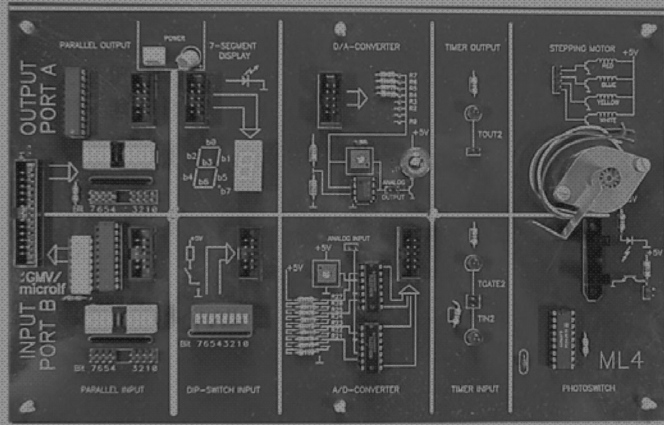
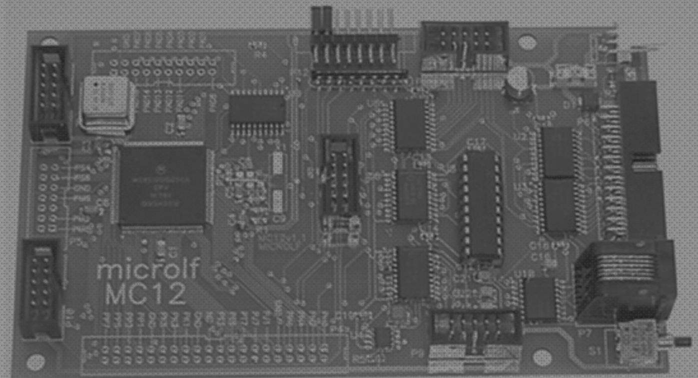
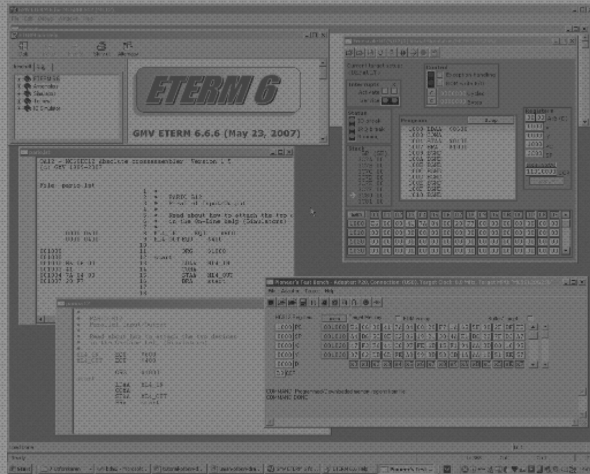
Utgåva 5 (2013)
Roger Johansson
och Rolf Snedsböl







Arbetsbok för MC12

Övningar med ETERM6 och XCC12

Art Nr: 120-19, ISBN 978-91-89280-24-3

Femte utgåvan, andra tryckningen

© 2005-2013 Roger Johansson, Rolf Snedsbøl och Göteborgs Mikrovaror (GMV)

e-post: info@gbgmv.se

Internet: http://www.gbgmv.se

Programvaror för användning tillsammans med denna arbetsbok finns på GMV's hemsida.

Kopieringsförbud

Detta verk är skyddat av upphovsrättslagen. Kopiering är förbjuden utöver lärares rätt att kopiera för undervisningsbruk enligt BONUS-Presskopias avtal. BONUS-avtal tecknas mellan upphovsrättsorganisationer och huvudman för utbildningssamordnare, exempelvis kommuner och högskolor/universitet.

Förord

Denna arbetsbok är avsedd i första hand som självstudiematerial. Arbetsboken ger en introduktion till enkortsdatorn ***microlf MC12*** uppbyggd kring mikrocontrollern ***Freescale MC68HCS12*** och laborationskort avsedda för MC12. Utöver denna arbetsbok förutsätts eleven ha tillgång till programvaran *ETERM* (för avsnitt 5 även *XCC*) med tillhörande simulatordelar. Arbetsboken omfattar övningsuppgifter fördelade på fem avsnitt:

I) Det inledande avsnittet behandlar *programutveckling i assemblerspråk*. Syftet med avsnittet är att introducera viktiga, grundläggande begrepp. Detta innebär såväl assemblerprogrammering av en mikroprocessor (speciellt MC68HCS12) men framför allt den använda programutvecklingsmiljön.

II) I detta avsnitt ska eleven självständigt bygga upp en mindre applikation, denna gång kring ett tangentbord och en display-modul. I avsnittet läggs stor vikt på ett *strukturerat arbetssätt.* Lämpliga metoder för programutveckling belyses.

III) Under hela detta avsnitt arbetar eleven självständigt med att bygga upp ett enkelt skrivargränssnitt. I avsnittet introduceras *avbrott* introduceras och speciell vikt läggs också vid *synkroniseringsproblematik*.

IV) Här arbetar eleven med att bygga upp ett komplett applikationsprogram för att styra en liten borrmaskin. I avsnittet läggs speciell vikt på *specifikation* och *dokumentation*.

V) I detta avslutande avsnitt behandlas maskinnära programmering i programspråket 'C'. Syftet är att ge en introduktion till programutveckling i 'C' och assembler. Speciellt belyses kodgenerering och programmering som kombinerar användning av högnivåspråket 'C' med assemblerspråket.

Göteborg i oktober 2005, Roger Johansson och Rolf Snedsbøl

I denna upplaga 3 har flera rättelser gjorts, appendix F har tillkommit. Text och figurer har uppdaterats för anpassning till de senaste versionerna av *ETERM 6* och *XCC12*.

Göteborg i augusti 2008, Roger Johansson och Rolf Snedsbøl

I den fjärde utgåvan har omfattande ändringar gjorts. Avsnitten 2 och 3 har arrangerats om och flertalet uppgifter har reviderats. En rad nya uppgifter har tillkommit.

Göteborg i januari 2012, Roger Johansson och Rolf Snedsbøl

Den femte utgåvan skiljer sig från den förra genom att tryckfel har rättats.

Göteborg i januari 2013, Roger Johansson och Rolf Snedsbøl

Innehåll:

Avsnitt 1
Programutveckling i assemblerspråk

Avsnitt 2
Tangentbord och Display

Avsnitt 3
Synkronisering och avbrott

Avsnitt 4
Programmeringsprojekt: Borrmaskinen

Avsnitt 5
Maskinnära programmering i C och assemblerspråk

Appendix
A MC12 IO-adresser för laborationskort
B Symboler i flödesdiagram
C MC12/dbg12 minnesdisposition
D Motorola S-format
E ASCII representationen
F Exceptionvektorer
G XCC objektfilsformat
H XCC script filer

Avsnitt 1

Programutveckling i assemblerspråk

Syften:

Det viktiga syftet med detta första avsnitt är att ge dig en introduktion till hur man programmerar i assemblerspråk. Du får därför lära dig att hantera verktyg för utveckling och test av datorprogram där programmets instruktioner, så långt det är möjligt, motsvaras av så kallade maskininstruktioner.

