---------------|---------------|
| (x)= (i+j) >> 2      | unsigned long       | 6             |               |
| (x)= input_buffer[i] | unsigned short      | 27            | 0x002879@DATA |
| + input_buffer       | unsigned short[256] | 0x002860@DATA | 0x002860@DATA |
| (x)= i               | unsigned long       | 25            | 0x001054@DATA |
| + Add new expression |                     |               |               |

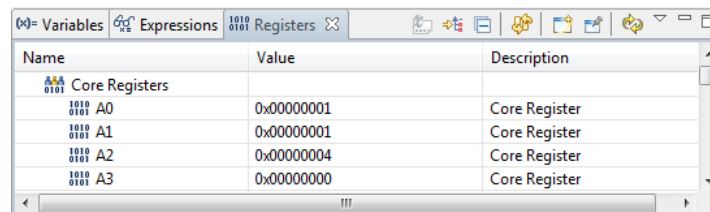
Slika 15 - Prikaz izraza

- **Zadatak:**

- Uneti naziv neke od promenljivih unutar vašeg projekta u prikaz izraza
- Uneti matematički izraz koji koristi minimum dve promenljive
- Uneti izraz koji predstavlja indeksiranje 23-eg elementa *output\_buffer* niza

#### 4.8.6 Prikaz registara

Prikaz registara (*Registers View*) omogućava pregled sadržaja registara procesorskog jezgra i perifernih registara uređaja u toku izvršenja programa.



| Name           | Value      | Description   |
|----------------|------------|---------------|
| Core Registers |            |               |
| A0             | 0x00000001 | Core Register |
| A1             | 0x00000001 | Core Register |
| A2             | 0x00000004 | Core Register |
| A3             | 0x00000000 | Core Register |

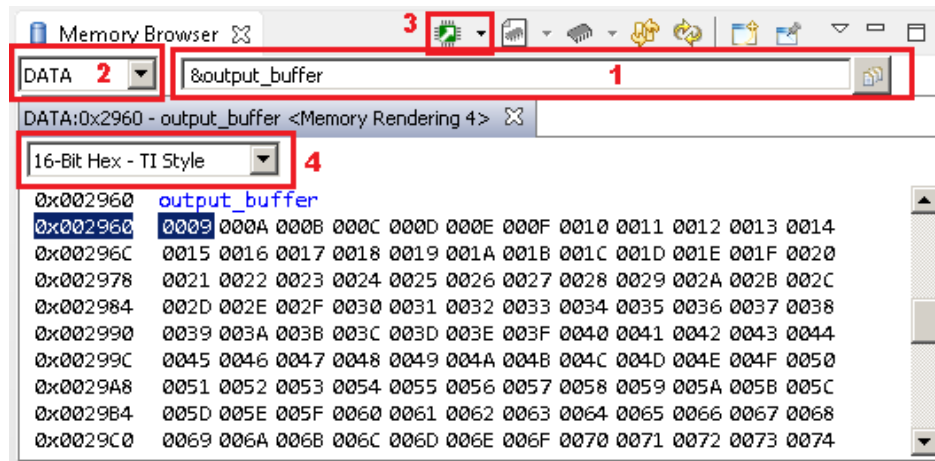
Slika 16 - Prikaz registara

#### 4.8.7 Prikaz memorije

Prikaz memorije (*Memory View*) nije otvoren u podrazumevanim podešavanjima Debug perspektive. Otvaranje ovog prikaza vrši se odabirom opcije *View --> Memory Browser*.

Prikaz memorije omogućava pregled sadržaja RAM memorije, i prikaz vrednosti različitim formatima (znakovi, celobrojni (označeni/neoznačeni), hexadecimalni, *float*). Prikaz željene memorijske lokacije vrši se zadavanjem memorijskog prostora i početne adrese. Moguće je zadati adresu kao tačnu vrednost ili izraz.

Korišćenjem ovog prikaza moguće je menjati vrednosti u memoriji DSP uređaja. Moguće je sačuvati skup vrednosti u datoteku na računaru na kom je CCS pokrenut i učitati vrednosti iz datoteke koja je ranije sačuvana. Moguće je i izvršiti popunjavanje memorije odabranom vrednošću.



