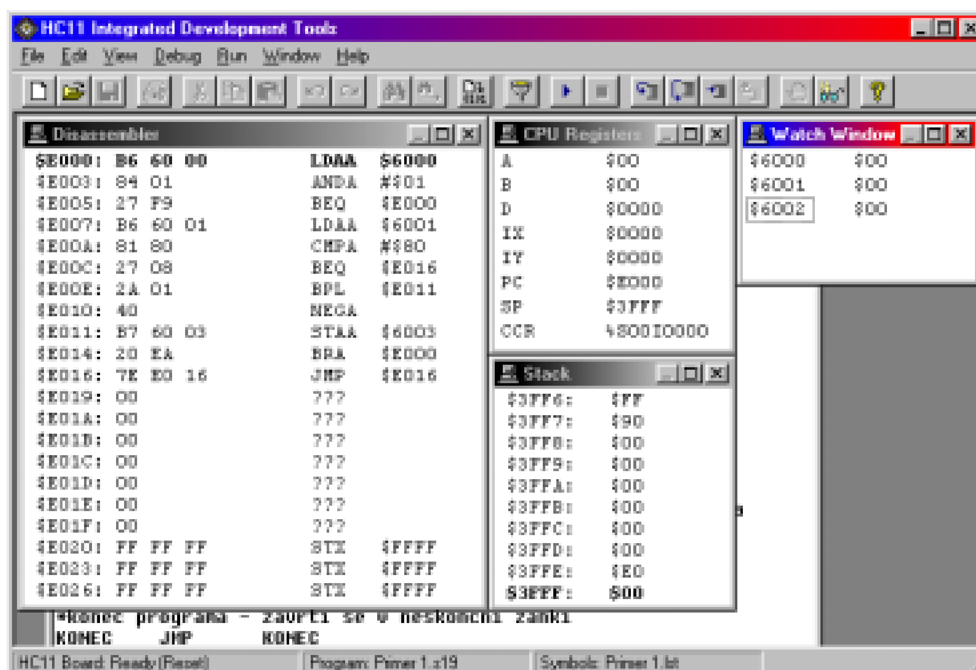
D
Razvojno okolje in simulator

V prejšnjem dodatku smo spoznali strojno opremo razvojnega sistema. Za razvijanje programov pa sama strojna oprema ni dovolj, potrebna je tudi programska oprema, to je razvojno okolje z uporabniškim vmesnikom. Osnovni namen razvojnega okolja HC11IDT je omogočanje enostavne uporabe razvojnega sistema.

Preden lahko razvojno okolje uporabljamo, ga moramo naložiti na računalnik. Okolje samo je dosegljivo (brezplačno) na spletni strani http://www.fe.uni-lj.si/~lrnv/racunalnistvo1/. Na tej strani so dosegljive tudi vse informacije o celotnem razvojnem sistema (tako programski kot strojni opremi). S te strani prenesemo datoteko HC11IDT.ZIP na naš računalnik, jo razširimo in poženemo program SETUP.EXE. Ta nam na računalnik namesti programski paket HC11IDT in ustvari vse potrebne bližnjice. Ko je razvojno okolje enkrat nameščeno na računalnik, ga lahko začnemo spoznavati. Primer uporabniškega vmesnika razvojnega okolja vidimo na sliki D-1.

Uporabniški vmesnik je kljub velikemu številu funkcij, ki jih nudi, razmeroma enostaven. Vse funkcije so hitro dosegljive iz ukazne in orodne vrstice, pomembnejše imajo celo bližnjice. Sam vmesnik je razdeljen na več delov. Čisto zgoraj se nahaja naslovna vrstica (angl. Title bar). Takoj pod

njo je vrstica z menuji (angl. Menu bar), preko katere so dosegljive vse funkcije, ki nam jih okolje nudi. Bližnjice do pogosteje rabljenih funkcij so na orodni vrstici (angl. Toolbar), skrajno na dnu pa je statusna vrstica (angl. Status bar). Statusna vrstica nam posreduje najpomembnejše informacije o stanju razvojnega sistema, predvsem informacijo o stanju ploščice z mikrokrmilniškim učnim sistemom.



Slika D-1 *Uporabniški vmesnik.*

Možna stanja mikrokrmilniškega sistema so: HC11Board: Not Connected (ni priključen), HC11Board: Not Responding (se ne odziva), HC11Board: Running (program v pogonu), HC11Board: Ready-Reset (v resetu), HC11Board: Ready-Break (prekinjen) in HC11Board: Ready - Illegal Opcode (nepravilna koda).

Posebno stanje je simuliran način delovanja (angl. Simulator mode). V njem je vključen simulator mikrokrmilnika. Za zaganjanje programov sicer ne potrebujemo priključenega razvojnega sistema, je pa zato nekoliko okrnjena funkcionalnost sistema.

Na koncu omenimo še delovno površino uporabniškega vmesnika, po kateri razporejamo razna okna, ki jih med razvijanjem in preskušanjem programov potrebujemo (različna okna in njihove funkcije bomo spoznali kasneje na konkretnih primerih).

Razvojno okolje nam poleg samega pisanja in prevajanja programov omogoča tudi poganjanje le-teh na mikrokrmilniški enoti. V to je vključeno tudi koračno izvajanje (izvede se le en ukaz), opazovanje različnih spremenljivk (pomnilniških lokacij) in nenazadnje tudi postavljanje ustavitvenih točk (angl. breakpoints). Uporabo teh funkcij je najlaže prikazati kar na primeru enostavnega programa.

Najprej malo o programu samem. Ta program je nekoliko spremenjena različica programa za izračun absolutne vrednosti, ki smo ga srečali v 6. poglavju. Spremenjen program na začetku primerja vnešeno vrednost z –128 (binarno 10000000) in v primeru enakosti prekine izvajanje (program se zavrti v neskončni zanki). S tem se rešimo napake pri računanju absolutne vrednosti števila –128 in si hkrati zagotovimo možnost, da izvajanje programa prekinemo. Poleg omenjene spremembe smo v program namerno vnesli nekaj napak, da bomo lahko poleg osnovnih spoznali tudi funkcije razvojnega okolja, ki jih uporabljamo za odkrivanje napak, ki so v praksi skoraj vedno sestavni del programov.