Du kommer att introduceras till en mikroprocessors instruktionsrepertoir och avsnittet är grundläggande för förståelsen av programmerbara digitala system

Inledning

Denna arbetsbok innehåller en lång rad övningar som bland annat syftar till att du ska lära dig behärska enkel programutveckling i assemblerspråk (maskinnära programmering).

Arbetsboken har organiserats i fem avsnitt, indelade i moment som du arbetar dig igenom i tur och ordning.

Detta första moment har ägnats helt åt beskrivningar av grundläggande begrepp. Vi börjar med en övergripande beskrivning av det använda utvecklingssystemet ETERM. Vi behandlar också programutvecklingsprocessen, dvs implementering och test av programvara.

Om du inte tidigare gjort det, börja nu med att installera ETERM på din persondator.

Arbetsboken behandlar laborationsdatorn MC12 som är uppbyggd kring microcontrollern Freescale MC68HCS12, eller kortare HCS12.

ETERM och programutvecklingen

ETERM är ett utvecklingssystem för programutveckling i assemblerspråk. ETERM har anpassats för undervisningsändamål. Det finns flera varianter av ETERM beroende på vilken måldator (laborationsdator) som används. I denna arbetsbok behandlas ETERM FÖR MC12.

ETERM omfattar funktioner för:

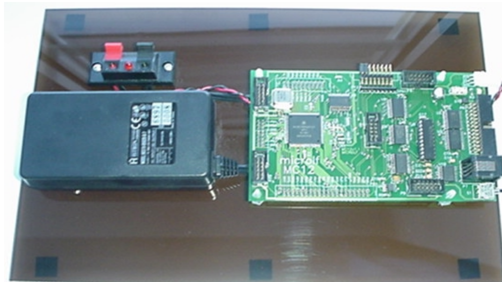
- **Textredigering**, källtexten skrivs/redigeras med hjälp av en *Editor*, filnamnet ska sluta med .s12 (source HCS12) eller .asm. Färgad syntax används för att hjälpa dig upptäcka enklare stavningsfel.
- **Assemblering**, källtexten översätts till en laddfil som innehåller maskinkoden, med tillägget .s19 av den inbyggda assemblatorn. Vid assembleringen skapas också en listfil (med tillägget .lst) som kortfattat kan sägas innehålla information från såväl källtexten som laddfilen.
- **Test**, laddfilen överförs till den inbyggda simulatorn eller till MC12 via ETERM's terminalfunktion med hjälp av respektive programdels laddningsprogram.

Under detta första moment kommer vi att behandla de två första funktionerna textredigering och assemblering. Innan vi beskriver funktionerna ska vi dock titta lite grand på vad assemblerprogrammering egentligen är för något.

Programutveckling i assembler skiljer sig inte särskilt mycket från programutveckling i något högnivåspråk. Programmet skrivs i form av källtext, dvs. en textfil som innehåller instruktioner och direktiv till assemblatorn. Då programmet, eller en lämplig del av det, är färdigt måste det översättas till maskinkod innan programmet kan testas i en måldator eller simulator. Översättningen av programmet kallas assemblering (ung. ”sätta samman”) och utförs av assemblatorn. Vid assembleringen skapas en laddfil och en listfil.

I laddfilen finns programmet representerat på en form som kan överföras till laborationsdatorn där den tolkas som instruktioner och data.

Då programmet, i form av laddfil, överförts till laborationsdatorn kan det exekveras (utföras) och man kan då kontrollera programmets funktion.



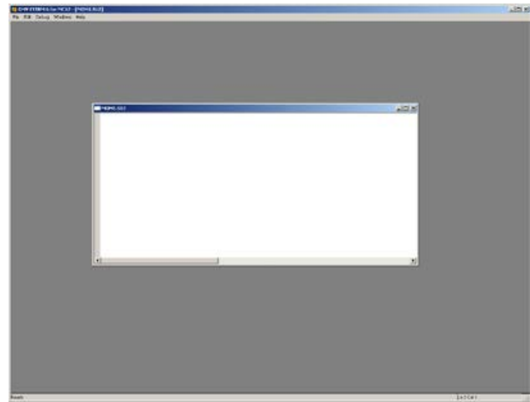
Laborationsdator MC12

Programexekvering kan även utföras i ETERM’s inbyggda simulator och vi återkommer till detta i senare moment.

I slutet av detta moment ska vi ge exempel på hur laddfil och listfil kan se ut men vi börjar med att visa hur man redigerar och assemblerar med ETERM. Starta nu ETERM FÖR MC12 från programmenyn.

Skapa ett assemblerprogram

Du skapar en ny källtextfil genom att välja File | New från menyn. Därefter skriver du in namnet på den fil du vill skapa, skriv nu `Mom1.s12` och klicka på Save. Om du inte anger något filnamnstillägg (eller anger ett annorlunda filnamnstillägg) lägger ETERM automatiskt till .asm. Det är en god ide att alltid skriva in filnamnets korrekta tillägg (S12) redan när filen skapas så att denna sedan enkelt kan identifieras som en källtextfil för MC12. Nu skapas ett nytt fönster:



ETERM*’s editor skapar ett nytt fönster där du kan redigera din källtext.*

Uppgift 1

Skriv nu in källtext med följande programsekvens:

Mom1.s12

```
;
;        Mom1.s12
;
         ORG     $1000
start:   LDAA    $600
         STA     $400
         JMP     start
```

Slut Uppgift 1

Observera hur texteditorn färglägger din text.

- Ett giltigt symbolfält färgas grönt
- En giltig instruktion (eller ett assembler-direktiv) färgas blått
- En giltig operand (eller argument till direktiv) färgas röd.
- Kommentarer färgas grå.

Notera speciellt hur instruktionen:

`STA     $400`

färgas grön, dvs. tolkas som en symbol. Detta beror på att vi (avsiktligt) stavat instruktionen fel, rätt stavning är `STAA`. Låt felet vara kvar, vi ska strax rätta till det. För att spara filen använder du nu File | Save.

- Rader som inleds med ’`;`’ tolkas som kommentarer.
- Med direktivet `ORG` (origin) anger du programmets startadress.
- Dollartecknet anger att påföljande talvärde ska tolkas på hexadecimal form.

Om du snabbt vill göra en kopia av denna källtext gör du File | Save As och väljer ett nytt namn.

