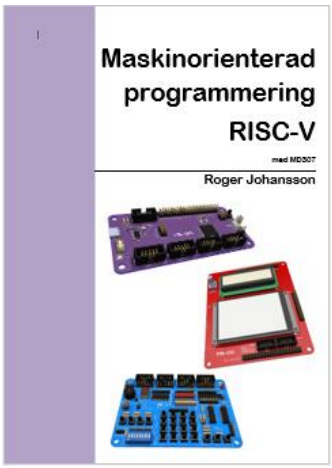


GCC och *SimServer* för *MD307*

inledande övning 3

Under tidigare övningar har vi använt *ETERM8* och *GDB* tillsammans med *SimServer* för att studera test och felavhjälning för assemblerprogram på såväl maskinnivå (*machine level debugging*), som källtextnivå (*source level debugging*).

Efter detta är det nu dags att använda ett komplett system för utveckling, test, felsökning och felavhjälpning i programspråket C. Ramverket för detta (*IDE, Integrated Development Environment*) heter *Codelite* och använder såväl *GCC, GNU C Compiler*, som *GDB*. Eftersom ett sådant ramverk sköter kommunikationen med *GDB* behöver du inte längre själv ge kommandon till *GDB* som du gjorde under inledande övning 2.

INNEHÅLL

Version: 2024-12-01 17:35:00

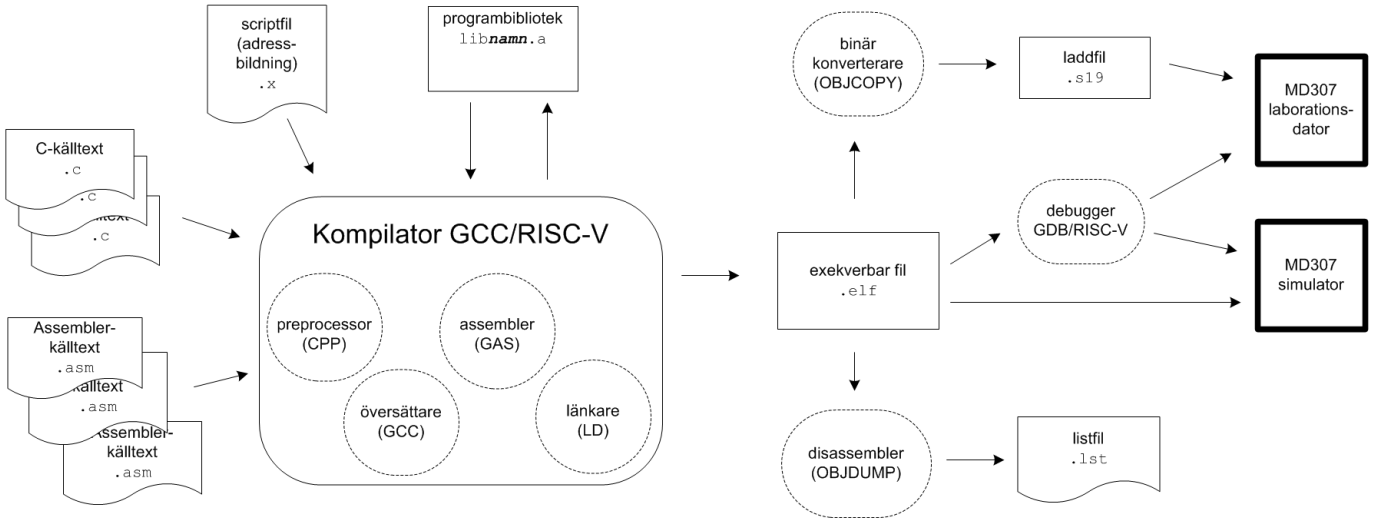
Programutveckling med Codelite

Under denna övning kan du lära dig använda *Codelite*.

- Vi ger först en översikt av programutvecklingsprocessen, de steg och verktyg som används.
- Vi beskriver sedan hur du skapar ett arbetsutrymme (*Workspace*) förbereder detta för användning tillsammans med färdiga projektmallar. Vi beskriver kortfattat projektmallens användning av *konfigurationer*, dvs hur maskinkod skapas för olika exekveringsmodeller (simulator/hårdvara, källtextdebugger osv.).
- Därefter ger vi ett konkret exempel på hur du skapar, kompilerar och länkar (kallas också "bygger") din applikation. I samband med detta visas hur utgår från en färdig projektmall som är speciellt anpassad för laborationsdatorn *MD307*.
- Slutligen visar vi hur du använder *CodeLite* tillsammans med debuggern *GDB* och *SimServer* för att testa och felsöka i dina program.

ÖVERSIKT, VERKTYG OCH UTVECKLINGSPROCESSEN

Följande figur ger en översiktlig bild av programutvecklingsprocessen:



FIGUR 1: PROGRAMUTVECKLINGSPROCESSEN

De olika verktygen finns för olika typer av processorer, följande tabell ger namnen på verktygskedjan för RISC-V.