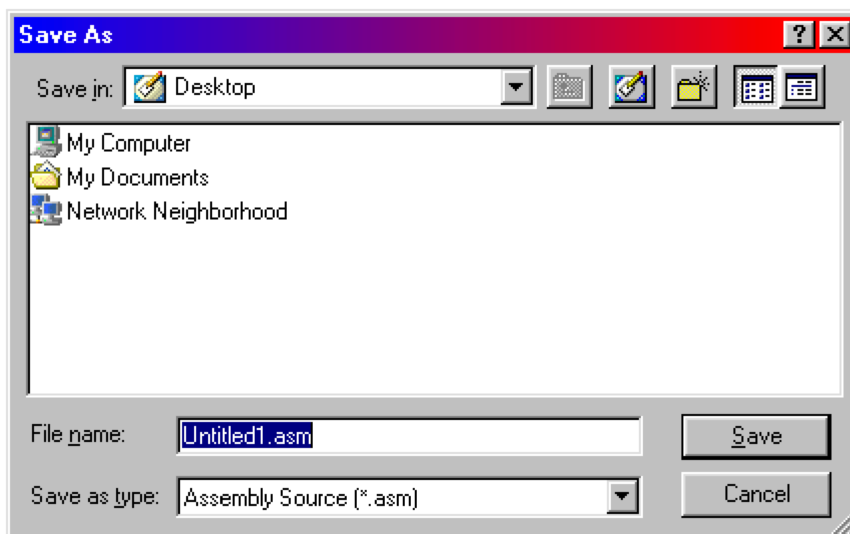
Poglejmo si torej ta program:

```
VHOD_S  EQU  $6000
VHOD_P  EQU  $6001
IZHOD_P EQU  $6003
        ORG  $E000
*branje podatka:
START   LDAA VHOD_S
        ANDA #$10
        BEQ  START
        LDAA VHOD_P
*primerjava z -128:
        CMPA #%10000000
        BEN  KONEC
*izračun absolutne vrednosti:
        CMPA #$00
        BGE  PLUS  naprej, če je nenegativen
        NEGA       sicer dvojiški komplement podatka
*izpis podatka:
PLUS    STAA IZHOD_P
*skok na začetek:
        BRA  STRAT
*konec programa - zavrti se v neskončni zanki
KONEC   JMP  KONEC
*reset vektor:
        ORG  $FFFE
        FDB  START
        END
```

Prvi korak, ki ga moramo narediti, je, da sam program pretipkamo v okno urejevalnika besedila. Še enkrat, program moramo pretipkati točno tako, kot je napisan, *brez popravljanja*! Novo urejevalno okno ustvarimo z ukazom

Slika D-2 *Prevajalnik nas vpraša, če želimo program shraniti.*

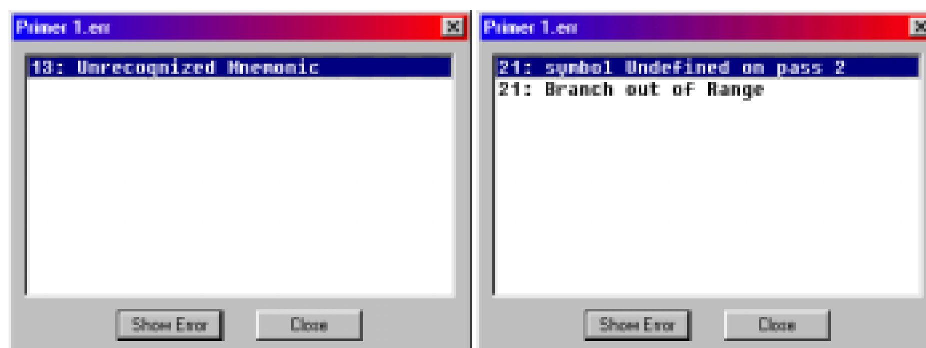
File->New. V to okno pretipkamo naš program (brez nepotrebnih tipkarskih napak, seveda). Ko je program uspešno pretipkan, ga lahko poskusimo prevesti v strojno kodo z ukazom **File->Assemble**. Okolje nas pred prevajanjem vpraša, če želimo natipkano besedilo shraniti v datoteko (slika D-2), kjer kliknemo Yes in v pogovornem oknu, ki ga vidimo na sliki D-3, izberemo mapo in ime datoteke, kamor bomo naš program shranili.



Slika D-3 *Natipkan program moramo shraniti.*

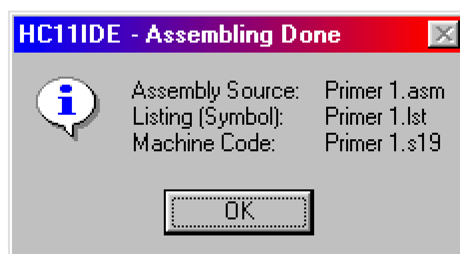
Ko je program varno shranjen, se začne prevajanje. V program smo nalašč pretihotapili nekaj napak, na katere nas prevajalnik opozori. Prva napaka, na katero naletimo (levo okno na sliki D-4), je *13: Unrecognized*

mnemonic. Tako nam prevajalnik skuša dopovedati, da je v vrstici 13 namesto na veljaven mikrokrmilniški ukaz naletel na nepoznano besedo. Ko dotično vrstico programa pozorneje pogledamo, opazimo, da je natipkano BEN. Napako brž popravimo, BEN spremenimo v BNE, in program znova prevedemo. Tudi v drugo ne gre brez zapletov – prevajalnik nam javi kar dve napaki (desno okno na sliki D-4).



Slika D-4 *Prevajalnik nam javlja napake, najprej eno, nato pa kar dve.*

Zopet se poglobimo v nakazano vrstico (vrstica 21), kjer naj bi se nahajal nedefiniran simbol (21: symbol Undefined on pass 2) in pa vejitev izven dosega (21: Branch out of Range). Prvo sporočilo nas opozarja, da smo uporabili še nedefinirano oznako (najverjetneje zopet tipkarska napaka), druga pa, da je cilj vejitve predaleč. Prvo napako je razmeroma lahko najti, saj takoj opazimo, da smo namesto oznake START napisali STRAT. Ko to popravimo, program zopet prevedemo in začuda, napak ni več. Zadnji dve napaki sta bili očitno povezani in je popravek prve odpravil tudi drugo. Sedaj, ko so vse slovnične napake odpravljene, nas prevajalnik pozdravi s sporočilom, da je prevajanje uspelo (slika D-5).



Slika D-5 *Prevajanje je uspelo.*

Sedaj, ko smo program uspešno prevedli, nam ne ostane drugega, kot da ga poženemo. Prevedeni program prenesemo na mikrokrmilniško ploščico z ukazom **File->Download** in ga poženemo z **Run->Run**.