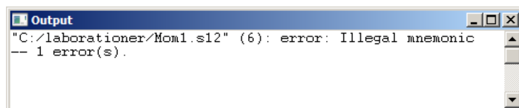
Assemblering

Du assemblerar källtexten genom att välja Debug | Assemble från menyn. Om det aktiva fönstret (fönster med mörkblå namnlist) är en källtextfil assembleras denna direkt. Om det är någon annan typ av fönster som är aktivt öppnas en dialogruta där du får ange namnet på den fil du vill assemblera.

Uppgift 2

1. Assemblera nu filen `Mom1.s12`

Assemblatorn kommer att klaga på den felstavade instruktionen:

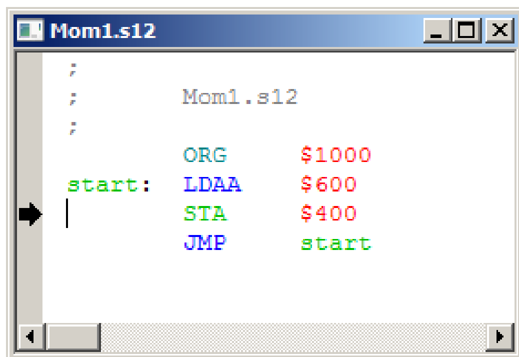


Felutskrift från assemblator

Efter filnamnet, med fullständig sökväg (som kan se annorlunda ut i ditt exempel) skrivs radnummer inom parentes, därefter typ av fel.

'`Illegal Mnemonic`' betyder att det inte finns någon sådan instruktion (STA).

2. Dubbelklicka nu (vänster knapp) på fel-utskriften. Markören i marginalen pekar ut raden i källtextfilen som genererat felet.



OBSERVERA

Det är i allmänhet meningslöst att försöka testa ett program som gett felutskrift vid assembleringen.

3. Rätta felet och assemblera på nytt.

Slut Uppgift 2

Tänk på att korrekt "färgad syntax" inte nöd-vändigtvis innebär att ditt assemblerprogram är korrekt, det är snarare till för att göra dig uppmärksam på enklare stavfel, syntaxfel etc. Därför kan det i bland hända att du får felmeddelanden även om du stavat såväl instruktioner som operander riktigt.

Låt oss uppehålla oss vid det program vi skrivit och assemblerat. Detta innehåller såväl kommentarer som instruktioner och assemblerdirektiv. Vi har också definierat en symbol (`start`).

Programsekvensen placeras med start på adress 1000_{16} i minnet.

```
    ORG    $1000
```

Om direktivet utelämnas kommer adress 0 att användas som startadress, men där finns inget minne så det kommer inte att fungera.

Instruktionen

```
    LDAA   $600
```

(*LoaD Ackumulator A*) läser ett värde till processorns register A. Operanden (`$600`) anger att värdet hämtas från adress 600_{16} i måldatorns minne.

Instruktionen

```
    STAA   $400
```

(*STore Ackumulator A*) skriver värdet från processorns register A till adress 400_{16}.

Instruktionen `JMP` (*jump*) är en ovillkorlig programflödesändring (kallas i bland "hopp"). Till skillnad från att hämta och utför nästa instruktion, vilket är det normala, kommer i stället nästa instruktion att vara den som finns på adressen som anges av `JMP`-instruktionens operand. Eftersom det ofta är krångligt att hålla reda på sådana adresser kan assemblatorn hantera symboler.

```
    JMP    start
```

refererar alltså till symbolen `start`. Symbolen måste också vara definierad någonstans i programmet. En symbol definieras genom att den inleder en rad i källtexten och tilldelas då ett värde av assemblatorn. I vårt exempel får symbolen `start` värdet 1000_{16}.

Om radens första tecken är blankt uppfattas detta som att raden inte definierar någon symbol, då förväntas i stället namnet på någon instruktion (mnemonic) eller ett assemblerdirektiv.

Ett användbart assemblerdirektiv är `EQU` (equate). Det kan användas för att definiera konstanter och fasta adresser. Adressen 600_{16} kan till exempel vara ansluten till en in-port i måldatorns's minne. Vi kan då definiera:

```
InPort   EQU    $600
```

På motsvarande sätt kan en ut-port på adress 400_{16} definieras:

```
OutPort  EQU    $400
```

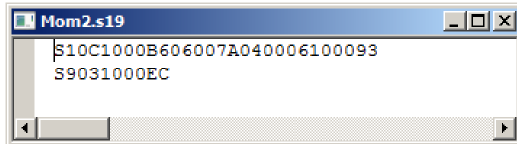
Uppgift 3

1. Ändra i källtexten `Mom1.s12` så att portarna refereras som symbolerna `InPort` och `OutPort` enligt ovan. Spara filen som `Mom2.s12`.
2. Assemblera också `Mom2.s12`.

Slut Uppgift 3

Laddfil och Listfil

Laddfilen används för att överföra programmet (som maskininstruktioner) till måldatorn. I måldatorn finns en funktion som kan ta emot och tolka laddfilsformatet Kommandot skickas automatiskt av ETERM. Assemblatorn skapar laddfilen med filnamnstillägget .S19. Det är knappast nödvändigt att känna till laddfilsformatet i detalj men vi visar ändå hur laddfilen för vårt programexempel `Mom2.s12` ser ut.



Här visar vi bara ett exempel på en laddfil. En fullständig beskrivning av laddfilsformatet finner du i Appendix D.

Listfilen kan i bland vara användbar då man testar sitt program. Listfilen innehåller dels det ursprungliga assemblerprogrammet men dessutom information om den kod som skapats vid assembleringen och på vilka adresser koden hamnat. En del av listfilen för vårt program-exempel ser ut på följande sätt:

```
File: Mom2.lst
                              1. ;
                              2. ;        Mom2.s12
                              3. ;
       0000 0600              4. InPort   EQU     $600
       0000 0400              5. OutPort  EQU     $400
001000                        6.          ORG     $1000
001000 B6 06 00               7. start:   LDAA    InPort
001003 7A 04 00               8.          STAA    OutPort
001006 06 10 00               9.          JMP     start
001009                       10.  |
```

Listfilen innehåller förutom källtexten information om absoluta adresser och den kod som genereras vid assembleringen.

Listfilen används vanligtvis för att identifiera absoluta adresser som man angett med hjälp av symboler. Exempelvis vill man kunna kontrollera vissa variablers värden (minnesinnehåll på någon adress) eller sätta så kallade *brytpunkter* för programexekvering, vi återkommer till detta längre fram.