Verktyg	Generiskt namn	Namn i verktygskedjan för RISC-V
C-kompilator	gcc	riscv32-unknown-elf-gcc
Assemblator	as	riscv32-unknown-elf-as
Länkare	ld	riscv32-unknown-elf-ld
Binärfil konverterare	opjcopy	riscv32-unknown-elf-opjcopy
Dissassembler	objdump	riscv32-unknown-elf-objdump

TABELL 1: GCC VERKTYGSKEDJA FÖR RISC-V

Starta CodeLite

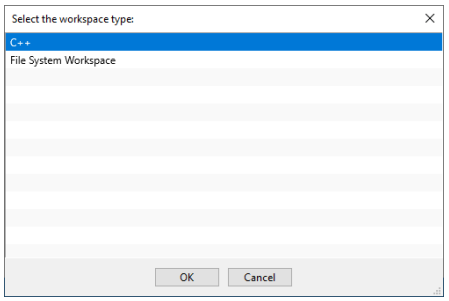
Innan du startar *CodeLite*, försäkra dig om att du följt installationanvisningarna och dessutom installerat:

- GCC för RISC-V
- Anpassade projektmallar (`startup-simple`, `startup` och `startup-crt`) för *MD307*

Nu kan du starta *CodeLite*.

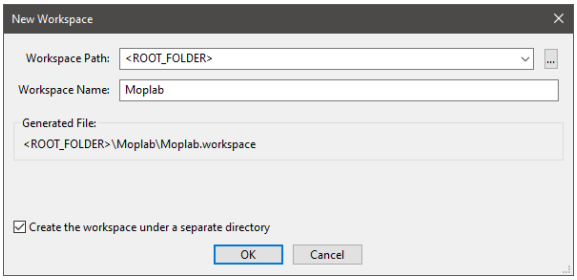
Skapa ett nytt ”Workspace”

Skapa ett nytt arbetsutrymme, *Workspace*, där du senare lägger till programmeringsprojekt av olika slag. Det är lämpligt att du förbereder ett nytt arbetsbibliotek för detta.



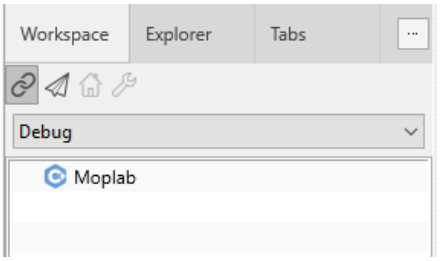
Välj Workspace | New Workspace från menyn.

Välj projekttyp C++ och klicka OK.



Tänk då också på att inte använda mellanslag eller specialtecken i sökvägen till arbetsbiblioteket för att undvika senare problem med verktygskedjan. Klicka OK då du valt lämpligt bibliotek.

I *CodeLites* Workspace View kommer nu ditt nya Workspace fram. Här kan du senare succesivt skapa och lägga till nya programprojekt för olika typer av uppgifter, *applikationer*.



Globala inställningar för arbetsutrymmet

Då du skapat ett nytt arbetsutrymme behöver du samtidigt komplettera inställningarna för kompilatorer du vill använda. De färdiga *projektmallarna* som kommer att användas förutsätter att miljövariabler definierats. Miljövariabeln används i färdiga mallar för projektinställningar och krävs för att rätt kompilator ska användas.

Välj

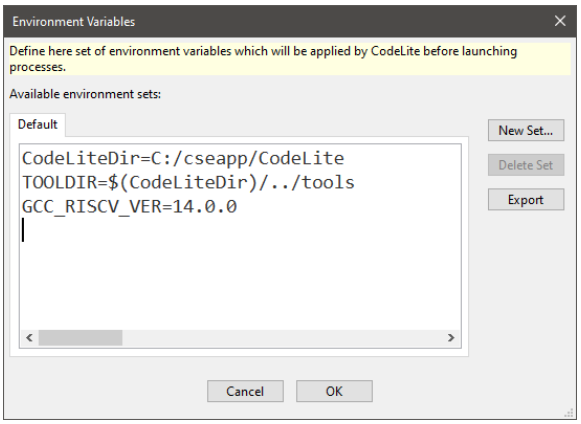
Settings | Environment Variables

från huvudmenyn.

Miljövariabeln CodeLiteDir har redan skapats av *CodeLites* installationsprogram.

Lägg till miljövariabeln TOOLDIR enligt figuren till höger. Denna används för att ange sökvägen för rotbiblioteket till alla installerade GCC-kompilatorer.

Lägg till miljövariabeln GCC_RISCV_VER som anger den version av GCC som används. Denna krävs för att rätt bibliotek ska länkas tillsammans med dina program.



Slutligen, för att färdigställa förberedelserna i arbetsutrymmet måste vi definiera ytterligare konfigurationer för de programprojekt vi senare ska skapa.

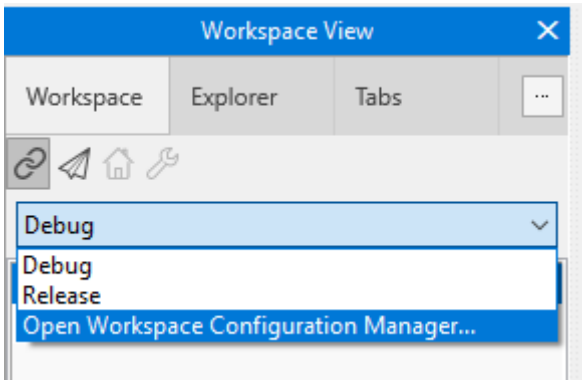
Från början finns två konfigurationer: `Debug` och `Release`. Projekt som kompileras i dessa konfigurationer kan laddas ned och exekveras i *MD307* med dess inbyggda debugger *dbg307*.