Že ob prvem poskusu vnosa pridemo do grozljivega odkritja - stvar ne deluje tako, kot bi morala, saj vnešeni podatek sploh ni sprejet (LED dioda In_Fresh ne ugasne), kaj šele obdelan. Prišli smo do točke, ko se program sicer uspešno prevede, deluje pa še vedno ne.

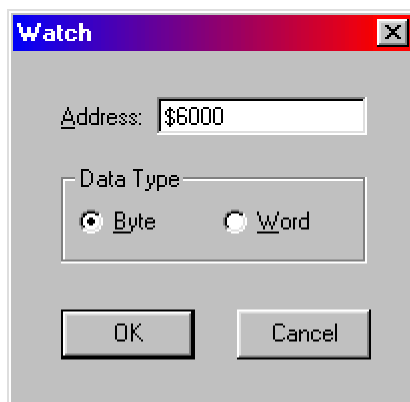
Napakam, ki so preprečevale, da bi se program prevedel, pravimo *sintaktične* napake. Te napake so povezane s sintaktičnimi oziroma slovničnimi pravili zapisovanja programa in jih je relativno enostavno odpraviti, saj nam jih največkrat prevajalnik kar sam pokaže. Mi smo sintaktične napake že odpravili. Napakam, ki se pokažejo šele ob zagonu programa, pravimo *logične* napake. Tem napakam pravimo tudi hrošči (angl. bug), še zlasti, če se uspejo v programu ohraniti do časa njegove distribucije uporabnikom. Odpravljanju logičnih napak oziroma hroščev zato pravimo tudi *razhroščevanje*.

Našega programa se bomo sedaj torej lotili s funkcijami za razhroščevanje, ki nam jih okolje ponuja. Pri uporabi teh funkcij postopamo malo drugače kot pri iskanju sintaktičnih napak. Najprej moramo sami približno predvideti, kje se napake nahajajo. V našem primeru predvidevamo, da je napaka nekje v prvi zanki, kjer čakamo na svež podatek iz vhodne enote. Na sliki D-1 je prikazan uporabniški vmesnik po nalaganju programa v pomnilnik mikrokrmilnika. Ob zagonu okolja se avtomatično odpro okna Disassembler, CPU Registers in Stack. Okna Stack ne rabimo, saj ne šarimo po sistemskem skladu in ga lahko zapremo, po drugi strani pa si odpremo okno Watch (**View->Watch Window**, slika D-6).

Slika D-6 *Okno Watch.*

V tem oknu lahko spremljamo vrednosti spremenljivk (pomnilniške lokacije) med izvajanjem programa. Vrednost, ki nas ta hip zanima, se nahaja na naslovu $6000 (register stanja vhodnega vmesnika) in tako kliknemo **Debug->Add watch** ter v okno vpišemo naslov $6000 (slika D-7). Sedaj lahko v oknu Watch spremljamo vrednost na naslovu $6000. Da

ta funkcija pride do izraza, moramo program izvajati korakoma (angl. single step). Vrednost opazovanih spremenljivk se osveži po vsakem izvedenem koraku, obstaja pa tudi možnost dinamičnega osveževanja spremenljivk. Do tega pridemo, če v oknu Watch kliknemo z desnim gumbom miške in označimo opcijo Auto Refresh (slika D-8).



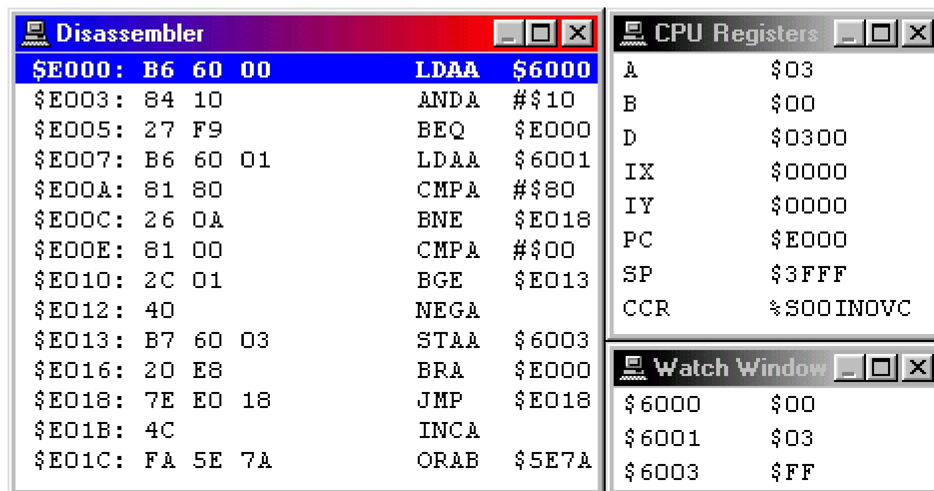
Slika D-7 *Okno Add Watch.*

Slika D-8 *Opcije okna Watch.*

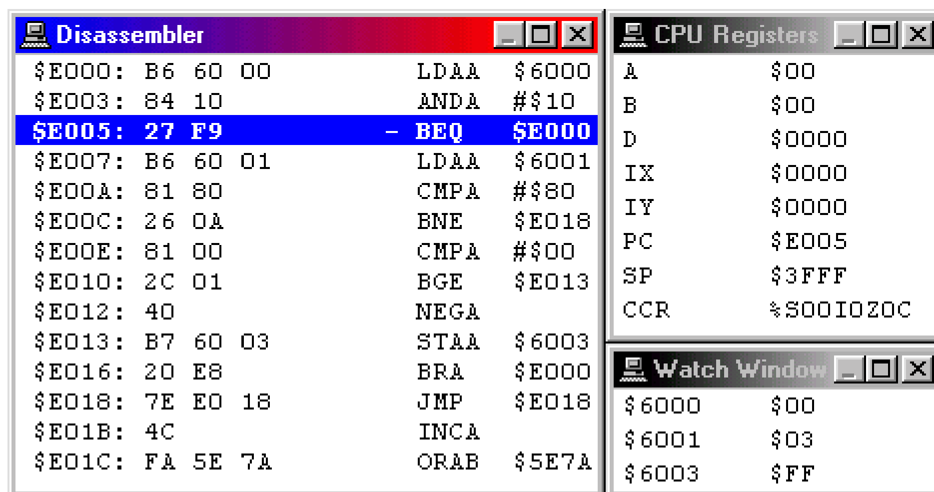
Vrednost spremenljivk se tako preveri in ustrezno osveži nekaj 10-krat na sekundo. V istem oknu kliknemo še Add Watch in v že znanem oknu (slika D-7) dodamo še naslova $6001 in $6003, to je vhodni in izhodni podatkovni register.