Uppgift 4

1. Assemblera nu om programmet med följande startadress:

   `ORG     $2000`

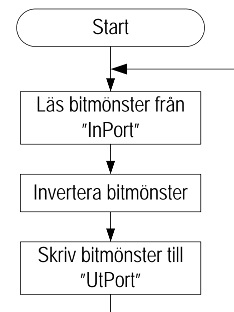
2. Studera listfilen. Vilka skillnader upptäcker du i fälten med adresser och kod/data?

Slut Uppgift 4

Vi avslutar detta avsnitt med att konstruera ett mycket enkelt program som vi fortsätter arbeta med i nästa avsnitt.

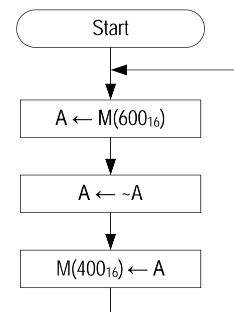
Uppgift 5

1. Utgå ifrån `Mom2.s12` och skapa en ny källtext `Complement.s12` och skriv ett assemblerprogram som utför instruktioner enligt följande flödesdiagram:



Ledning:
1-komplementet av en operand är den bitvis inverterade operanden. Instruktionen
`COMA`
inverterar bitmönstret i register A.

Flödesplanen kan formaliseras ytterligare enligt följande figur:



2. Spara programmet, du ska få tillfälle att undersöka det alldeles strax.

Slut Uppgift 5

Simulatorns grundläggande funktioner

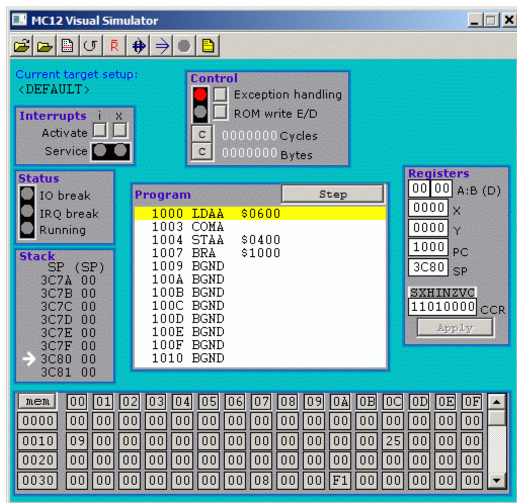
Programdelen *Simulator* i *ETERM* kan användas för att simulera instruktionsutförandet i laborationsdatorn. Med simulatorns hjälp kan du få en första inblick i hur assemblerinstruktioner fungerar. Du kan utföra ett assemblerprogram instruktionsvis och i lugn och ro studera effekterna. Glöm inte att spara dina källtextfiler, de flesta moment utgår från att återanvända kod.

Under detta moment kommer vi att introducera simulatoranvändning. Först visar vi hur du kan ladda program till simulatorn och utföra programmet instruktionsvis. Du kommer också att se hur man kan koppla ”kringenheter” till simulatorn. Simulatorn har många fler funktioner och dessa kommer att presenteras och illustreras efter hand i kommande moment.

Översikt

Vi ska nu fortsätta arbeta med programmet i `Complement.s12` från föregående moment. Eftersom programmet utför in- och ut- matning måste vi dock göra vissa förberedelser innan vi provar programmet i simulatorn.

1. Välj från menyn
   File | Open (`Complement.s12`)
   källtexten öppnas i ett nytt fönster.
2. Assemblera filen.
3. Välj nu från menyn
   Debug | Simulator (`Complement.s19`)



Simulatorns fönster har ett verktygsfält och en rad olika sektioner (avbrott, kontroll, status, stack, program, register och minne).

Till simulatorn finns också en fristående del som kallas "IO-simulator" (Input/Output-simulator). Dess uppgift är att simulera olika omgivningar till laborationsdatorn, dvs. de enheter som inmatning sker från och utmatning sker till. Eftersom IO-simulatorn innehåller olika typer av kringenheter och dessa kan kopplas till olika adresser i *MC12's* adressområde måste vi göra vissa inställningar.

Placera nu markören någonstans i simulator-fönstret utanför programsektionen, högerklicka på pekdonet.

En **popup-meny** visas

Menyn ger flera alternativ, några av menyvalen har du dessutom tillgång till via **verktygsfältet** överst i simulatorfönstret.

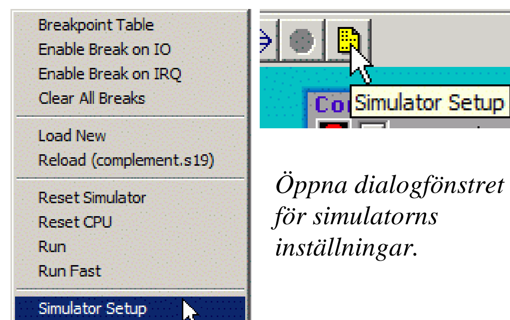
Simulatorns popup- meny visas om du högerklickar utanför programsektionen

Längst ned finner du menyvalet Simulator Setup, börja med att välja denna.

Högerklicka pekdonet och välj

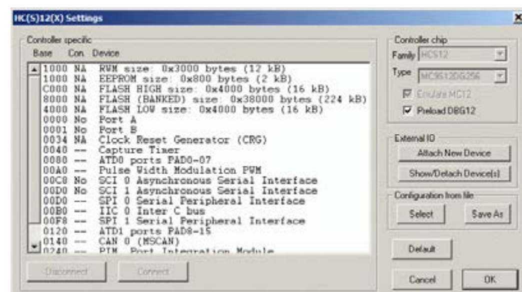
Simulator Setup

eller använd verktygsfältet.



Öppna dialogfönstret för simulatorns inställningar.

Du kan göra en rad olika inställningar av simulatorn även om du som regel klarar du dig långt med standardinställningarna.



Dialogfönstret HCS12 Settings för inställningar av simulatorn.

Vi sammanfattar simulatorns olika inställningsmöjligheter här, du hittar en utförlig beskrivning i ETERM's hjälpsystem.

- Controller specific, här listas de anslutningsmöjligheter som erbjuds direkt av den använda mikrodatorns centralenhet och som dessutom implementerats i simulatorn
- Controller Chip, HCS12 finns i flera familjer med många olika varianter. I denna simulatorversion är varianten förvald, samma som i laborationsdatorn MC12.
- Emulate MC12, förvald, simulatorn konfigureras för att efterlikna en standard MC12 med inbyggd monitor/debugger DBG12. Detta påverkar dispositionen av det interna minnet, anslutningar av kringenheter m.m.
- Preload DBG12, MC12's monitor/debugger DBG12 laddas också till simulatorn, detta gör att simulatorn i hög grad kan efterlikna den verkliga laborationsdatorn MC12.
- External IO här kan du ansluta kringenheter, eller visa (ta bort) tidigare anslutna enheter.
- Configuration from file, här kan du spara och återställa specifika inställningar du gjort. Vi kommer att använda dessa för att åstadkomma unika inställningar för olika uppgifter framöver.
- Default, återställer inställningar till en känd konfiguration, i detta fall för att simulera MC12 i standardutförande.

