GCC och *Simserver* för *MD407*

inledande övning 3

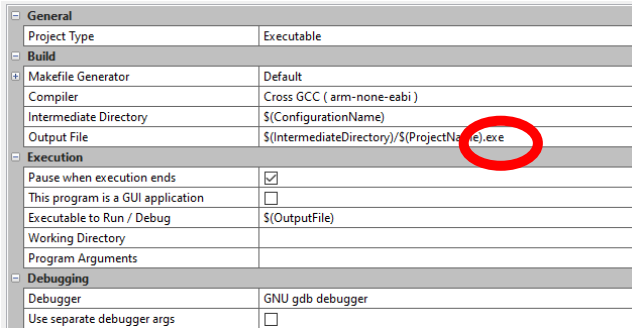
Under tidigare övningar har vi använt *ETERM8* och *GDB* tillsammans med *SimServer* för att studera test och felavhjälning för assemblerprogram på såväl maskinnivå (*machine level debugging*), som källtextnivå (*source level debugging*).

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.

OBS: CodeLite, version 16 för Windows.

Då project för GCC-ARM skapas, lägger Codelite (felaktigt) till ändelsen ”.exe” till namnet på den färdiga exekverbara filen. Då du skapat ett nytt sådant projekt ska du därför direkt ta bort denna ändelse.

Högerklicka på projektets namn och välj ’Settings’.



INNEHÅLL

Programutveckling med Codelite 2
Skapa ett nytt ”Workspace” 3
Skapa ett nytt projekt 5
Test och felsökning med Codelite 8

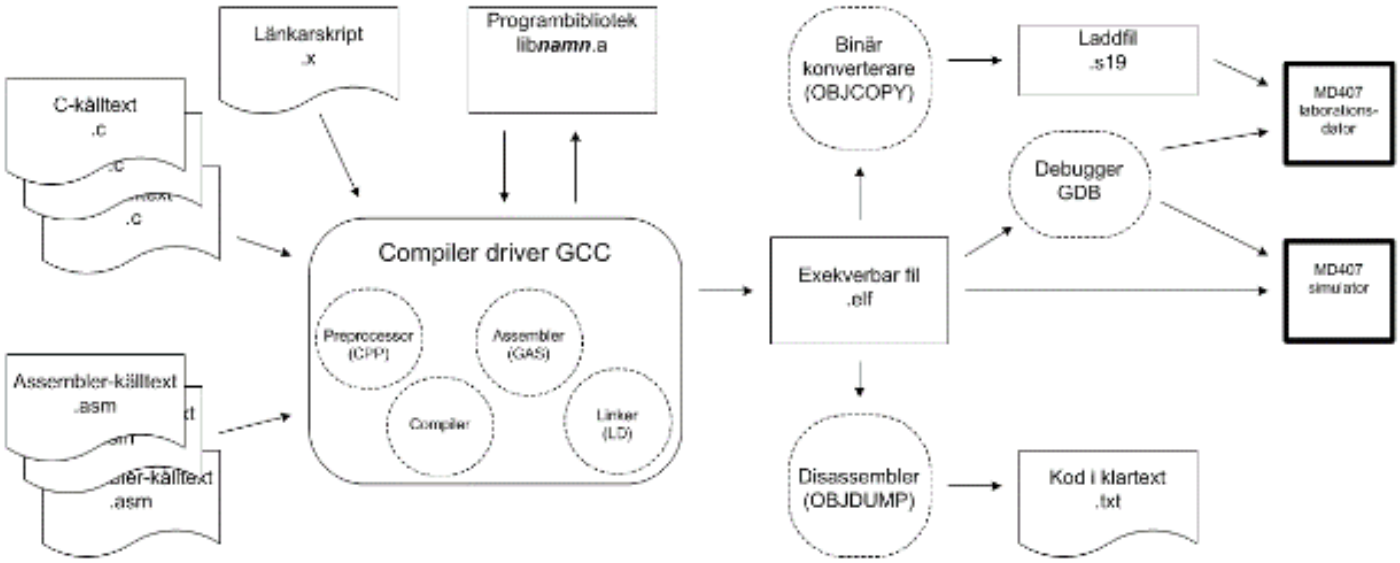
Programutveckling med Codelite

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 en applikation generellt byggs upp och hur den så småningom hamnar i datorns minne för att exekveras.
- Därefter ger vi ett konkret exempel på hur du skapar, kompilerar-länkar (kallas också "bygger") din applikation. I samband med detta visas hur utgår från en färdig mall som är speciellt anpassad för laborationsdatorn *MD407*.
- 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