Slika 17 - Prikaz memorije (1-početna adresa, 2-adresni prostor, 3-opcije za upis/čitanje/popunjavanje memorije, 4-format prikaza vrednosti)

- **Zadatak:**

- Pokrenuti prikaz memorije (*View->Memory Browser*)
- Za adresu postaviti adresu početka *output\_buffer* niza (za globalne promenljive iznad adrese početka promenljive stoji labela sa nazivom promenljive)
- Za tip podataka odabrati celobrojni označeni tip veličine 16 bita
- Popuniti prvih 20 članova niza brojem 23.
- Sačuvati prvih 20 članova niza u datoteku na disku
- Izmeniti ručno iz Memory Browser prikaza prvi i 15-ti element niza *output\_buffer*
- Učitati prethodno sačuvanu datoteku. Da li su izmene nakon čuvanja datoteke poništene?

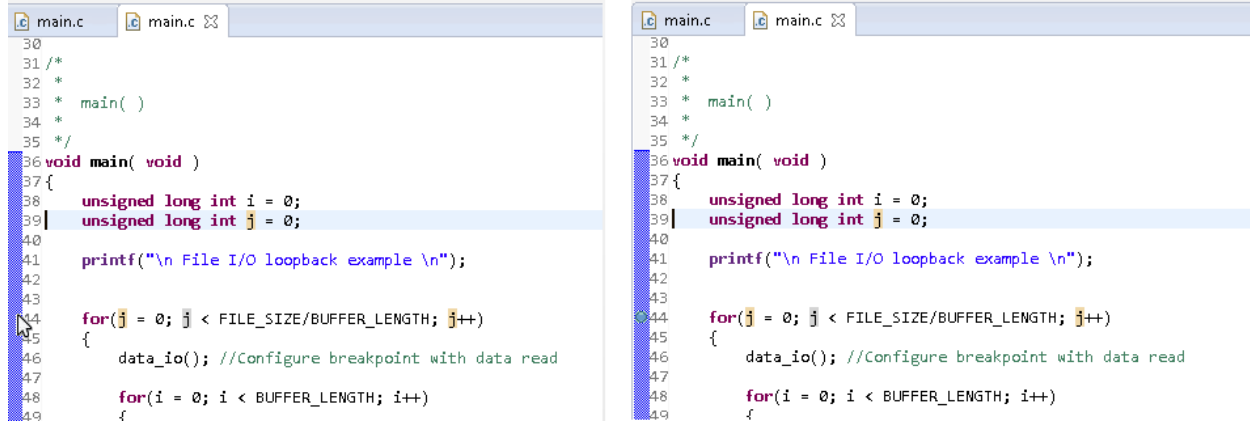
#### 4.8.8 Tačke prekida

*Breakpoints* ili tačke prekida služe za zaustavljanje izvršenja programa na liniji na kojoj su postavljene, odnosno pre izvršavanja prve instrukcije definisane tom linijom. Tačke prekida se mogu, kako pre pokretanja, tako i u toku izvršavanja programa, dodavati, brisati ili privremeno deaktivirati. Postoji i mogućnost postavljanja uslova za zaustavljanje izvršenja programa na liniji na kojoj je postavljena konkretna tačka prekida. Uslovi mogu biti logički ili uslovi vezani za broj zaustavljanja na konkretnoj tački. CCS nudi i definisanje akcije koje će se izvršiti kada izvršavanje programa stigne do tačke prekida.

Dodavanje nove tačke prekida je moguće izvršiti na više načina: dvostrukim levim klikom u prikazu izvornog koda, sa leve strane linije na koju želimo da dodamo tačku prekida, desnim klikom na isto mesto i odabirom opcije *Toggle Breakpoint* iz padajućeg menija ili postavljanjem kursora na željenu liniju i pritiskom na kombinacije tastera *Ctrl+Shift+B*. Dodatu tačku prekida potvrđuje plavi kružić u vertikalnoj liniji, koji se nalazi na svakoj liniji za koju je postavljena tačka prekida.

Nakon dodavanja tačke prekida, program pokrenut sa ciljem kontrolisanog izvršavanja zaustavlja se na željenoj liniji, odnosno pre izvršenja instrukcija koje ona definiše.