Vi vill lägga till två nya:

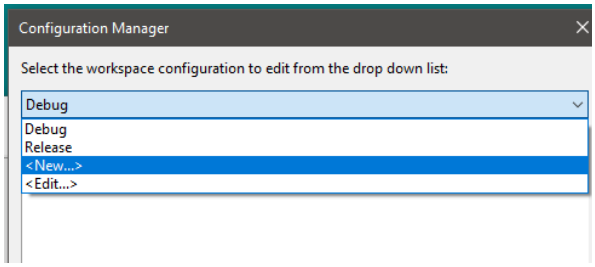
`Debug-sim` – avsett att utföras i *SimServer*

`Debug-ocd` – avsett att utföras i *MD307* med `openocd`.

Aktivera listvalet för konfigurationer och välj 'Open Workspace Configuration Manager...'



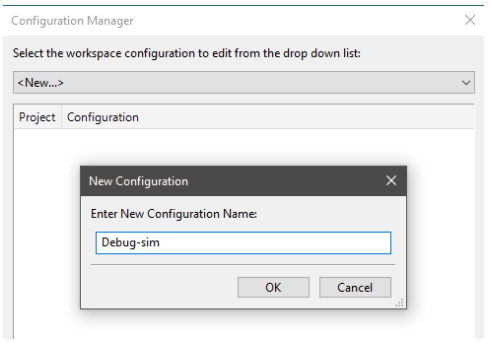
Välj <New> för att definiera en ny konfiguration.



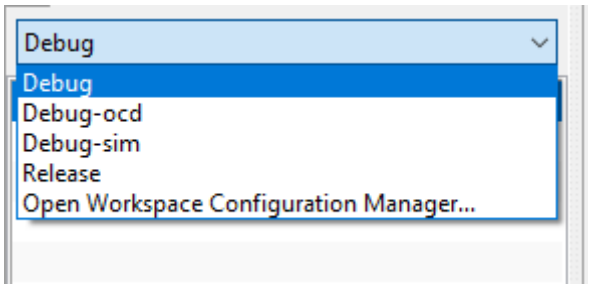
VIKTIGT:

Skapa behållare för egendefinierade konfigurationer:

- Ange namnet `Debug-sim`, klicka OK
- Gör på samma sätt för att definiera `Debug-ocd`



Kontrollera slutligen genom att på nytt aktivera listvalet för konfigurationer. Det ska nu se ut som i figuren till höger.



Den beskrivna processen behöver du göra för varje nytt arbetsutrymme (*Workspace*) du skapar.

Fortsättningsvis kommer du att skapa dina olika *programprojekt* inom detta arbetsutrymme, vilket nu blir betydligt enklare.

PROJEKTMALLAR, STARTFILER OCH KONFIGURATIONER

Vi använder olika startsekvenser beroende på applikationens komplexitet. Varje startsekvens finns klar i någon av de olika projektmallarna. Tre olika varianter finns förberedda:

- `md307-startup-simple`: innehåller den enklaste startup-sekvensen avsedd för de tidiga applikationerna vi arbetar med.
- `md307-startup`: kompletterats för applikationer som använder avbrottshantering.
- `md307-startup-crt`: den mest omfattande, kompletterad för applikationer som dessutom använder standard-C biblioteket.

Inledningsvis använder vi den enkla projektmallen, de övriga återkommer vi till efterhand som vi möter situationer som kräver dessa.

Genom att tillhandahålla olika inställningar kan nu samma programmeringsprojekt enkelt framställas i olika *konfigurationer*. Det finns fyra olika förberedda konfigurationer som skiljer sig åt genom att de anpassats för olika exekveringsmiljör:

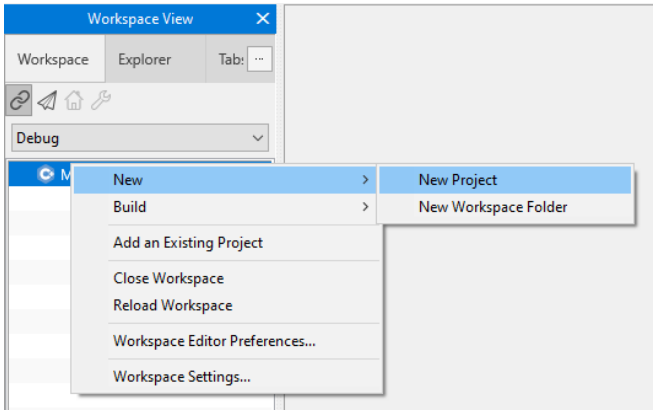
- Debug – kodgenerering utan kodförbättring - denna variant är avsedd att laddas till laborationsdatorn *MD307* och därefter testas/exekveras med hjälp av den inbyggda debuggern *dbg307*. Test sker, utan koppling till källtexter, på låg nivå (*machine level debugging*).
- Debug-sim – definierar macrot `SIMULATOR` - kodgenerering utan kodförbättring - denna variant är avsedd att laddas till simulatorn *SimServer* och därefter testas och exekveras med av *GDB*. Varianten tillåter därför debugging på källtextnivå.
- Debug-ocd – definierar macrot `COLDSTART` - kodgenerering utan kodförbättring - denna variant är avsedd laddas till laborationsdatorn *MD307* och därefter testas/exekveras via kortets debug-adapter (WCH-LINK) med hjälp av *GDB*. Även denna variant tillåter därför debugging på källtextnivå.
- Release – kodgenerering med kodförbättring - denna variant är avsedd laddas till laborationsdatorn *MD307* och därefter exekveras med hjälp av den inbyggda debuggern *dbg307*. Ytterligare användning är att studera listfilen med den genererade assemblerkoden som ofta skiljer sig markant från debug-versionerna.