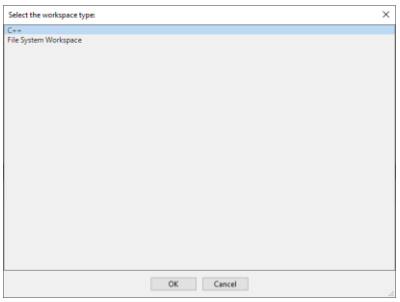
De olika verktygen finns för olika typer av processorer, följande tabell ger namnen på verktygskedjan för ARM.

Verktyg	Generiskt namn	Namn i verktygskedjan för ARM
C-kompilator	gcc	arm-none-eabi-gcc
Assemblator	as	arm-none-eabi- as
Länkare	ld	arm-none-eabi- ld
Binärfil konverterare	opjcopy	arm-none-eabi-opjcopy
Dissassembler	objdump	arm-none-eabi-objdump

TABELL 1: GCC VERKTYGSKEDJA FÖR ARM

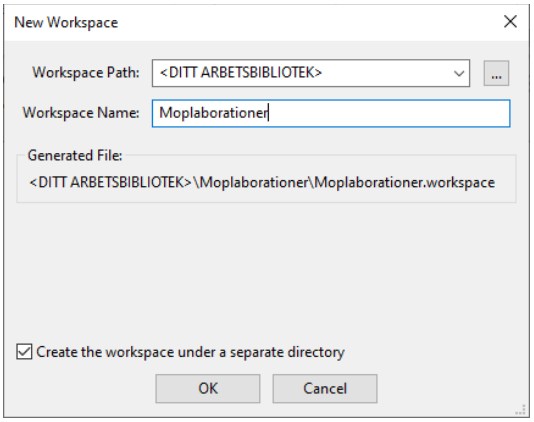
Skapa ett nytt ”Workspace”

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

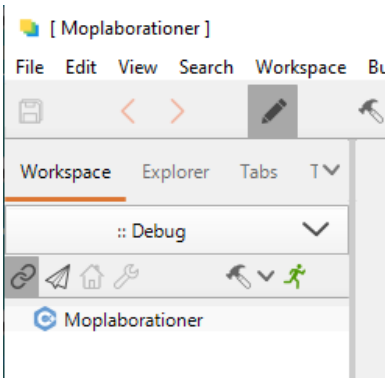


Välj Workspace | New Workspace från menyn.
Välj projekttyp C++ och klicka OK.

Tänk då också på att även här undvika mellanslag och 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*.



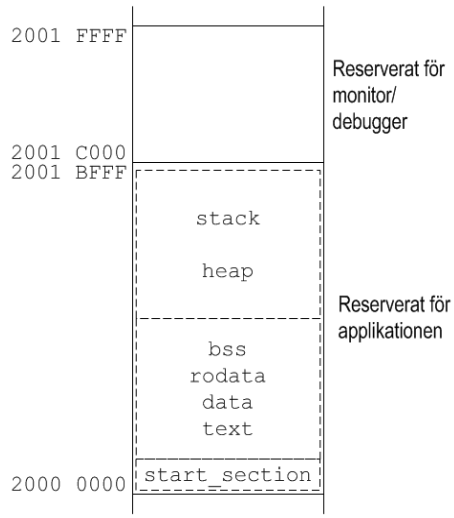
Innan vi går vidare med att skapa projekt ska vi titta lite grand på den omgivning våra program ska exekveras. Vi behöver reda ut hur applikationen byggs upp.

MINNESANVÄNDNING MD407, APPLIKATION I RWM

I figur 2 visas hur *RWM* disponeras i *MD407*. Vi har valt att placera applikationsprogrammet med start på minnets första adress, 20000000, och uppåt. Applikationens olika delar, *segment*, placeras här i den ordning figuren illustrerar. Detta styrs med *länkarskriptet* (`md407-ram.x`. Övre delen av minnet, är reserverat för *MD407*:s inbyggda monitor/debugger. Hos *MD407* används alltså maximalt 112kB av *RWM* för applikationsprogrammet medan resterande 16 kB har reserverats för det residenta programmet.