Priprema za vežbe iz predmeta Osnovi algoritama i struktura DSP 1  
Vežba 1 – Uvod u digitalnu obradu signala

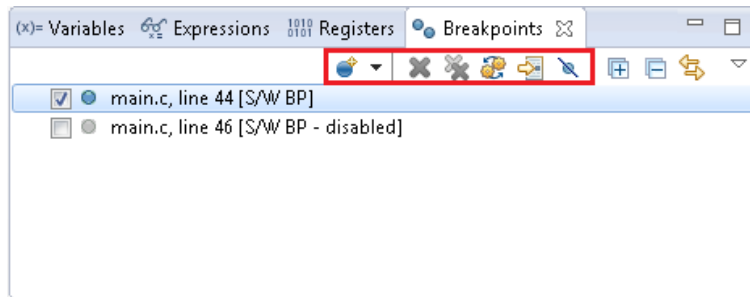


Slika 18 - Dodavanje tačke prekida dvostrukim klikom

Brisanje tačke prekida se vrši na isti način kao i dodavanje.

Privremeno stavljanje van funkcije (deaktiviranje) se vrši desnim klikom na tačku prekida i odabirom opcije *Disable Breakpoint*. Prazan kružić u vertikalnoj sivoj liniji označava postojanje trenutno neaktivne tačke prekida.

Jednostavan pregled postojećih tačaka prekida nudi prikaz tačaka prekida (*Breakpoints View*) koji se otvara preko menija *View-> Breakpoints*. Ovaj prikaz, pored mogućnosti pregledanja i pretrage svih tačaka prekida u projektu, nudi mogućnosti lakog dodavanja, brisanja i deaktiviranja tačaka prekida, brisanja i deaktiviranja svih tačaka prekida i skok na liniju u kodu za koju je određena tačka prekida vezana.



Slika 19 - Breakpoints View

- **Zadatak:**
  - Postaviti tačku prekida na poziv *fwrite* funkcije
  - Nastaviti izvršavanje programa do zadate tačke prekida (Resume/Continue button)
  - Dodati još jednu tačku prekida
  - Pokrenuti izvršavanje programa do sledeće zadate tačke prekida
  - Aktivirati opciju Disable All i pokrenuti nastavak daljeg izvršenja koda.
  - Pauzirati program, isključiti Disable All opciju
  - Isprobati opciju *goto source file*

#### 4.8.9 Posebna podešavanja tačka prekida

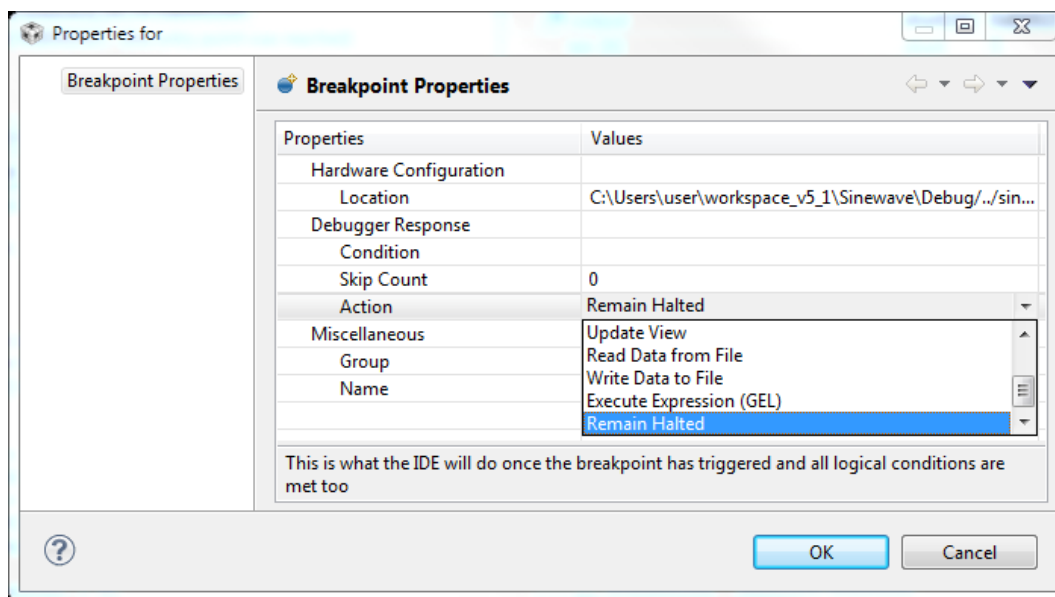
Za svaku tačku prekida moguće je postaviti dodatna podešavanja. Prozor za podešavanje tačke prekida otvara se pritiskom desnog tastera miša na tačku prekida (u prikazu izvršnog koda ili prikazu tačaka prekida) i odabirom opcije *Breakpoint Properties*.