Skillnader mellan debug-konfigurationerna är små: Debug-sim kräver en sekvens som förbereder och startar *GDB*. Detta gäller också för Debug-ocd, men här kan det också komma att krävas en annan startsekvens för applikationsprogrammet.

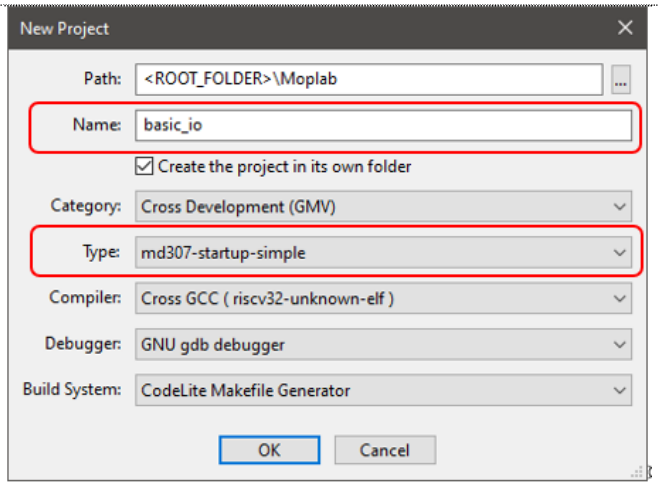
Skapa ett nytt projekt

Du kan skapa ett nytt projekt genom att välja
File | New | New Project
från huvudmenyn.

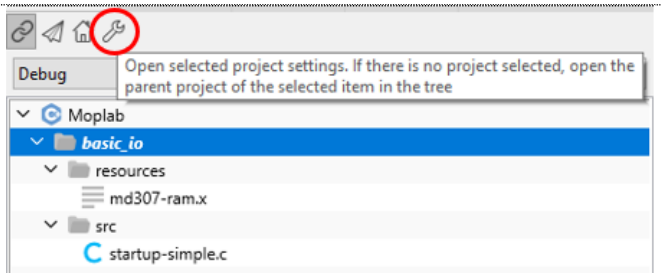
Alternativt, högerklicka på namnet för ditt *Workspace*, en popupmeny ger dig då ett antal alternativ,
Välj New | New Project



Nu kan du välja någon av de färdiga projektmallarna:
Välj Type: `md307-startup-simple`
Name: Skriv in projektets namn
(här `basic_io`).
Välj nu inställningar enligt figuren till höger.
Klicka därefter Ok.
Två filer kopieras nu till det nya projektbiblioteket: ett enkelt *länkarscript* (`md307-ram.x`) och en C-fil (`startup-simple.c`) där du kan redigera ditt nya program.

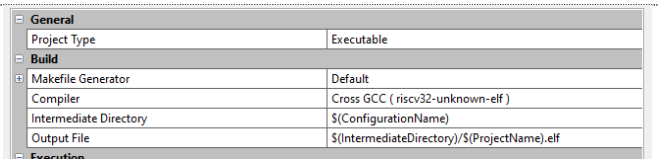


Nu är det också lämpligt att anpassa och kontrollera inställningarna för projektet.
Öppna dialogen för projektinställningar genom att klicka på verktyget i navigationsfönstret.

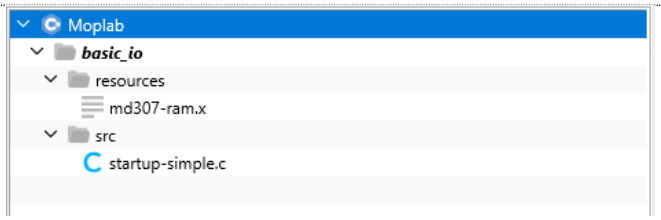


Ändra Makefile Generator till Default.

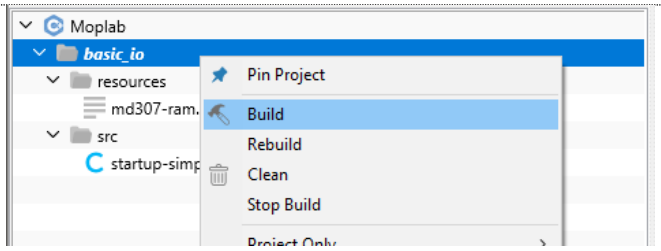
Ändra också ändelsen för Output File till `.elf`



Det kan vara lämpligt att döpa om filen `startup-simple.c` till något applikationspecifikt (högerklicka på filnamnet), vi gör dock inte det här. I stället beskriver vi den givna koden i `startup-simple.c`.

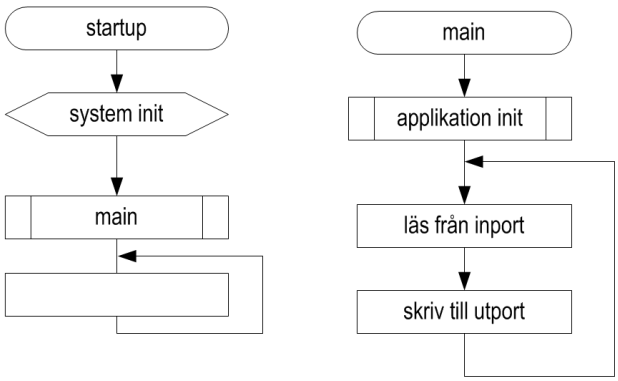


Kontrollera slutligen att projektes kan ”byggas” genom att högerklicka på projektets mapp och välja Build.