Programkod och data indelas i olika sektioner, figur 3 beskriver hur minnesdispositionen för ett komplett program, under exekvering, kan se ut. Figuren förstås bäst mot bakgrund av hur ett program översätts, sparas, laddas till primärminnet för att därefter utföras.

Applikationen består av en initierad del, *Image*, bestående av programkod och data. Den andra delen kallar vi *Run-Time*, Såväl programkod som data placeras i namngivna *sektioner* som kan ha godtyckliga namn. I figuren används exempelvis sektionerna `bss`, `data`, `rodata` och `text`. Dessa namn används också av *GCC* vid översättningen.



FIGUR 2: ANVÄNDNING AV RWM I MD407

prefix

Prefixet, det som också kallas *startupkod*, placeras först. Prefixet ingår ofta i den så kallade *runtime-miljön* som installeras tillsammans med kompilatorn. Ofta kan det finnas olika sådana miljöer och de kallas typiskt `crt`, (*c-run-time*).

applikationskod

Här placeras all övrig programkod. Den får inte vara självmodifierande, dvs sektionen förutsätts vara *read-only*. Kompilatorn gör en "bild" av maskinkod som laddas i minnet. Av tradition namnges en sektion med programkod `text`.

initierade variabler

Deklarationer som `int a = 2; char array[]={”String”};` på toppnivå kan användas för att deklarera globala variabler. Innehållet är definierat från start, men kan komma att ändras under exekvering. Kompilatorn måste göra en "bild" även av detta segment för att dessa initialvärden ska kunna laddas till minnet före exekvering. *GCC* använder regelmässigt två olika segment `data` och `rodata` (*read-only data*) för initierade globala variabler.

icke initierade variabler

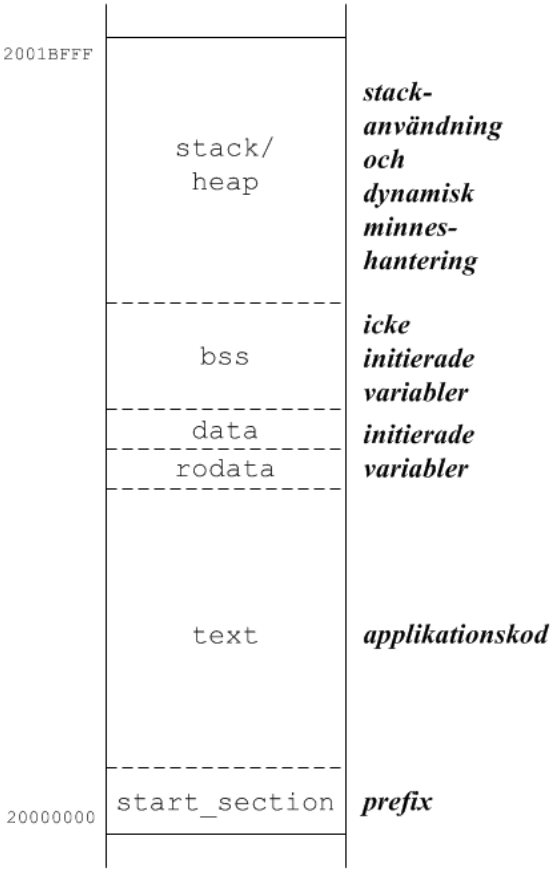
Deklarationer som: `int a; char array[34];` osv, används för att deklarera variabler utan att samtidigt ge dessa initiala värden. Eftersom variablerna inte har något definierat innehåll från start behöver kompilatorn bara hålla reda på var, i minnet dessa hamnar. Det behövs alltså ingen "bild". Oinitierade variabler placeras därför i ett särskilt segment. Av tradition kallas detta `bss`, *block started by symbol*, även namnet `COMMON` förekommer.

stack

Stacken används av program under exekvering. Minne måste reserveras för att rymma den maximala stackanvändningen.

heap

Heapen benämns ofta det minnesutrymme som reserverats för dynamisk minneshantering. Exempelvis `malloc(),free()` i standard-C biblioteket kräver ett sådant minnesutrymme.