Anslutning av kringenheter

Vi ska nu koppla samman portar mellan MC12-simulatorn och IO-simulatorns kringenheter. I detta inledande exempel ansluter vi adress 400_{16} till en ut-enhet och adress 600_{16} till en in-enhet.

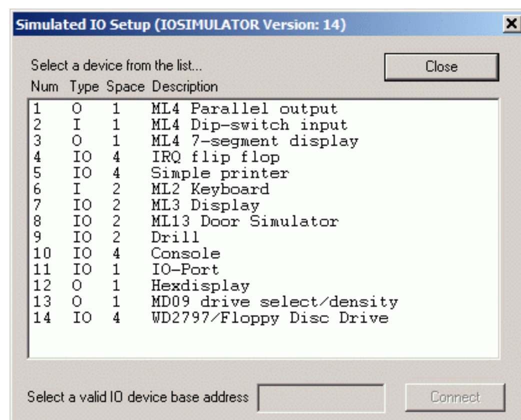
I sektionen

External IO

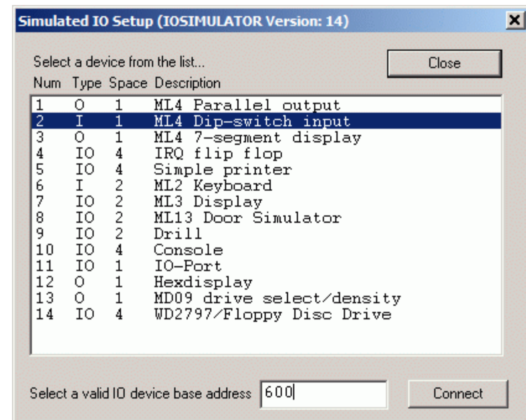
klickar du nu på

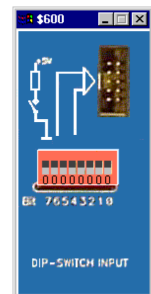
Attach New Device

Nu öppnas en ny dialogbox Simulated IO Setup som visar vilka kringenheter som kan anslutas.



1. Klicka på enheten
   ML4 Dip-switch input
2. skriv in adressen 600 som basadress.
3. Klicka på Connect



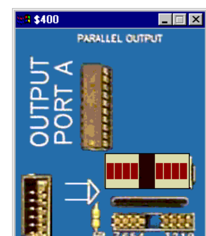


Ett nytt fönster skapas nu, denna enhet i IO-simulatorn används för att simulera en **8-bitars omkopplare**.

Vi ska senare se hur omkopplaren kan användas för att ge indata till vårt program.

Vi kan ansluta flera enheter och för vårt inledande programexempel behöver vi nu också en 'ut-enhet'.

1. Välj denna gång
   ML4 Parallel output
2. skriv in adressen 400 som basadress.
3. Klicka på Connect



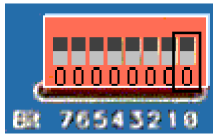
Nu öppnas ytterligare ett nytt fönster med den simulerade utenheten, en **ljusdiodramp**.

Efter att ha öppnat dessa kringenheter för simulering av in- och utmatning stänger du dialogboxen Simulated IO Setup genom att klicka på Close.

> *TIPS*:
>
> Du kan spara aktuella inställningar för den konfiguration du gjort:
>
> Välj (i Configuration from File : Save As – och ange ett lämpligt filnamn för konfigurationen.
>
> Nästa gång du vill konfigurera på samma sätt väljer du Select och anger den fil som innehåller konfigurationen du avser.

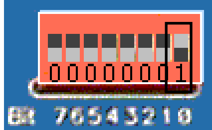
Välj slutligen OK för att stänga fönstret HCS12 Settings för simulatorinställningar.

Omkopplaren ML4 Dip switch input



Fältet för bit 0 representeras av den grå/svarta ytan. Genom att klicka i detta fältet ändrar du omkopplarens utsignal mellan 0 och 1.

Ställ in värdet 1 på omkopplaren genom att klicka på fältet för bit 0.



Din 'klickning' motsvarar en omställning av denna knapp. Du kan ändra varje knapp på samma sätt.

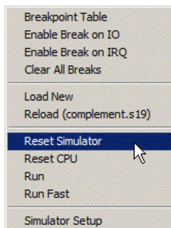
Test av program

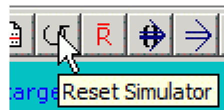
Då vi nu anslutit enheter för såväl in- som utmatning börjar vi bli redo att testa vårt program.

Högerklicka pekdonet och välj

Reset Simulator

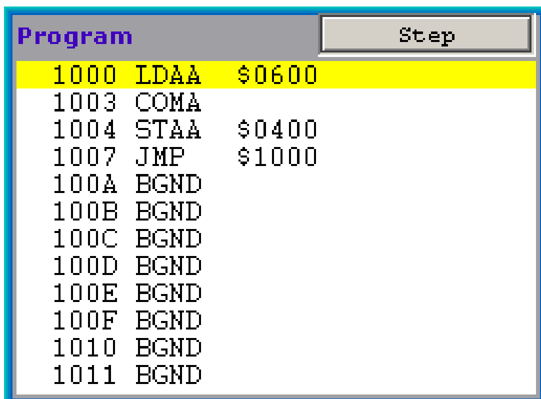
eller klicka på motsvarande ikon i verktygsfältet.





Simulatorn försätts i ett begynnelsetillstånd.

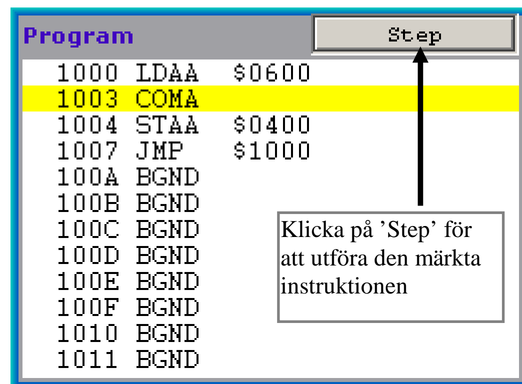
Programsektionen uppdateras och visar nu en disassemblering av simulatorns minne.



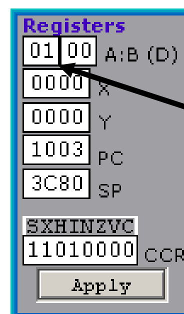
Simulatorn initierar programräknaren så att den innehåller adressen till den första instruktionen i laddfilen. I programsektionen ser du längst till vänster, en adress, därefter en 'mnemonic' och slutligen en operand. Instruktionen som står i tur att utföras har en gul bakgrund.