SKAPA EN APPLIKATION

Vi ska nu skapa en enkel applikation. Den är uppdelad i två delar: En `startup`-sekvens där alla initieringar som ska göras före anropet av applikationsprogrammet (`main`) görs, *system init*, och själva applikationsprogrammet.



FIGUR 2: ENKEL APPLIKATION MED IN- OCH UTMATNING

Då vi använder *GCC* får vi också tillgång till utvidgningar till programspråket, *språktillägg*, som är speciellt avsedda att underlätta maskinorienterad programmering.

Vi ska nu se två olika språktillägg, utöver C-språket, som möjliggör arbete direkt med den underliggande hårdvaran:

- *Inline* assemblerkod
- *Nakna* funktioner

Medan fördelarna med sådana här språktillägg är uppenbara ska man samtidigt komma ihåg att programmet kan komma att bli mindre portabelt, dvs. möjligheterna att kompilera källtexterna i andra miljöer än *GCC* kan påverkas.

INLINE ASSEMBLERKOD

Det finns speciella RISC-V-instruktioner som inte används direkt av C-kompilatorn men som ändå krävs vid maskinnära programmering.

Ett uppenbart sätt att hantera detta är att konstruera funktioner (subrutiner) som kräver användning av sådana instruktioner, direkt som assemblerkod, assemblera dessa och länka tillsammans med övrig kod som vi då skrivit i C. Vi måste då tänka på att vi följer de konventioner som gäller för kodgenereringen för att kunna skicka parametrar till funktioner och returvärden i från dessa.

Som ett alternativ till att skriva separata assemblerfiler kan vi använda *inline assemblerkod*. Detta kan dels spara moment i programutvecklingen men också ge en bättre överskådlighet av funktioner som annars tenderas att stuvas undan som "svårbegriplig" kod.

EXEMPEL 1 "IN LINE" ASSEMBLERKOD

Följande konstruktion:

```
__asm volatile(" nop ");
```

sätter in en `NOP`-instruktion (”No operation”) i programmet.

Ytterligare ett sätt att använda *inline*-konstruktioner visas i följande exempel på en minimal *startup*-funktion för ett C-program.

EXEMPEL 2 MINIMAL STARTUP FÖR MD307

Minimal startupfunktion, i funktionen ges först stackpekaren ett lämpligt värde, därefter anropas själva applikationsprogrammets funktion `main`. Avslutningsvis placeras en `jump`-instruktion som gör att programexekveringen fortsätter här i all oändlighet om `main`-funktionen skulle avslutas.

```
void startup ( void )
{
__asm__ volatile(" la sp,0x2001C000\n");
__asm__ volatile(" jal main\n");
__asm__ volatile(".lowexit: j .lowexit\n");
 }
```

Här har vi kompilerat exemplet ovan till assemblerkod och dessutom klippt bort åtskillig text för att se den genererade koden som då blir: Vi ser att kompilatorn lagt ut tre inledande instruktioner (prolog) innan vår instruktionssekvens, därefter tre avslutande instruktioner (epilog). De extra kompilatorgenererade instruktionerna återges här kursivt.	`startup:` *`addsp,sp,-16`* *`sw s0,12(sp)`* *`adds0,sp,16`* `la sp,0x2001C000` `jal main` `.lowexit: j .lowexit` *`lw s0,12(sp)`* *`addsp,sp,16`* *`ret`*

För en godtycklig funktion krävs normal användning av prolog/epilog enligt exempel 2 men i detta fall är det överflödigt. Vi vill ju att vår `startup`-funktion *bara* ska innehålla våra egna instruktioner. För sådana här fall använder man ett så kallat *attribut* (`"naked"`) som talar om för *GCC* att vi inte vill ha prolog/epilog i vår funktion:

```
void startup(void) __attribute__((naked));
```

Man kan också styra i vilken *sektion* man vill att funktionen ska placeras. I detta fall vill vi att funktionen `startup` placeras först, framför all annan kod i applikationen. Eftersom vi tidigare definierat en speciell sektion ("start_section") för just detta ändamål kan vi nu placera vår startup på rätt ställe med attributet *section*:

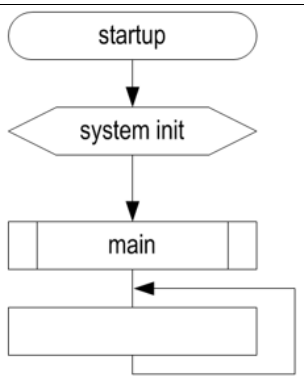
```
__attribute__((section (".start_section"))
```

Vår applikationsmall blir då:

```
__attribute__((naked)) __attribute__((section (".start_section")) )
void startup ( void )
{
__asm__ volatile(" la sp,0x2001C000\n");   /* set stack */
__asm__ volatile(" jal main\n");           /* call main */
__asm__ volatile(".L1: j .L1\n");          /* never return */
}
```

Denna startup fungerar fint med konfigurationerna `Debug`, `Debug-sim` och `Release`. Vill vi däremot använda `Debug-ocd` måste vi lägga till programkod som gör nödvändiga initieringar av hårdvaran i *MD307*. I det enklaste fallet innebär detta att aktivera de båda portarna D och E som vi kommer att använda i vår applikation.