FIGUR 3: MINNESANVÄNDNING FÖR PROGRAM UNDER EXEKVERING

Låt oss sammanfatta detta. Vid kompilering av en källtextfil skapas en objektmodul med följande information/innehåll:

- `text`-sektion innehållande en "bild" av programkoden.
- `data`-sektion innehållande en "bild" av initierade data som kan komma att ändras under programexekvering.
- `rodata`-sektion innehållande en ”bild” av initierade data som *inte* kan ändras under programexekvering.
- Storleken av `bss`-sektionen.
- Symboltabell innehållande alla globala symbolers relativa adresser (offset till sektionens början) i respektive sektion. Observera att alla symboler är relokerbara, dvs absoluta adresser i minnet har ännu ej bestämts.

Då programmet ska exekveras utförs följande:

1. Prefix adderas till text-sektionen.
2. Minnesbehov för sektionerna text, data, rodata och bss bestäms.
3. Segmenten relokeras med hjälp av symboltabellen.
4. Minnesbehov för stack och heap bestäms (av operativsystemet).
5. Totala minnesbehovet är nu känt och tillräckligt primärminne kan reserveras för programmet.
6. Programmets initierade segment ("bilder") kopieras till sin respektive plats i primärminnet.
7. Stackpekare initieras och programmet startas (i prefix).

Observera speciellt hur förfarandet förutsätter att denna procedur upprepas inför varje exekvering av programmet.

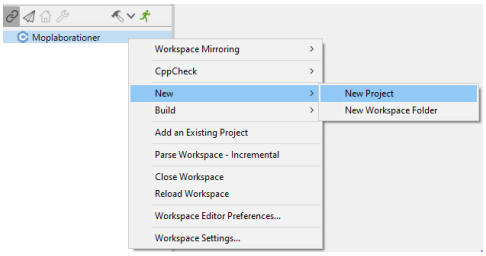
Då man arbetar i en kors-utvecklingsmiljö, som med GCC för ARM, har man som regel inget operativsystem som utför detta utan bara en enkel debugger i målsystemet. Detta innebär att moment som normalt utförs enligt någon strategi bestämd av operativsystemet, nu måste utföras manuellt. Följande punkter är speciellt viktigt att iaktta:

- Stackpekare måste initieras
- Det finns ingen verklig dynamisk minneshantering tillgänglig
- Programmet måste laddas, från utvecklingssystem till måldatorsystem mellan varje exekvering, oavsett om det har ändrats eller ej. Detta gäller dock bara om programmet har ett `data`-segment, eftersom den ursprungliga initieringen *kan* ha ändrats under en tidigare exekvering av programmet.

Skapa ett nytt projekt

Du kan skapa ett nytt projekt genom att högerklicka på namnet för ditt Workspace, en popupmeny ger dig då ett antal alternativ,

Välj New | Create New Project



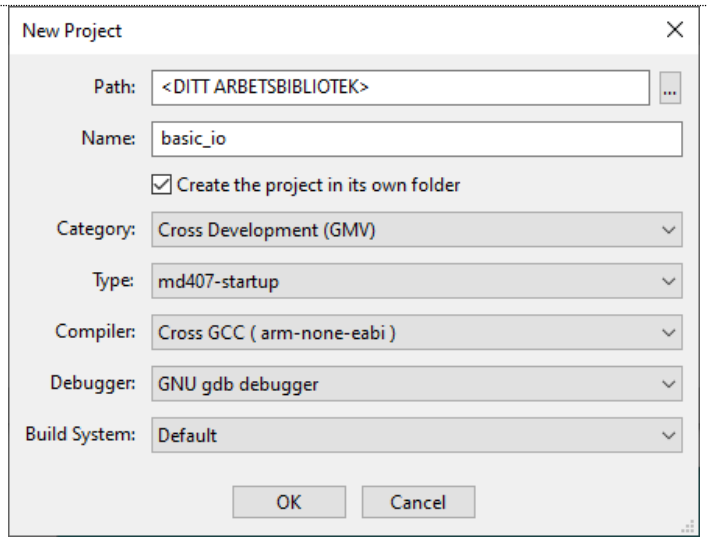
Nu kan du välja någon av de färdiga projektmallarna:

Name: Skriv in projektets namn (här `basic_io`).