Kontrollera nu att värdet 1 är inställt på ML4 Dip Switch Input dvs. bit 0 är 1, övriga bitar är 0.

Du låter simulatorn utföra instruktionen som märkts med <u>gul</u> bakgrund genom att klicka på Step.



Instruktionen utförs, nästa instruktion märks med gul bakgrund. Notera hur värdet i processorns ackumulator A ändras i registersektionen.



Det värde du ställde in på omkopplaren hamnar nu i ackumulator A...

Utför ytterligare en instruktion genom att klicka på Step. Simulatorn har nu utfört

```
COMA
```

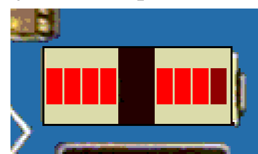
Observera instruktionens inverkan på innehållet i ackumulator A:

Värdet i ackumulator A har bit-komplementerats.

Fortsätt nu och utför nästa instruktion:

```
STAA  $400
```

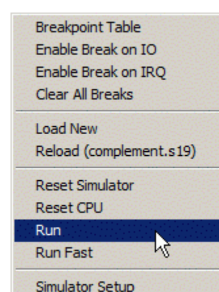
Värdet i ackumulator A skrivs till ljusdiodrampen.

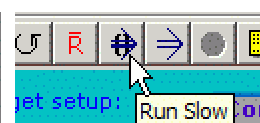


Ljusröda indikatorer tolkar du som '1', de mörkröda tolkas som '0'...

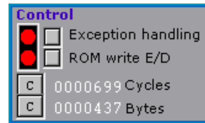
Då du testat ett programs funktion genom att utföra det instruktionsvis kan du också utföra det med Run-kommandot.

Starta simulatorns Run-kommando från menyn eller genom att klicka på dess ikon i verktygsfältet.





Programexekveringen startar, instruktionerna utförs nu i långsam takt, c:a 10 instruktioner per sekund, detta ger möjlighet att följa programflödet utan att för den sakens skull behöva stega igenom programmet. Dioden Running tänds i statussektionen. Observera också hur registerfönstrens innehåll och programsektionen uppdateras mellan varje instruktions exekvering.



I kontrollsektionen kan du se hur många bytes som används, respektive processorcykler som avverkats, under exekveringen. Räknarna nollställs genom att du klickar på intill liggande C-knapp.

Om du väljer Run Fast i stället för Run simuleras instruktionsexekveringen med maximal hastighet. Simulatorns sektioner uppdateras då inte på samma sätt men du ser hur räknarna i statussektionen uppdateras och att dioden Running lyser.

Då du valt Run (eller Run Fast) kommer verktygsfältet att få ett annat utseende. Det enda aktiva alternativet är nu Halt-ikonen. På motsvarande sätt, om du högerklickar, kommer popup-menyn att ha ett annorlunda utseende, välj Halt för att avbryta den simulerade instruktionsexekveringen.

Uppgift 6

Ställ in följande olika värden på omkopplaren – utför programmet med Run, prova också med att utföra programmet med Run Fast. Under simulatorns programexekvering ändrar du inställningarna på omkopplaren och observerar ändringarna hos ljusdiodrampen, fyll i följande tabell:

Inställt värde (binärt)	Avläst värde (binärt)
1111 0000	
1010 1010	
1100 0011	

Slut Uppgift 6

Efter att ha undersökt instruktionen COM, går vi nu vidare och undersöker ytterligare instruktioner hos MC68HCS12.

Adresseringssätt

MC68HCS12 har en rad olika *adresseringssätt*, dvs. metoder att bestämma operanden. I de inledande exemplen har vi redan stiftat bekantskap med några och du ska nu få tillfälle att bekanta dig med några grundläggande varianter. Vi återkommer längre fram till ytterligare adresseringssätt.

Inherent

Operanden anges implicit i instruktionen, dvs. operandfält saknas för dessa instruktioner, några exempel på sådana instruktioner är:

Mnemonic	Funktion	Operation
NOP	Ingen operation	
TAB	Kopiera A till B	(A)→B
TBA	Kopiera B till A	(B)→A
ABA	Addera B till A	(A)+(B) → A
ABX	Addera B till X	(X)+(B) → X
SBA	Subtrahera B från A	(A)-(B) → A
INCA	Inkrementera A	(A)+1 → A
INX	Inkrementera X	(X)+1 → X

Uppgift 7

1. Placera värdet 10_{16} i register A, värdet 20_{16} i register B, värdet 5556_{16} i register X och klicka på Apply.
2. Utför följande instruktionssekvens i simulatorn, observera registerinnehållen. Ange, efter varje instruktion registrens nya innehåll.

Instruktion	Register
	A = 10_{16} B = 20_{16} X = 5556_{16}
ABA	A =
ABX	X =
INCA	A =
INX	X =
SBA	A =

Slut Uppgift 7

Immediate:

Omedelbar adressering, operanden anges som en konstant. Detta adresseringssätt kan användas tillsammans med merparten av instruktionerna för HCS12.

Operanden kan vara något värde K, exempelvis så som i uttrycket:

Utport = Inport + K

Det kan också vara någon adress till en speciell minnesposition.

I de närmast följande uppgifterna har du möjlighet att bekanta dig med en rad användbara instruktioner och dessa enkla adresseringssätt.

Binära logiska operationer

Ytterligare exempel från gruppen av logikinstruktioner har vi i form av

`ANDA` (*AND,* logiskt OCH)
`ORAA` (*OR,* logiskt ELLER)
`EORA` (*XOR*, logiskt exklusivt ELLER)

Instruktionen AND utför bitvis logiskt OCH mellan operanderna. Låt "`InPort`" symbolisera det värde vi ställer in på omkopplaren. Låt "`UtPort`" symbolisera det värde som skrivs till ljusdiodrampen. Låt `K` symbolisera någon konstant. Vi kan då utföra operationen:

`UtPort ← K AND InPort`

med följande instruktionssekvens:

```
LDAA  InPort
ANDA  #K
STAA  UtPort
```

Uppgift 8

Skriv en instruktionssekvens för operationen

UtPort ← $0F_{16}$ AND InPort

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiod-rampen och fyll i följande tabell:

InPort (binärt)	Avläst värde (binärt) UtPort
1111 0000	
1010 1010	

Slut Uppgift 8

Uppgift 9

Ändra nu programmet för att utföra operationen:

UtPort ← $0F_{16}$ OR InPort

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiod-rampen och fyll i följande tabell:

InPort (binärt)	Avläst värde (binärt) UtPort
1111 0000	
1010 1010	

Slut Uppgift 9

Uppgift 10

Slutligen, ändra nu programmet för att utföra operationen

UtPort ← $0F_{16}$ XOR InPort

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiod-rampen och fyll i följande tabell:

InPort (binärt)	Avläst värde (binärt) UtPort
1111 0000	
1010 1010	
1100 0011	

Slut Uppgift 10

Aritmetiska operationer

Vi använder aritmetiska operationer för att utföra de fyra olika räknesätten addition, subtraktion, multiplikation och heltalsdivision. Operationerna är binära, dvs:

`resultat ← operand1 OP operand2`

Addition och subtraktion är väl kända operationer. Om `operand2` är en konstant `K` kan vi använda:

`ADDA #K` eller
`SUBA #K`

Uppgift 11

Undersök instruktionen `ADDA`, skriv en programsekvens som utför:

`UtPort ← InPort + K`

Använd ackumulator A för operationen.