Program sedaj naložimo v pomnilnik mikrokrmilnika (**File->Download**), pri čemer se v oknu Disassembler prikaže koda našega programa. Sedaj že lahko začnemo korakoma izvajati program. Kliknemo na okno Disassembler in **Run->Step Into**. Pri tem se kurzor v oknu Disassembler pomakne v

ustrezno vrstico (vrstica je osvetljena). Na slikah D-9 in D-10 so prikazana okna Disassembler, CPU Register in Watch pred in po dveh izvedenih korakih. Pri tem so se ustrezno spremenile tudi vrednosti v ostalih oknih (CPU Registers in Watch). **Run->Step Into** ni edina funkcija, ki izvede samo en korak. Podobno naredi tudi **Run->Step Over.** Razlika med tema funkcijama je v načinu, kako obravnavata podprograme. Medtem, ko Step Into sledi podprogramu (korakoma izvajamo vse ukaze podprograma), Step Over podprogram preskoči, izvede ga kot bi bil en sam ukaz.



Slika D-9 *Stanje tik po nalaganju programa.*



Slika D-10 *Stanje po dveh korakih programa.*

Kot dodatno pomoč nam okolje samo izračuna smer pogojne vejitve. Na sliki D-10 je v osvetljeni vrstici pred ukazom BEQ narisan -, kar nam nakazuje, da je cilj vejitve pred to vrstico. Podobno, če bi bil namesto - narisan +, bi bil cilj po tej vrstici. V primeru, da pred ukazom ni ne + ne - se vejitev ne izvrši (pogoj ni izpolnjen), če pa sta hkrati + in -, je cilj vejitve kar ista vrstica (glej slike D-11, D-12, D-13 in D-14). Ta pomoč, četudi se na prvi pogled zdi nepomembna, nam lahko zelo olajša delo, saj nam ni treba peš računati pogojev za skok – to za nas naredi okolje samo.

```
$E003: 84 01            ANDA  #$01
$E005: 27 F9          - BEQ   $E000
$E007: B6 60 01         LDAA  $6001
```

Slika D-11 *Vejitev nazaj*.

```
$E003: 84 01            ANDA  #$01
$E005: 27 06          + BEQ   $E00D
$E007: B6 60 01         LDAA  $6001
```

Slika D-12 *Vejitev naprej*.

```
$E003: 84 01            ANDA  #$01
$E005: 27 F9            BEQ   $E000
$E007: B6 60 01         LDAA  $6001
```

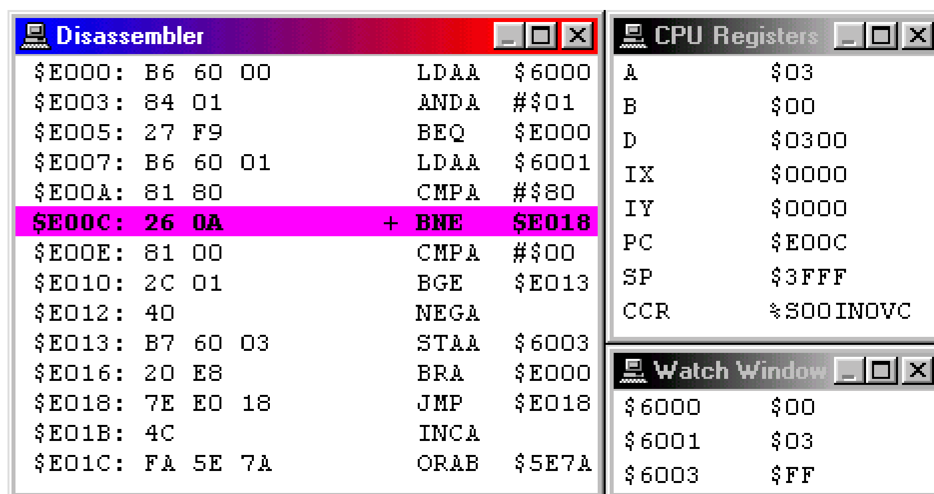
Slika D-13 *Ni vejitve, pogoj ni izpolnjen*.

```
$E001: 01               NOP
$E002: 7E E0 02       ± JMP   $E002
$E005: 01               NOP
```

Slika D-14 *Skok nase*.

Če poskušamo izvesti še nekaj korakov programa (nekajkrat pritisnimo **Run->Step Into**), ugotovimo, da se program vrti v zanki, kar je tudi pravilno, saj do sedaj še nismo vnesli nobene vrednosti v vhodno enoto. Vtipkajmo v vhodno enoto neko vrednost (poljubno) in pritisnimo tipko In_Ack (s tem potrdimo vnos). Takoj ko izvedemo korak, vidimo, da se vrednost na naslovu $6000 v oknu Watch spremeni na $01, kar pomeni, da nas v podatkovnem registru vhodne enote čaka svež podatek. V tem primeru bi se morala zanka zaključiti, vendar se tudi po večkratnem korakanju to ne zgodi. V tej zanki čakamo na postavitev LSB bita v registru stanja vhodnega vmesnika. Kako torej, da se zanka ne zaključi, četudi je bit postavljen? Zanko si poglejmo podrobneje v originalnem zbirnem programu. Stanje