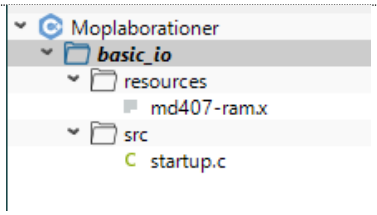
Välj nu inställningar enligt figuren till höger.

Klicka därefter Ok.

Två filer kopieras nu till det nya projektbiblioteket: ett enkelt *länkarscript* (`md407-ram.x`) och en C-fil (`startup.c`) där du kan redigera ditt nya program.

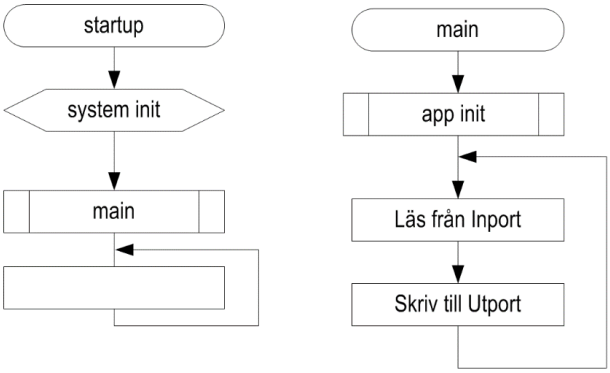


Det kan vara lämpligt att döpa om filen `startup.c` till något applikationspecifikt (högerklicka på filnamnet), vi gör dock inte det här. I stället beskriver vi den givna koden i `startup.c`.



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 4: 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 tre olika språktillägg, utöver C-språket, som möjliggör arbete direkt med den underliggande hårdvaran:

- *Inline* assemblerkod
- *Nakna* funktioner
- Fasta (reserverade) register

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. I detta avsnitt behandlar vi *inline* assemblerkod och *nakna* funktioner, vi återkommer till reserverade register i ett kommande avsnitt.

INLINE ASSEMBLERKOD

Det finns ett antal olika ARM-instruktioner (exempelvis. `MRS`, `MSR`, `SEV`, `SVC`) som helt enkelt inte kan uttryckas i C. Samtidigt är dessa instruktioner nödvändiga om vi exempelvis vill programmera *undantagshantering* (beskrivs i kapitel 6) för att kunna använda processorns avbrottsmekanismer.

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 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 MD407

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

```
void startup ( void )
{
__asm volatile(" LDR R0,=0x2001C000\n");
__asm volatile(" MOV SP,R0\n");
__asm volatile(" BL main \n");
__asm volatile(" B .\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 två inledande instruktioner (prolog) innan vår instruktionssekvens, därefter två avslutande instruktioner (epilog).
	`startup:`
	`push {r7,LR}`
	`add r7,sp,#0`
	`LDR R0,=0x2001C000`
	`MOV SP,R0`
	`BL main`
	`B .`
	`mov sp,r7`
	`pop {r7,PC}`

För en godtycklig funktion krävs normal användning av prolog/epilog 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 kan vi nu placera vår startup på rätt ställe med attributet *section*:

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

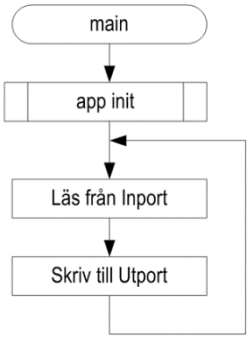
Vår färdiga applikationsmall blir därför:

```
__attribute__((naked))
__attribute__((section(".start_section")) )
void startup ( void )
{
__asm volatile(" LDR R0,=0x2001C000\n");
__asm volatile(" MOV SP,R0\n");
__asm volatile(" BL main \n");
__asm volatile(" B .\n");
}
int main(int argc, char **argv)
{
}
```

Vi fortsätter nu med main-funktionen. Denna kan implementeras på följande sätt enligt flödesplanen (jämför programmet med `mom1.asm` i kapitel 1):