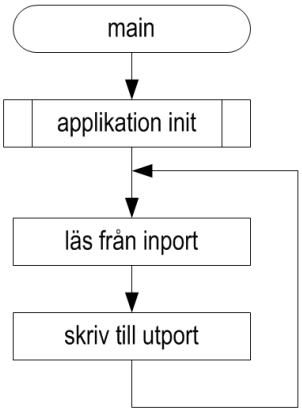
```
__attribute__((naked)) __attribute__((section (".start_section")) )
void startup ( void )
{
__asm__ volatile(" la sp,0x2001C000\n");   /* set stack */
#ifdef COLDSTART
   /* enable GPIO D and E clocks */
   * ((volatile unsigned int *) 0x40021018) |= (1<<6)|(1<<5);
   /* reset GPIO D and E */
   * ((volatile unsigned int *) 0x4002100C) |= (1<<6)|(1<<5);
   * ((volatile unsigned int *) 0x4002100C) &= ~((1<<6)|(1<<5));
#endif
__asm__ volatile(" jal main\n");           /* call main */
__asm__ volatile(".L1: j .L1\n");          /* never return */
}
void main(void)
{
}
```



Denna startup-sekvens används i den förberedda projektmallen md307-startup-simple.

Vi fortsätter nu med main-funktionen. Denna kan implementeras på följande sätt enligt flödesplanen (jämför programmet med `mom1.asm` i arbetsbokens första kapitel):

```
void app_init ( void )
{
   * ( (unsigned long *) 0x40011400) = 0x44444444;
   * ( (unsigned long *) 0x40011800) = 0x22222222;
}

void main(void)
{
   unsigned short s;
   app_init();
   while(1){
        s= *(( unsigned short *) 0x40011408);
        * ( (unsigned short *) 0x4001180C) = s;
   }
}
```



Test och felsökning med Codelite

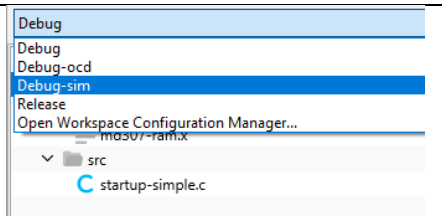
UPPGIFT 1

Redigera den enkla applikationen tillsammans med *startup*-proceduren i projektets källtextfil, `startup-simple.c`.

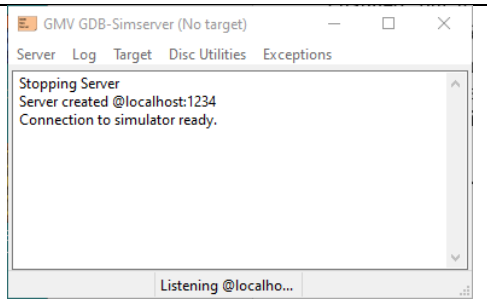
```
void app_init ( void )
{
    * ( (unsigned long *) 0x40011400) = 0x44444444;
    * ( (unsigned long *) 0x40011800) = 0x22222222;
}

void main(void)
{
    unsigned short s;
    app_init();
    while(1){
        s= *(( unsigned short *) 0x40011408);
        * ( (unsigned short *) 0x4001180C) = s;
    }
}
```

Välj konfiguationen `Debug-sim`.

Bygg sedan projektet (`basic_io`), rätta eventuella fel, om allt är rätt hittills ska den exekverbara filen `startup` skapas i projektets underbibliotek `Debug-sim`.



Nu ska det vara klart att börja testa programmet, starta *SimServer*.



Kontrollera att rätt måldator (*MD307*) är vald.

Server | Set target

I *CodeLite*, välj fliken med källtextfilen `startup.c`, placera markören på den första raden med exekverbar kod i *main*-funktionen, Tryck F9 (se även menyalternativ under Debugger) för att sätta ut en brytpunkt, (tryck F9 igen för att ta bort den).

```
void main(void)
{
    unsigned short s;
    app_init();
    while(1){
        s= *(( unsigned short *) 0x40011408);
        * ( (unsigned short *) 0x4001180C) = s;
    }
}
```

Starta nu programmet och exekvera det fram till brytpunkten med:

Debugger|Start/Continue Debugger F5.

Observera hur *CodeLite* märker ut nästa exekveringspunkt i programmet med en grön pil i marginalen.

```
void main(void)
{
    unsigned short s;
    app_init();
    while(1){
        s= *(( unsigned short *) 0x40011408);
        * ( (unsigned short *) 0x4001180C) = s;
    }
}
```

Då debuggern startats (F5) visas också en verktygslist (Windows och Linux) som ett komplement till att styra *GDB* med snabbvalstangenter.