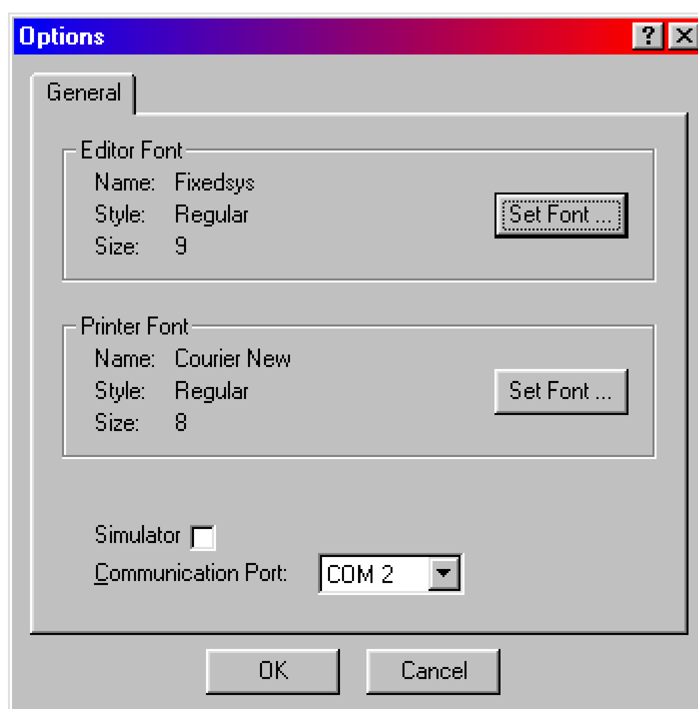
spodnjega bita registra preverjamo s tem, da izvedemo logični IN med vrednostjo registra, ki smo jo prenesli v akumulator A, in masko, ki bi morala biti 00000001 (rezultat bo od 0 različen le, če je spodnji bit operanda 1). Po podrobnejši preučitvi zanke, kaj hitro odkrijemo napako – konstanto $01 smo napačno odtipkali kot $10. To popravimo ter program prevedemo in poženemo. Sedaj program dejansko prebere vhodni podatek (lučka In_Ack ugasne), želenega izpisa na izhodni enoti pa vendarle ni – še ena napaka. Poskusimo vnesti še en podatek – lučka In_Ack ostane prižgana, kar nakazuje čakanje mikrokrmilnika v zanki. V našem programu sta zanki samo dve, od teh smo prvo že počistili, druga pa je neskončna (slika D-9, vrstica $E018). V drugi zanki se znajdemo, če vnesemo vrednost –128, kar je pogoj za zaustavitev programa. Ker preverjanje tega pogoja očitno ne deluje pravilno, si ga oglejmo pobližje. Program zopet naložimo v pomnilnik mikrokrmilnika in korakoma privedemo program do točke, kjer se napaka pokaže. Da nam ni treba tolikokrat pritisniti **Run->Step Into**, nam okolje ponuja ustavitvene točke (angl. breakpoints). Te postavljamo v želenih vrsticah z ukazom **Debug->Toggle Breakpoint**. Z miško kliknemo v vrstico z naslovom $E00C in postavimo ustavitveno točko (vrstica se obarva v vijolično, ko odmaknemo kurzor pa ostane rdeča, slika D-15). Sedaj lahko program poženemo z **Run->Run** in sam se bo ustavil v vrstici z ustavitveno točko (te vrstice še ne bo izvedel), nakar lahko program



Slika D-15 *Okno Disassembler s postavljeno ustavitveno točko.*

izvajamo korakoma naprej tako kot prej. S tem se izognemo dolgotrajnemu korakanju, saj se do ustavitvene točke program odvija s polno hitrostjo mikrokrmilnika. Sedaj nas zanima vsebina akumulatorja A.

Poglejmo si torej stanje, ko dosežemo ustavitveno točko. V akumulatorju A je vrednost $03, kar je vse prej kot $80, ki je postavljena kot pogoj za ustavitev programa. Nekaj mora biti narobe z vejitvijo. Uporabljeni vejitveni ukaz je BNE, za kar po kratkem brskanju po tabeli ukazov ugotovimo, da pomeni vejitev, če je rezultat različen od nič. To ni ravno to, kar hočemo, saj mi iščemo enakost. Po ponovni poglobitvi v tabelo ukazov pridemo na dan z vejitvijo BEQ, ki veji izvajanje programa, če je rezultat enak nič. Če pogledamo originalni program, vidimo, da smo prvotno napisali BEN KONEC, kar smo brez premisleka popravili v BNE KONEC. Očitno smo se zatipkali malce drugače, kot smo predvideli, saj se pravilno popravljena vrstica glasi BEQ KONEC. Ostane nam le še, da tako popravljen program prevedemo, prenesemo na mikrokrmilniško ploščico in ga zaženemo. Program sedaj deluje pravilno!



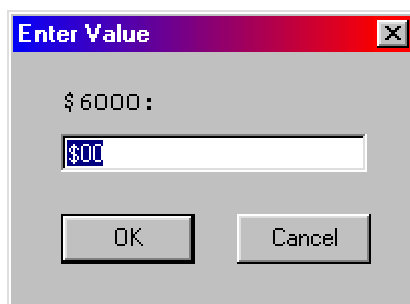
Slika D-16 *Okno Options*.

Zadnja napaka nas privede do naslednjega zaključka – ne le pri odpravljanju logičnih, temveč tudi pri odpravljanju sintaktičnih (pravopisnih) napak, moramo imeti v mislih želeno delovanje programa!

Že s tem kratkim primerom smo videli, kako lahko se nam v program prikradejo napake, kaj šele, če bi bil naš program dolg na stotine vrstic! Resnega programiranja si skoraj ne moremo zamisliti brez funkcij za razhroščevanje, ki smo jih tu spoznali.

Poudariti moramo, da smo podani primer preskušali z uporabo ploščice z učnim sistemom, kar pa ni edini način. Razvojno okolje namreč vsebuje tudi simulator mikrokrmilniškega sistema. Med uporabo simulatorja ali emulatorja (ploščice z učnim sistemom) izbiramo v menuju **View->Options** (slika D-16).

Simulator je sicer enostaven, saj je poleg procesorskega jedra simulirana le najnujnejša periferija (SCI - asinhroni komunikacijski vmesnik), vendar bi naš program lahko preskusili in zagnali tudi z njegovo pomočjo, to je brez priključenega mikrokrmilniškega sistema in vhodno-izhodne enote. Edina razlika, ki se pojavi, je, da bi morali ročno vnesti ustrezne vrednosti na naslove registrov vhodnega vmesnika (slika D-17).



Slika D-17 *Ročni vnos vrednosti na naslov v pomnilniku.*

Vsebino pomnilniške lokacije ročno nastavimo tako, da dvakrat kliknemo na ustrezen naslov v oknu Watch, da se nam odpre okno za vnos vrednosti. Z ročnim vnašanjem simuliramo vhodno-izhodno enoto. Rezultate take simulacije je treba vedno jemati z določeno mero skeptičnosti, saj predpostavljamo, da enota deluje pravilno, kar pa je v realnem svetu vse prej kot samo po sebi umevno.