```
void app_init ( void )
{
   * ( (unsigned long *) 0x40020C00) = 0x00005555;
}
void main(void)
{
   unsigned char c;
   app_init();
   while(1){
        c = (unsigned char) *(( unsigned short *) 0x40021010);
        * ( (unsigned char *) 0x40020C14) = c;

   }
 }
```

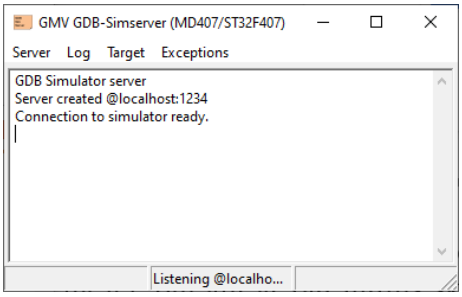


Test och felsökning med Codelite

UPPGIFT 1

Redigera den enkla applikationen tillsammans med *startup*-proceduren i projektets källtextfil, `startup.c`. Bygg projektet (`basic_io`), rätta eventuella fel, om allt är rätt hittills ska den exekverbara filen `startup` skapas i projektets underbibliotek `Debug`.

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



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).

```
void main(void)
{
    char c;
    app_init();
    while(1){
        c = (unsigned char) *(( unsigned short *) 0x40021010);
        * ( (unsigned char *) 0x40020C14) = c;
    }
}
```

Starta nu programmet och exekvera det fram till brytpunkten, Debugger|Start/Continue Debugger F5. Observera hur *CodeLite* märker ut nästa exekveringspunkt i programmet med en grön pil i marginalen.