Låt `K` vara 5. Ställ in följande olika värden på omkopplaren – utför programmet - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär 2-komplementsform (tal <u>med</u> tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
0000 0111			
1111 1001			
0111 1110			

Slut Uppgift 11

Uppgift 12

Undersök instruktionen `SUBA`, skriv en programsekvens som utför:

`UtPort ← InPort - K`

Använd ackumulator A för operationen.

Låt `K` vara 15. Ställ in följande olika värden på omkopplaren – utför programmet - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär 2-komplementsform (tal <u>med</u> tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
0000 0111			
1111 1001			
1000 0010			

Slut Uppgift 12

För subtraktion har vi också ett specialfall för operationen:

`resultat ← 0-operand`

vilket vi också skriver:

`resultat ← -operand`

dvs. unärt minus. Operationen är så vanlig att den motsvaras av en speciell instruktion:

`NEGA` ($A \leftarrow -A$)

Uppgift 13

Undersök instruktionen `NEGA`. Skriv en programsekvens som utför:

```
UtPort ←  -InPort
```

Använd ackumulator A för operationen.

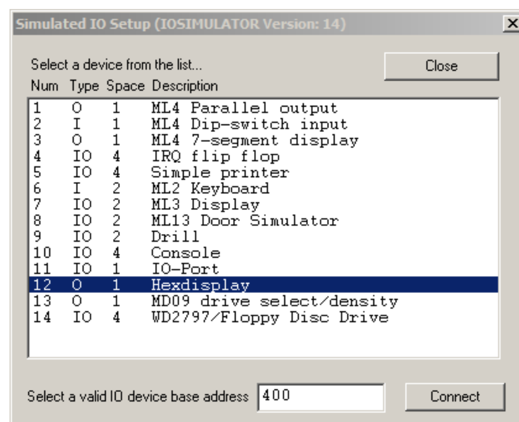
Ställ in följande olika värden på omkopplaren – utför programmet - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär 2-komplementsform (tal med tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
0100 0000			
1100 0000			
1000 0000			

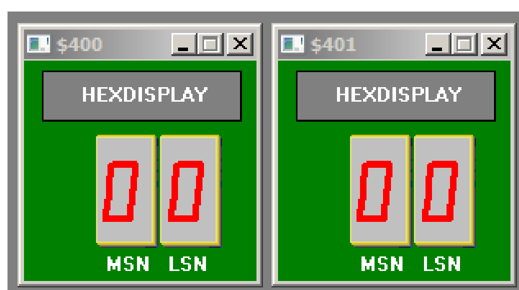
Slut Uppgift 13

Multiplikation av 8-bitars tal (utan tecken) kan utföras med instruktionen MUL. Operanderna placeras först i ackumulator A respektive ackumulator B. Det 16-bitars resultatet ersätter operanderna och finns efter operationen i registerparet A:B (även kallat D).

För att undersöka denna instruktion är inte längre ljusdiodrampen en lämplig utmatningsenhet. För denna övning är det bättre att använda enheten *Hexdisplay*, som visar ett 8-bitars värde som två hexadecimala siffror.



För att visa hela resultatet, dvs 16 bitar, är det lämpligt att ansluta två enheter på adresserna 400_{16} och 401_{16}. Vi ska sedan visa de mest signifikanta 8 bitarna på indikatorn med adress 400_{16} och de minst signifikanta bitarna på indikatorn med adress 401_{16}.



Uppgift 14

Undersök instruktionen `MUL`.

En programsekvens som utför:

```
UtPort ←  InPort * 3
```

kan skrivas:

```
LDAA    InPort
LDAB    #3
MUL
STAA    UtPort
STAB    UtPort+1
```

`STAA` och `STAB` kan ersättas av den 16-bitars instruktionen

```
STD     UtPort
```

Ställ in följande olika värden på omkopplaren – utför programmet - läs av indikatorerna med hexadecimala siffror och fyll i följande tabell. Översätt från hexadecimal form (tal utan tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort		
binärt	dec.	hex(400,401)		dec.
00000001	1			
00001010	10			
00101000	40			
00110010	50			
01100100	100			

Slut Uppgift 14

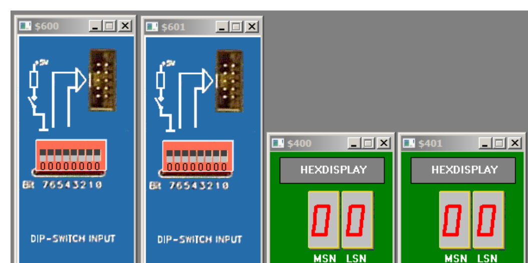
Division av två heltal A och B definieras som bekant av: $\frac{A}{B} = Q + \frac{r}{B}$

där även Q (kvoten) och r (resten) är heltal.

- Med *heltalsdivision* menar vi den del av operationen som resulterar i Q.
- Den del av operationen som resulterar i r kallar vi *restdivision*.

För heltalsdivision av två 16-bitars tal utan tecken används instruktionen `IDIV` (*Integer Divide*) och för heltalsdivision av två 16-bitars tal med tecken används `IDIVS` (*Integer divide signed*). Operanderna placeras först i register D respektive register X. Efter utförd operation finns kvoten av divisionen i register X medan resten har placerats i register D.

Inför nästa uppgift är det lämpligt att lägga till ytterligare en omkopplare på adress 601_{16} som inmatningsenhet så att ett 16-bitars tal kan anges som operand.



Uppgift 15

Undersök instruktionerna `IDIV` och `IDIVS`.

En programsekvens som utför:

```
UtPort ←  InPort / 30
```

kan, för tal utan tecken, skrivas:

```
LDD    Inport
LDX    #30
IDIV
STX    UtPort    ; kvoten
STD    UtPort    ; resten
```

Ställ in följande olika värden på omkopplarna – stega igenom programmet - läs av indikatorerna med hexadecimala siffror och fyll i följande tabell.