Pri prejšnjem primeru smo spoznali osnovne prijeme za razhroščevanje, na naslednjem primeru pa si bomo podrobneje ogledali razliko med funkcijama **Run->Step Into** in **Run->Step Over**. Omenili smo že, da se razlika pojavi v primeru podprogramov, ki se izvedejo na dva različna

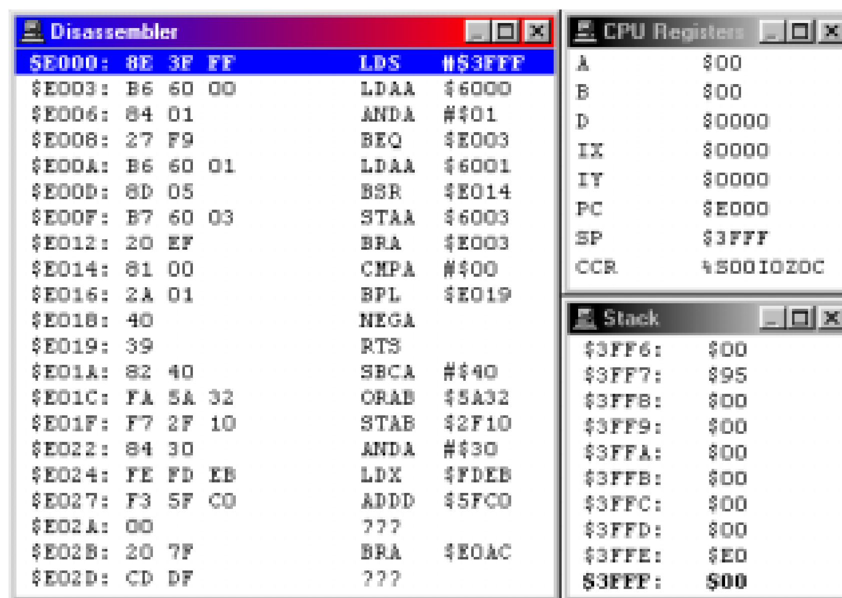
načina. V ta namen prejšnji program za izračun absolutne vrednosti spremenimo tako, da izračun absolutne vrednosti zapremo v podprogram ABS. Razen tega je bistvena razlika tudi v vrstici z oznako START. V tej vrstici z ukazom LDS #$3FFF nastavimo skladovni kazalec na vrednost $3FFF, ki je naslov dna sistemskega sklada. Ta korak je bistvenega pomena, če v programu uporabljamo podprograme. Program je tokrat brez napak, saj je namenjen le prikazu mehanizmov klicanja podprogramov.

Spremenjen program izgleda takole:

```
VHOD_S   EQU  $6000
VHOD_P   EQU  $6001
IZHOD_P EQU  $6003

         ORG  $E000
START    LDS  #$3FFF
BERI     LDAA VHOD_S
         ANDA #$01
         BEQ  BERI
         LDAA VHOD_P
*klic podprograma
         BSR  ABS
         STAA IZHOD_P
         BRA  BERI

*podprogram za izračun absolutne vrednosti
ABS      CMPA #$00
         BPL  PLUS
         NEGA
PLUS     RTS

         ORG  $FFFE
         FDB  START
         END
```

Zopet moramo program pretipkati v urejevalnik. Najprej zapremo vsa okna in z **File->New** odpremo sveže okno urejevalnika. Program pretipkamo in pazimo, da ne vnesemo kake napake. Postopek je enak kot v primeru prvega programa. Pravilno pretipkani program prevedemo s **File->Assemble**. Zopet nas okolje vpraša ali želimo program shraniti in moramo vnesti lokacijo in ime datoteke, v katero želimo program shraniti (sliki D-2 in D-3). Če se pri pretipkavanju nismo zmotili, se program uspešno prevede (slika D-5) in ga lahko naložimo v pomnilnik mikrokrmilniške ploščice (**File->Download**). Po nalaganju se zopet odprejo okna Disassembler, CPU Registers in Stack (slika D-18). Okna stack sedaj

ne zapiramo, saj nas zanima, kaj se dogaja s skladom med koračnim izvajanjem programa.



Slika D-18 *Uspešno naložen program.*

Poskusimo izvršiti nekaj ukazov programa! Nekajkrat kliknimo **Run->Step Into** in opazujmo okno Stack. Takoj opazimo, da je ena vrstica poudarjena (odebeljena pisava). Ta vrstica predstavlja trenutni vrh sklada in vanjo je usmerjen skladovni kazalec. Ker na sklad še nismo odložili nič, je vrh sklada še vedno na naslovu $3FFF, kakor smo ga nastavili na začetku. Zanimivo pri tem je, da se vrednost nad vrhom sklada spreminja, še več, ta vrednost sovpada z naslovom ukaza, ki se mora izvršiti (je osvetljen v oknu Disassembler). Kaj takega z našim znanjem o delovanju sklada nismo pričakovali, saj bi sklad moral ostati nespremenjen, dokler nanj česa ne odložimo, oziroma, dokler ne pokličemo podprograma. To nenavadno obnašanje sklada si lahko pojasnimo, če se zavemo, da se na mikrokrmilniškem sistemu izvaja BIOS, program, ki skrbi za komunikacijo med strojno opremo in programskim razvojnim okoljem. Ta izkorišča vrh sklada za svoje delovanje, pri tem pa skrbi, da ne poškoduje uporabnikovih podatkov na skladu. To seveda dobro deluje, vrednosti, ki se na skladu pojavljajo, so vedno nad vrhom sklada in nam ne povzročajo nobenih težav. Podatki, ki jih na sklad odložimo, te vrednosti enostavno prepišejo.

Sedaj, ko smo to podrobnost razčistili, se posvetimo našemu primeru. Pri tem programu nas zanima predvsem, kaj se dogaja s skladom med klicem

podprograma, zato postavimo ustavitveno točko (**Debug->Toggle Breakpoint**) na ukaz, ki podprogram kliče, to je BSR ABS na naslovu $E00D (slika D-19). Ko je ustavitvena točka postavljena, lahko program poženemo z **Run->Run**, saj se bo ta sam ustavil, ko bo prispel do ustavitvene točke (slika D-20).