Polje *Condition* predstavlja uslov koji mora biti zadovoljen da bi tačka prekida bila aktivna.

Polje *Skip Count* označava koliko puta je potrebno da se izvrši linija na kojoj je tačka prekida da bi se izvršenje programa zaustavilo (ili desila neka druga definisana akcija). Na primer, ukoliko ovo polje sadrži vrednost 10, svaki deseti prolaz kroz liniju koja je označena ovom tačkom prekida, program će se zaustaviti.

Polje *Action* definiše akciju CCS-a kada izvršenje programa stigne do definisane tačke prekida. Podrazumevana akcije jeste zaustavljanje izvršenja (*Remain halted*). Pored ove akcije moguće je:

- Ažurirati određeni prikaz
- Ažurirati sve prikaze
- Učitati podatke iz datoteke u memoriju uređaja
- Upisati podatke iz memorije u datoteku
- Izvršiti određenu naredbu iz GEL skripte
- Uključi/Isključi grupu tačaka prekida



Slika 20 - Podešavanje tačke prekida

- **Zadatak:**
  - Napraviti tačku prekida unutar *for* petlje po *i*, koja zaustavlja izvršenje programa kada promenljiva *i* ima vrednost 21
  - Nastaviti izvršenje dok se program ne zaustavi na zadatoj tački prekida.
  - Proveriti da li vrednost promenljive *i* odgovara zadatoj.

#### 4.8.10 Vizualizacija podataka

Code Composer Studio okruženje omogućava grafički prikaz podataka koji se nalaze u memoriji DSP uređaja. Ovaj alat nam omogućava iscrtavanje signala u vremenskom i frekventnom domenu.

Pokretanje alata za iscrtavanje grafika vrši se odabirom opcije *Tools -> Graph* i odabirom jedne od ponuđenih opcija.

- Iscrtavanje signala u vremenskom domenu: *Single Time* i *Dual Time*
- Iscrtavanje signala u vremenskom domenu: *FFT Magnitude*, *FFT Magnitude Phase*, *Complex FFT*, *Waterfall FFT*

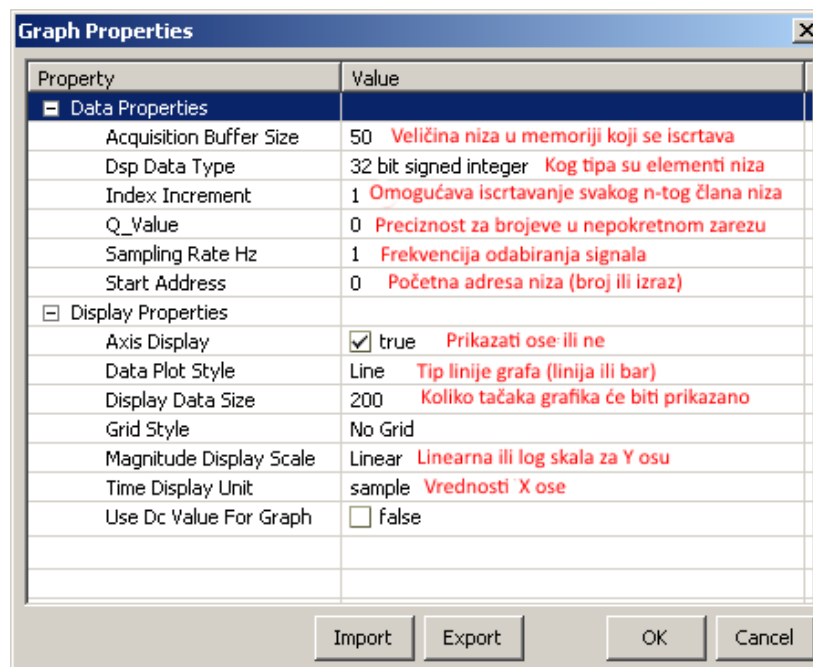
Prikaz svakog od pomenutih alata za iscrtavanje grafika sadrži traku sa alatima za podešavanje tog alata. Pomenuta traka sa alatima uključuje podešavanje kada se vrši osvežavanje podataka koji se isrtavaju (nikada, kontinualno, kada god je procesor ciljnog uređaja pauziran ili ručno), uvećanje i umanjeње prikaza, podešavanje parametara iscrtavanja itd.