Verktygslist för GDB i CodeLite:		Snabbval
	Starta programmet (fortsätt exekvera), motsvarar c-kommandot	F5
	Avsluta GDB och återgå till redigeringsläge, detta motsvarar quit-kommandot	Shift – F5
	Avbryt exekverande program, används då programmet startats med c-kommandot. Skickar <ctrl-c> till GDB för att stoppa det exekverade programmet	
	Visa aktuell rad, efter att ha avbrutit ett exekverande program kan det hända att markören för exekveringspunkten inte visas i textredigeringsfönstret. Använd denna funktion för att åtgärda detta	
	Återstarta program, observera att denna funktion inte används vid kors-utveckling eftersom CodeLite då försöker starta programmet direkt. Skulle du av misstag använda denna funktion med ARM-versionen måste du starta om GDB.	
	Utför nästa sats i programmet, om nästa sats är ett funktionsanrop kommer GDB att stega in i funktionen	F11
	Utför nästa *rad* i programmet, om raden exempelvis innehåller ett funktionsanrop kommer hela funktionen att utföras och nästa rad i källtexten blir den nya exekveringspunkten	F10
	Utför nästa assemblerintruktion i programmet	Ctrl – F10
	Exekvera färdigt aktuell funktion	Shift – F11

UPPGIFT 2

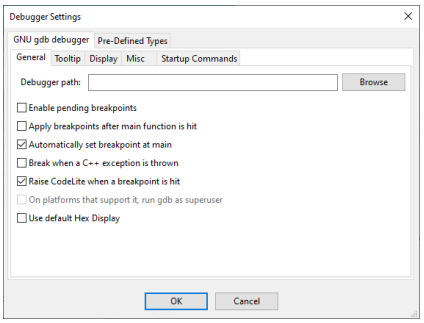
I *SimServer*, kopplar du nu upp en DIL-switch till port PD0-7 och en Bargraph till port PE0-7.

Använd menyn Debugger, eller verktygsfältets ikoner för att stega igenom programmet, kontrollera att det fungerar som det ska.

Du kan nu också passa på att göra några praktiska inställningar för *GDB* i *Codelite*, välj:

Settings | GDB Settings...

Under fliken General: Här kan det vara praktiskt att välja Automatically set breakpoint at main, så slipper du själv sätta ut den första brytpunkten i programmet:

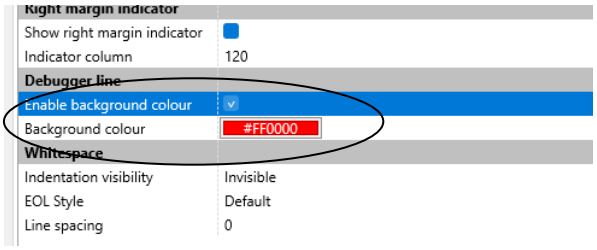


En annan praktisk inställning, denna gång under:

Settings | Preferences

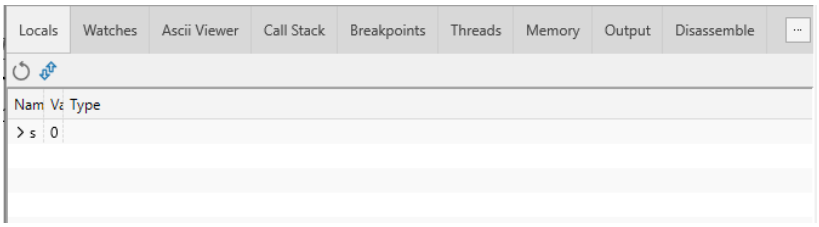
välj fliken Guides:

Du kan låta *Codelite* märka upp exekveringspunkten i programmet på ett tydligare sätt, här har vi till exempel angett att *Codelite* ska visa aktuell rad mot en röd bakgrund för att vi lättare ska kunna identifiera den.

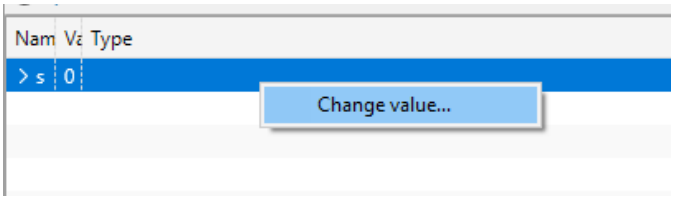


Då du startat *GDB* skapar *CodeLite* ett separat fönster 'Debugger' med en rad olika flikar.

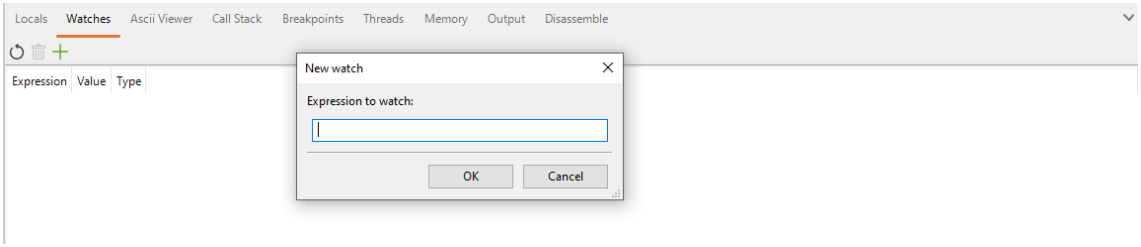
Locals: här får du en översikt av de lokala variablerna och deras innehåll.