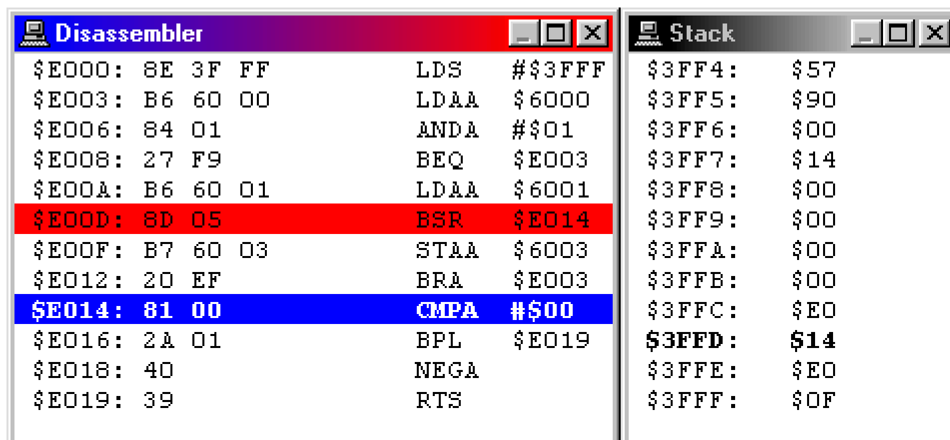
```
Disassembler
$E000: 8E 3F FF          LDS   #$3FFF
$E003: B6 60 00          LDAA  $6000
$E006: 84 01             ANDA  #$01
$E008: 27 F9             BEQ   $E003
$E00A: B6 60 01          LDAA  $6001
$E00D: 8D 05             BSR   $E014
$E00F: B7 60 03          STAA  $6003
$E012: 20 EF             BRA   $E003
$E014: 81 00             CMPA  #$00
$E016: 2A 01             BPL   $E019
$E018: 40                NEGA
$E019: 39                RTS
```

Slika D-19 *Postavljena ustavitvena točka.*

```
Disassembler                                Stack
$E000: 8E 3F FF          LDS   #$3FFF       $3FF4:  $57
$E003: B6 60 00          LDAA  $6000        $3FF5:  $90
$E006: 84 01             ANDA  #$01         $3FF6:  $00
$E008: 27 F9             BEQ   $E003        $3FF7:  $90
$E00A: B6 60 01          LDAA  $6001        $3FF8:  $00
$E00D: 8D 05             BSR   $E014        $3FF9:  $14
$E00F: B7 60 03          STAA  $6003        $3FFA:  $00
$E012: 20 EF             BRA   $E003        $3FFB:  $00
$E014: 81 00             CMPA  #$00         $3FFC:  $00
$E016: 2A 01             BPL   $E019        $3FFD:  $00
$E018: 40                NEGA               $3FFE:  $E0
$E019: 39                RTS                $3FFF:  $0D
```

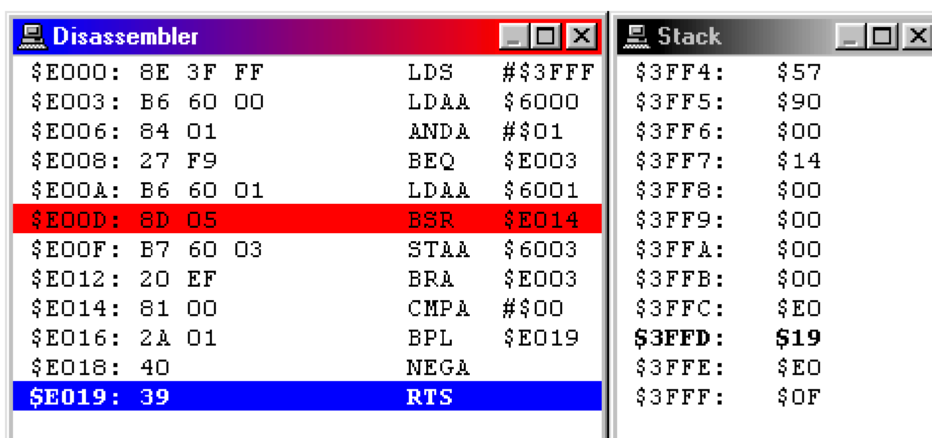
Slika D-20 *Stanje tik pred klicem podprograma.*

Do tega trenutka je skladovni kazalec še vedno postavljen na naslov $3FFF. Izvedimo še en korak z **Run->Step Into**. Znajdemo se v podprogramu, kar kaže slika D-21. Skladovni kazalec se je pomaknil za 2 bajta navzgor (njegova vrednost je sedaj $3FFD), pod vrhom sklada pa se nahaja naslov ukaza STAA $6003, to je $E00F.



Slika D-21 *Nahajamo se v podprogramu.*

Naredimo še tri korake, tako da bo kurzor na ukazu RTS (naslov $E019, slika D-22).



Slika D-22 *Tik pred povratkom iz podprograma.*

Izvedimo **Run->Step Into** še enkrat in opazujmo sklad. Takoj po izvedenem koraku se skladovni kazalec prestavi zopet na dno sklada ($3FFF), medtem ko se izvajanje programa nadaljuje na naslovu $E00F, ki se je ob klicu podprograma shranil na sklad (slika D-23).

Tako sledenje klicem podprogramov je koristno, če želimo vedeti, kaj se v podprogramu dogaja. Če nas interno delovanje podprograma ne zanima, ali če smo prepričani, da deluje pravilno, lahko uporabimo funkcijo **Run->Step Over**. Ta funkcija ravno tako izvede en korak programa, v

primeru klica podprograma pa slednjega izvede kot en sam ukaz. Podprogram se pri tem izvrši s polno hitrostjo. Da bomo videli, kako to v resnici deluje, program zopet poženimo z **Run->Run** in ko se ta ustavi na ustavitveni točki, namesto **Run->Step Into** uporabimo **Run->Step Over**. Stanje po tem koraku vidimo na sliki D-24. Tudi brez posebnih sposobnosti opazovanja lahko ugotovimo, da je stanje programa identično kot po povratku iz podprograma na sliki D-23.

```
Disassembler                                      Stack
$E000: 8E 3F FF          LDS   #$3FFF             $3FF4:   $57
$E003: B6 60 00          LDAA  $6000              $3FF5:   $90
$E006: 84 01             ANDA  #$01               $3FF6:   $00
$E008: 27 F9             BEQ   $E003              $3FF7:   $90
$E00A: B6 60 01          LDAA  $6001              $3FF8:   $00
$E00D: 8D 05             BSR   $E014              $3FF9:   $14
$E00F: B7 60 03          STAA  $6003              $3FFA:   $00
$E012: 20 EF             BRA   $E003              $3FFB:   $00
$E014: 81 00             CMPA  #$00               $3FFC:   $00
$E016: 2A 01             BPL   $E019              $3FFD:   $00
$E018: 40                NEGA                     $3FFE:   $E0
$E019: 39                RTS                      $3FFF:   $0F
```