Slika 21 - Traka sa alatima za podešavanje grafičkog prikaza podataka

Podrazumevano podešavanje čini osvežavanje kada god je procesor pauziran i korišćenje automatskog skaliranja uvećanja u zavisnosti od vrednosti signala.

Podešavanje parametara za iscrtavanje signala u vremenskom domenu:



Slika 22 - Podešavanje opcija grafika u vremenskom domenu

Podešavanje je moguće sačuvati u datoteku, i ponovo učitati po potrebi.



Većina polja podešavanja grafičkog prikaza u frekventnom domenu je ista kao i u vremenskom. Dodatna polja su:

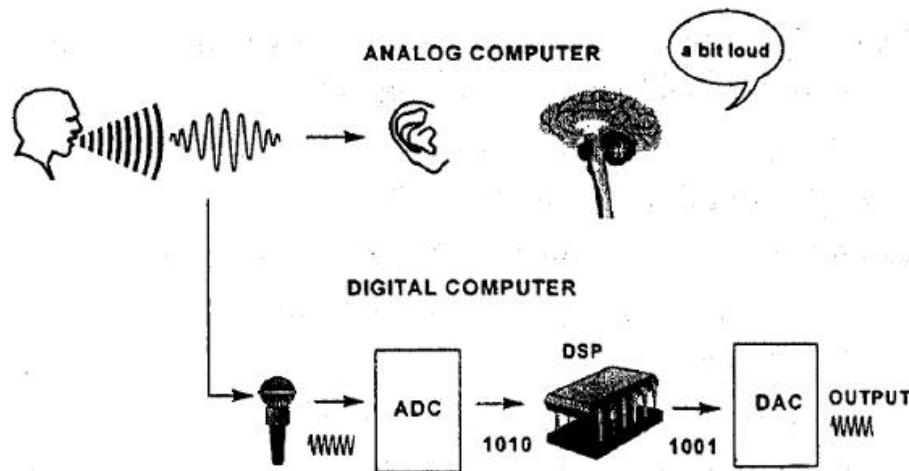
- FFT
  - FFT Frame Size – veličina okvira prilikom računanja FFT-a (jednak je  $2^{\text{FFT Order}}$ )
  - FFT Order – red računanja FFT-a
  - FFT Window function – Korišćena prozorska funkcija

Detaljnije objašnjenje ovih parametara biće predmet jedne od narednih vežbi.

- **Zadatak:**
  - Iscrtati vrednosti output\_buffer niza upotrebom alata *Graph -> Single Time*
  - Voditi računa o parametrima:
    - Veličina niza
    - Tip elemenata niza
    - Broj prikazanih elemenata (prikazati čitav niz na grafiku)
    - Ostali parametri neka zadrže podrazumevane vrednosti
  - Promeniti izgled grafa tako što će se umesto linije koristiti bar graf.

## 5 Obrada signala u realnom vremenu na TMS320C5535

Obrada signala u realnom vremenu podrazumeva učitavanje signala iz prirode, nekakvu obradu nad učitanim signalom i njegovu reprodukciju.



Slika 1 – Sistem za digitalnu obradu signala u realnom vremenu

Kao što znamo, svi signali u prirodi su analogni. Dakle ukoliko želimo da vršimo digitalnu obradu signala, pre toga taj signal moramo nekako predstaviti u diskretnom obliku. U okviru sistema za digitalnu obradu signala, komponenta koja vrši diskretizaciju signala naziva se analogno-digitalni konvertor (ADC). Nasuprot procesa diskretizacije signala postoji i proces za pretvaranje digitalnog signala u analogni, za koji je zadužen digitalno-analogni konvertor (DAC).

Kao što je već pomenuto TMS320C5535 eZdsp razvojna ploča poseduje TLV320AIC3204 programabilni audio kodek koji sadrži *ADC* i *DAC*.

Kako bi bilo moguće koristiti *ADC* i *DAC* potrebno je inicijalizovati TLV320AIC3204 komponentu, kao i pravilno inicijalizovati TMS320C5535 uređaj kako bi se omogućila komunikacija sa periferijama.

Rad sa TLV320AIC3204 kodekom biće objašnjen na primeru.

- **Zadatak:** Uvući projekat *ad\_da\_loopback* u radni prostor.