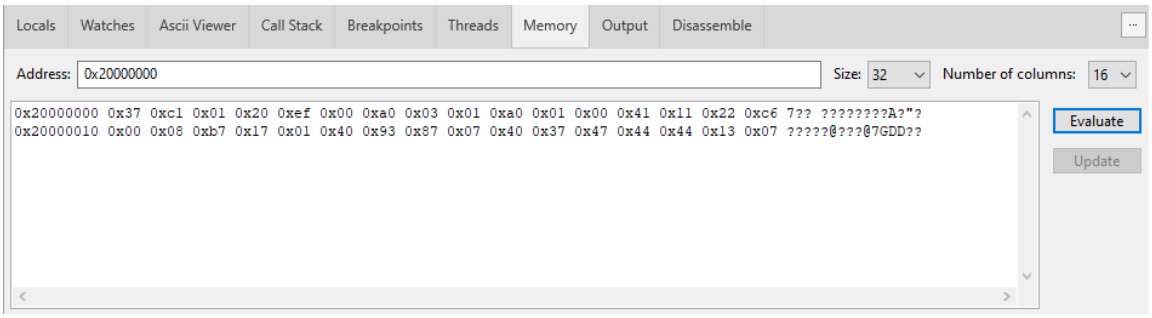
Du kan också ändra ett värde genom att högerklicka på raden med variabeln.



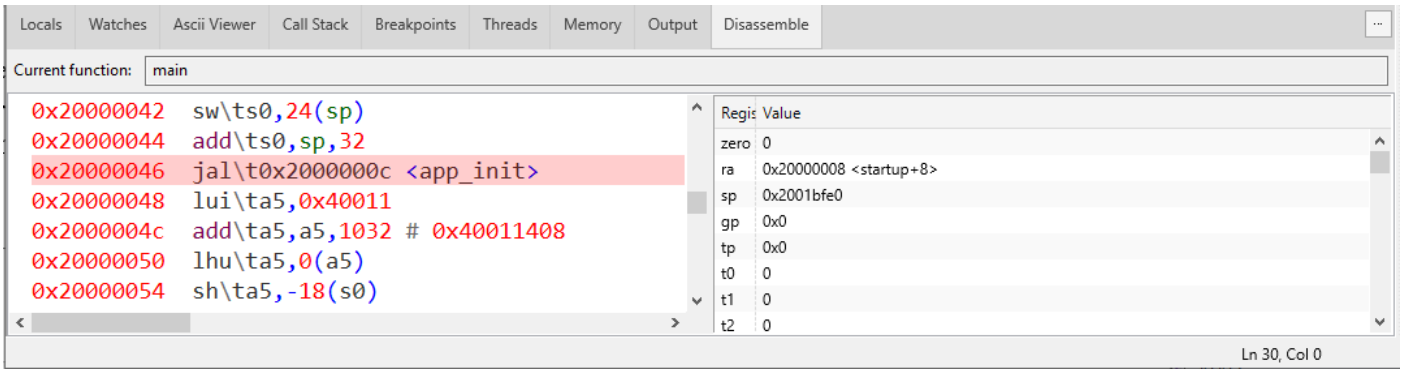
Watches: Används för att övervaka variabler med permanent lagringsplats i minnet. Lägg till den variabel du vill övervaka genom att klicka på det gröna '+'-tecknet i övre vänstra hörnet och därefter skriva in variabelns namn.



Memory: Används för att övervaka sammanhängande minnes innehåll. Tänk på att adressen ska skrivas med prefix '0x' om den anges på hexadecimal form.



Dissassemble: Då programmet stannat vid en brytpunkten kan du också växla till dissassembleringsfliken:



I fönstret till vänster ser du en dissassemblering av programmet där exekveringspunkten är uppmärkt. Du kan låta *GDB* utföra en assemblerinstruktion i taget (motsvarar stepi) med kommandot Next Instruction (Ctrl-F10).

Till höger visas processorns register och registerinnehållen. Du kan inte ändra registerinnehåll i *Codelite*.

Anm:

I Dissassemble-fönstret översätter *CodeLite* tecknet <TAB> med `\t` i stället för tabulaturen. Detta kan vara förvirrande i början.

Nu ska nu kunna redigera, kompilera en källext och därefter starta *GDB* från *Codelite* för att testa ditt program. Momenten är grundläggande för fortsättningen.

Studiematerial, Maskinorienterad programmering

Studiematerialet kring *MD307* består av:

- Laborationsdator *MD307* (*PTB-1000*)
- IO-kort -*PTB-110*
- Display-kort *PTB-111*
- Arbetsbok *Maskinorienterad programmering med MD307*.

Programvaror (Windows, Linux och MacOS)

- *ETERM8*
- *SimServer*
- *CodeLite*, lämplig distribution till detta material.
- *GCC för Windows (Mingw64)*.
- *GCC korskompilator för RISC-V*, distribution som kompletterats för användning tillssammans med *MD307*.

Dokumentation (PDF):

- *Quick Guide – MD307*
- *SimServer/IO-simulator*, användarhandbok.
- *PTB-1000* användarhandbok
- *PTB-100* användarhandbok
- *PTB-111* användarhandbok

Handledningar (PDF)

- *ETERM8 och MD307, inledande övning 1* – introducerande övning inför laborationer.
- *GDB och SimServer med MD307, inledande övning 2* – introducerande övning inför laborationer.
- *GCC och SimServer med MD307, inledande övning 3* – introducerande övning inför laborationer.
- *Laborationsuppgifter med simulator, MD307* – anvisningar för att genomföra en hel laborationsserie. Samtliga uppgifter har här utformats för att kunna genomföras enbart med hjälp av de programvaror som omfattas i kursen. Det behövs alltså inte tillgång till laborationsutrustningen. För flertalet uppgifter hänvisas direkt till arbetsboken, du behöver därför även denna för att arbeta med detta häfte.