Slika D-23 *Po povratku iz podprograma.*

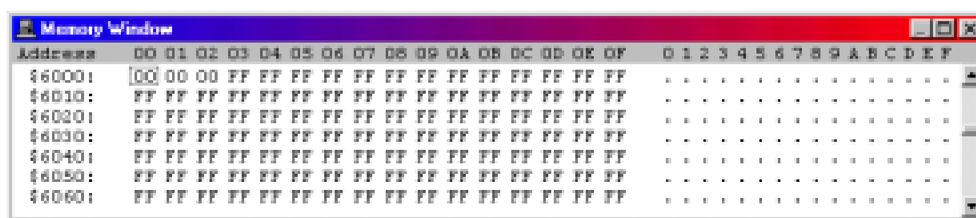
```
Disassembler                                      Stack
$E000: 8E 3F FF          LDS   #$3FFF             $3FF4:   $57
$E003: B6 60 00          LDAA  $6000              $3FF5:   $90
$E006: 84 01             ANDA  #$01               $3FF6:   $00
$E008: 27 F9             BEQ   $E003              $3FF7:   $90
$E00A: B6 60 01          LDAA  $6001              $3FF8:   $00
$E00D: 8D 05             BSR   $E014              $3FF9:   $14
$E00F: B7 60 03          STAA  $6003              $3FFA:   $00
$E012: 20 EF             BRA   $E003              $3FFB:   $00
$E014: 81 00             CMPA  #$00               $3FFC:   $00
$E016: 2A 01             BPL   $E019              $3FFD:   $00
$E018: 40                NEGA                     $3FFE:   $E0
$E019: 39                RTS                      $3FFF:   $0F
```

Slika D-24 *Stanje programa po operaciji* ***Run->Step Over***.

Funkcija **Run->Step Over** je torej enakovredna več zaporednim korakom skozi podprogram. Poleg te funkcije obstaja še funkcija **Run->Step Out**. Kot že samo ime pove, ta funkcija izstopi iz nečesa, in

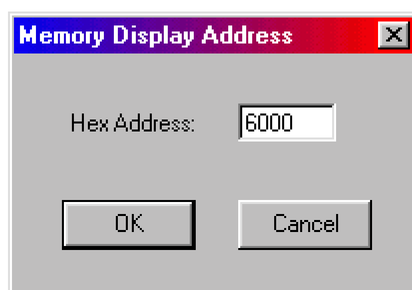
sicer iz podprograma. Uporabljamo jo, kadar nas zanima le začetni del podprograma. Takrat podprogramu sledimo, dokler nas ta zanima, z **Run->Step Out** pa ga zapustimo. Za ilustracijo zopet poženimo naš program z **Run->Run** in na ustavitveni točki izvedimo **Run->Step Into**, da vstopimo v podprogram. Stanje, ki ga vidimo, je enako kot na sliki D-21. Sedaj kliknimo **Run->Step Out**. V trenutku se spet znajdemo nazaj v glavnem programu, kot če bi podprogramu korakoma sledili vse do izhoda (slika D-23), **Run->Step Out** namreč izvede vse ukaze do vključno prvega ukaza RTS, na katerega naleti.

Do sedaj smo za opazovanje vsebin pomnilniških lokacij uporabljali okno Watch. Za spremljanje manjšega števila lokacij je tak način zelo dober, če pa moramo opazovati večje število skupaj postavljenih celic, na primer kakšen medpomnilnik (buffer) ali tabelo, nam takšen pristop ni več udoben. V takih primerih uporabljamo okno Memory (**View->Memory Window**). V tem oknu lahko opazujemo večje področje pomnilnika (slika D-25).



Slika D-25 *Okno Memory.*

V vsaki vrstici je izpisanih 16 pomnilniških lokacij, ki so predstavljene kot šestnajstiške vrednosti (srednji stolpec) oziroma ASCII znaki (desni stolpec), na začetku vrstice pa je naslov prvega bajta v vrstici. Naslov prve opazovane vrednosti nastavimo v oknu Memory Display Address (slika D-26), ki se pokaže takoj ko kliknemo na **View->Memory Window**. Naslov opazovanja pomikamo z uporabo miške (desni Scroll bar), tipkovnice ali tako, da z desnim gumbom kliknemo v okno in izberemo Display Address (slika D-27). Za razliko od okna Watch se tu vrednosti ne osvežujejo samodejno, osvežijo se le ob vsakem kliku na okno Memory. Dodatna slabost tega okna je tudi velika količina podatkov, ki nam jih prikaže. Da se dokopljemo do za nas zanimivih podatkov, je potrebnega kar nekaj iskanja, še posebej, če je v igri več pomnilniških celic, ki so tesno skupaj. Prava vrednost tega pristopa se pokaže v povezavi s tabelami, ker lahko opazujemo tudi več sto bajtov podatkov hkrati (okno Memory lahko povečujemo kot vse ostale).



Slika D-26 *Nastavljanje začetnega naslova opazovanja pomnilnika.*

Slika D-27 *Sprotno nastavljanje naslova opazovanja.*

Tako kot v oknu Watch, lahko tudi v oknu Memory ročno vnašamo vrednosti. Na želeno lokacijo dvakrat kliknemo ter enostavno vpišemo novo vrednost in ta se že postavi. Vnašamo lahko tako šestnajstiške vrednosti kot tudi ASCII znake (prve v srednjem, druge v desnem stolpcu). Paziti moramo le, da se celica, kamor vnašamo vrednost, nahaja v predelu bralno pisalnega pomnilnika. Če je prostor, v katerega pišemo, nezaseden ali je nanj priključen le bralni pomnilnik, se sprememba ne upošteva. To je tudi razlika med uporabo dejanske strojne opreme in simulatorja. Pri simulatorju je namreč cel naslovni prostor predstavljen kot bralno pisalni pomnilnik in zato lahko nastavljamo vrednosti na poljubne lokacije. Ne bo odveč, če poudarimo, da je treba biti pri takem vnašanju zelo pazljiv, da slučajno ne prepišemo napačnega podatka ali še huje, kar program sam.

S tem smo zaključili pregled najpogosteje uporabljenih funkcij za razhroščevanje, ki jih razvojno okolje ponuja. Vseh funkcij je seveda še več, vendar bi bil opis vseh preobsežen. Zahtevnejši uporabniki bodo našli podrobnejši opis celotnega razvojnega sistema tako strojne kot programske opreme na spletnem naslovu http://www.fe.uni-lj.si/~lrnv/racunalnistvo1/hc11board.pdf.