U okviru ovog projekta kao i prethodnog (*file\_io\_loopback*), nalaze se dve biblioteke i odgovarajuća zaglavlja za pristup funkcijama tih biblioteka:

- C55xx\_csl - *C55x Chip Support Library* pruža korisničku spregu za konfiguraciju i kontrolu DSP uređaja iz C55x familije.
- ez5535bsl - *C5535 ezDSP Board Support Library* predstavlja omot oko *CSL* biblioteke, sadrži funkcije za konfiguraciju i kontrolu DSP uređaja i periferija koje se nalaze na TMS320C5535 *eZdsp* razvojnoj ploči korišćenjem funkcija *CSL* biblioteke.

Funkcije ove dve biblioteke koristimo tako što uvučemo odgovarajuća zaglavlja za željenu periferiju, i zatim pozivamo odgovarajuće funkcije.

Pored ove dve biblioteke u okviru projekta *ad\_da\_loopback* nalaze se datoteke:

- aic3204.h
- aic3204\_init.c
- aic3204.c

Ove tri datoteke sadrže kod potreban za inicijalizaciju i rad sa TLV320AIC3204 kodekom.

Koraci za inicijalizaciju uređaja i kodeka:

- Uključiti zaglavlja za inicijalizaciju razvojne ploče i inicijalizaciju i rad sa kodekom
  - `#include "ezdsp5535.h"`
  - `#include "aic3204.h"`
- U okviru *main* funkcije pozvati funkciju za inicijalizaciju razvojne ploče
  - `Int16 EZDSP5535_init( );`
- Pozvati funkciju za inicijalizaciju AIC3204 hardvera i I<sup>2</sup>C sprege između AIC3204 i DSP procesora
  - `void aic3204_hardware_init(void);`
- Pozvati funkciju za podešavanje AIC3204 kodeka putem I<sup>2</sup>C sprege
  - `void aic3204_init(void);`
- Pozvati funkciju za postavljanje željene frekvencije odabiranja i pojačanja
  - `unsigned long set_sampling_frequency_and_gain(unsigned long, unsigned int);`
  - Napomena: Ukoliko se koristi mikrofonski ulaz, pojačanje postaviti na 30dB

Nakon ovih koraka kodek je inicijalizovan i komunikacija između AIC3204 i DSP procesora je pokrenuta putem I<sup>2</sup>S kanala. Koristeći podrazumevana podešavanja, AIC3204 je konfigurisan tako da učitava i emituje *stereo* signal (dva kanala). Odbirci su predstavljeni kao označene celobrojne vrednosti veličine 16 bita (*Int16*).

Učitavanje odbiraka signala sa ADC konvertora vrši se pozivom funkcije:

- `void aic3204_codec_read(Int16*, Int16*);`

A slanje vrednosti na DAC konvertor pozivom funkcije:

- `void aic3204_codec_write(Int16, Int16);`

Obe funkcije su blokirajuće. Kod čitanja nakon poziva čeka se da podaci budu spremni na izlazu iz ADC konvertora, tek nakon dobavljanja validnih podataka funkcija vraća vrednost. Kod pisanja, čeka se da DAC konvertor bude spreman za upisivanje vrednosti, nakon toga se vrednost upisuje i funkcija se završava. Istovremeno se vrši čitanje i pisanje odbiraka na oba kanala.

U okviru *ad\_da\_loopback* projekta između operacija čitanja i pisanja ne vrši se nikakva obrada nad odbircima. Pročitane vrednosti su direktno prosleđene na izlaz.

- **Zadatak:**
  - Prevesti projekat *ad\_da\_loopback*.
  - Pokrenuti izvršavanje prevedenog programa
- **Zadatak:**
  - Izmeniti *main* funkciju vašeg programa tako što ćete ulazni signal sa levog kanala pomnožiti sa 2, a ulazni signal sa desnog podeliti sa 2.
  - Pokrenuti program i poslušati razliku između dva kanala.

## 6 Zaključak

U okviru ove vežbe upoznali ste se sa razvojnom pločom *TMS320C5535 eZdsp* koja će biti korišćena u daljem toku ovog kursa. Demonstrirana je upotreba radnog okruženja *Code Composer Studio 6*, i alati koje ovo okruženje nudi za razvoj i analizu programske podrške namenjene izvršavanju na *TMS320C5535 eZdsp*. Objašnjen je kroz primer koncept obrade signala u realnom vremenu, kao i način inicijalizacije i upotrebe AD i DA konvertora ugrađenog na korišćenoj razvojnoj ploči. Znanje stečeno u toku izrade ove vežbe predstavlja neophodno predznanje za uspešnu izradu svake od narednih vežbi.