```
void main(void)
{
    char c;
    app_init();
    while(1){
        c = (unsigned char) *(( unsigned short *) 0x40021010);
        * ( (unsigned char *) 0x40020C14) = c;
    }
}
```

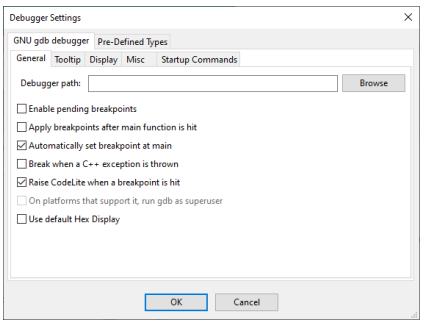
I *SimServer*, kopplar du nu upp en DIL-switch till port PE0-7 och en Bargraph till port PD0-7.

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

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

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

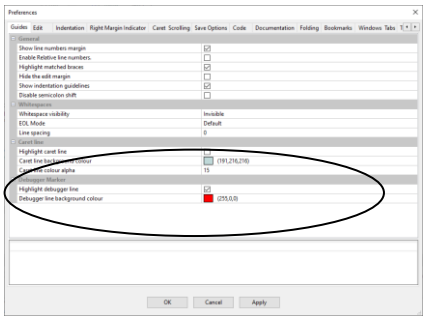
Settings | GDB Settings...



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:

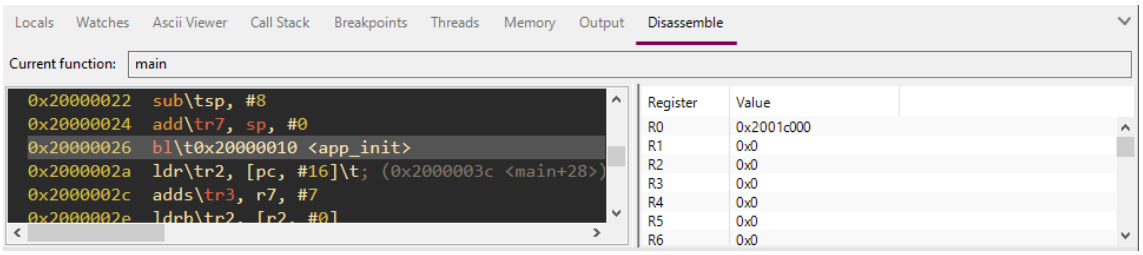
Settings | Preferences



är att 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.

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

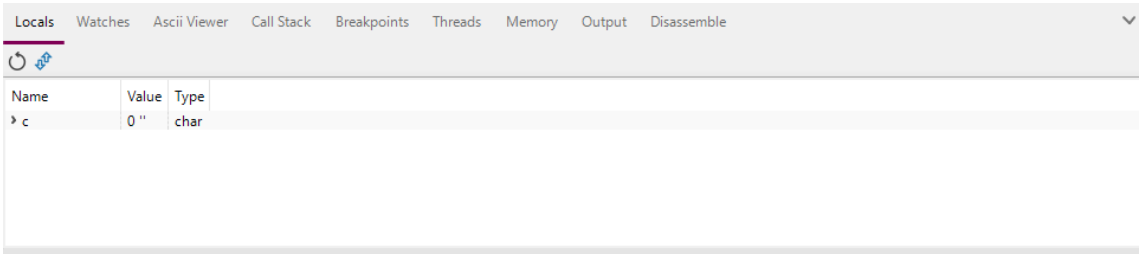


Anm:

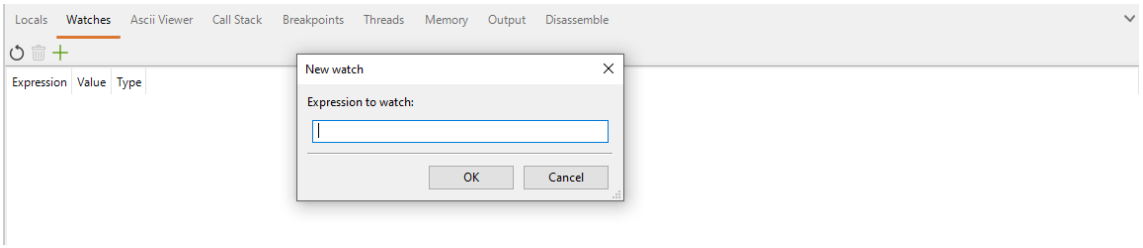
I Dissassemble-fönstret översätter *CodeLite* tecknet <TAB> med `\t` i stället för tabulaturen. Detta kan vara förvirrande i början.

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).

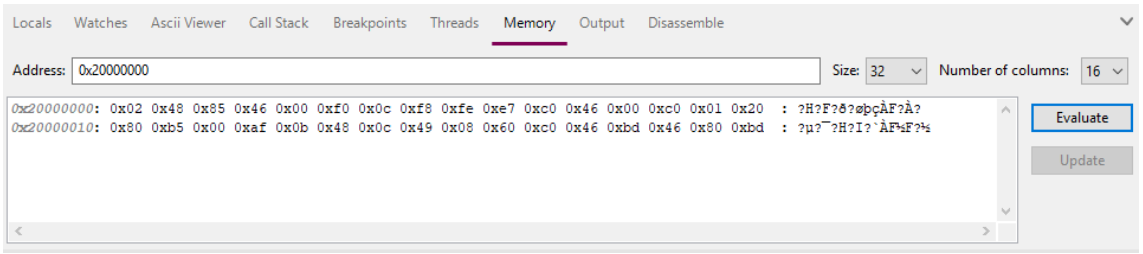
Locals: här får du en översikt av de lokala variablerna och deras innehåll. Du kan också ändra ett värde genom att högerklicka på raden med variabeln.



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.



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

Studiematerial, Maskinorienterad programmering

Utöver arbetsboken *Maskinorienterad programmering med MD407* ingår även följande material utformat så att du samtidigt kan arbeta självständigt med dessa:

Tryckt häfte

- *Quick Guide* – avsedd att anändas under tentamen eftersom det normal inte är tillåtet att använda utskrifter av PDF-version.

Programvaror: (Windows, Linux och MacOS)

- *ETERM8/SimServer*
- *CodeLite med GCC och GCC för ARM*, distribution som anpassats speciellt för detta material.

Dokument (PDF):

- *Quick Guide* - elektronisk form av det tryckta häftet
- *En snabb introduktion till maskinorienterad C* – kompendium som behandlar C speciellt som ett maskinnära programspråk.
- *ETERM8 och MD407, inledande övning 1* – introducerande övning inför laborationer.
- *GDB och SimServer med MD407, inledande övning 2* – introducerande övning inför laborationer.
- *GCC och SimServer med MD407, inledande övning 3* – introducerande övning inför laborationer.
- *Laborationsuppgifter med simulator, MD407* – 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.