InPort		Avläst värde UtPort IDIV			
binärt	dec.	kvot		rest	
		400	401	400	401
00000000 00100001	33				
00001100 11100100	3300				
10000000 11101101	33005				

Upprepa nu med instruktionen IDIVS.

Operand = Inport		Avläst värde UtPort IDIVS			
binärt	dec.	kvot		rest	
		400	401	400	401
00000000 00100001	33				
00001100 11100100	3300				
10000000 11101101	-32531				

Översätt från hexadecimal form till decimal form och sammanställ resultaten i följande tabell.

Operation dec.	Resultat kvot	rest
33/30		
3300/30		
33005		
-32531		

Slut Uppgift 15

Bortsett från division har vi behandlat aritmetiska operationer på 8 bitars tal, det finns ingen divisionsinstruktion för 8-bitars tal, men det är också enkelt att utvidga de tre övriga operationerna till 16-bitars tal.

- För addition använder vi `ADDD` (*add D*).
- Subtraktion använder `SUBD` (*subtract D*).
- För multiplikation av tal utan tecken använder vi `EMUL` (*extended multiplication*).
- För multiplikation av tal med tecken använder vi `EMULS` (*extended multiplication signed*).

Uppgift 16

Undersök instruktionen `ADDD`, skriv en programsekvens som utför:

```
UtPort ← InPort + K
```

Låt `K` vara 150. Ställ in följande olika värden på omkopplarna – utför programmet - läs av indikatorerna med hexadecimala siffror och komplettera tabellen.

InPort		Avläst värde UtPort	
binärt	dec.	hex.	dec.
00001100 11100100	3300		
01111111 11111100	32764		

Slut Uppgift 16

Uppgift 17

Undersök instruktionen `SUBD`, skriv en programsekvens som utför:

```
UtPort ← InPort - K
```

Låt `K` vara 237 Ställ in följande olika värden på omkopplarna – utför programmet - läs av indikatorerna med hexadecimala siffror och komplettera tabellen.

InPort		Avläst värde UtPort	
binärt	dec.	hex.	dec.
00001100 11100100	3300		
10000000 11100011	-32541		

Slut Uppgift 17

Multiplikationsinstruktionerna `EMUL` och `EMULS` multiplicerar samman de 16 bitars talen i register D och register Y, det 32-bitars resultatet ersätter operanderna med de mest signifikanta 16 bitarna i register Y och de minst signifikanta 16 bitarna i register D.

Uppgift 18

Undersök instruktionerna `EMUL` och `EMULS`.

En programsekvens som utför:

```
UtPort ←  InPort * 30
```

kan, för tal utan tecken, skrivas:

```
LDD    Inport
LDY    #30
EMUL
STY    UtPort    ; mest sign.
STD    UtPort    ; minst sign.
```

Ställ in följande olika värden på omkopplarna – stega igenom programmet - läs av indikatorerna med hexadecimala siffror och fyll i följande tabell.

InPort		Avläst värde UtPort EMUL			
binärt	dec.	reg Y		reg D	
		400	401	400	401
00000001 01001000	328				
11111110 10111000	-328				

Upprepa operationerna med instruktionen `EMULS`.

InPort		Avläst värde UtPort EMULS			
binärt	dec.	reg Y		reg D	
		400	401	400	401
00000001 01001000	328				
11111110 10111000	-328				

För in resultaten från hexadecimal form, översätt också till decimal form (16 respektive 32-bitars tolkning) och sammanställ i följande tabell.

	operation	Decimalt förväntat	Hexadecimalt (register Y/D)	Decimalt 32 bit (register Y/D)	Decimalt 16 bit (register D)
EMUL	328*30	9840			
EMUL	-328*30	-9840			
EMULS	328*30	9840			
EMULS	-328*30	-9840			

Slut Uppgift 18

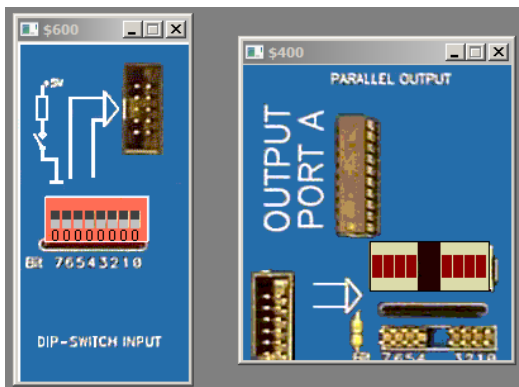
Skift instruktioner

Skiftoperationer används för att "flytta" bitar ett eller flera steg. HCS12 stödjer skiftoperationer med tre olika instruktioner:

logiskt skift	(LS)
aritmetiskt skift	(AS)
rotation med *carry*	(RO)

Logiskt och aritmetiskt skift används speciellt under aritmetiska operationer. Vänsterskift innebär som bekant att operanden multipliceras med 2, medan högerskift innebär att operanden divideras med 2.

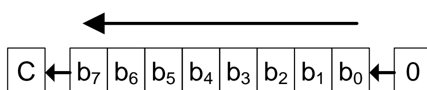
För följande uppgifter med olika skift-operationer är IO-simulatorerna *Dip-Switch Input* och *Parallel Output* lämpliga.



Logiskt skift

LSLA, LSLB ***logical shift left***

Såväl minnesinnehåll som innehållet i någon ackumulator (A eller B) kan användas som operand. Vi nöjer oss här med att undersöka *ackumulatorskift*. Operandens (dvs innehållet i ackumulator A eller B) mest signifikanta bit kopieras till C-flaggan. En nolla skiftas in i den minst signifikanta bitens position.



Uppgift 19

Undersök instruktionen LSL. Skriv en programsekvens som utför:

```
UtPort = (InPort) << 1
```

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär form (tal <u>utan</u> tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
01110000			
11110000			

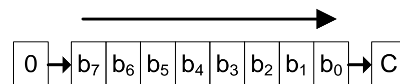
Fyll också i följande tabell. Översättningen från binär form till decimal form ska nu ske för tal **med** tecken.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
01110000			
11110000			

Slut Uppgift 19

LSR ***logical shift right***

Operandens minst signifikanta bit kopieras till C. En nolla skiftas in i den mest signifikanta positionen



Uppgift 20

Undersök instruktionen LSRA. Skriv en programsekvens som utför:

```
UtPort = (InPort) >> 1
```

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär form (tal <u>utan</u> tecken) till decimal form och fyll i respektive kolumner.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
01110000			
11110000			

Fyll också i följande tabell. Översättningen från binär form till decimal form ska nu ske för tal <u>med</u> tecken.

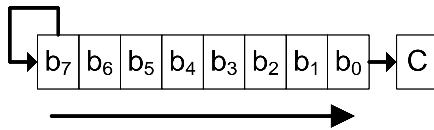
InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
01110000			
11110000			

Slut Uppgift 20

Aritmetiskt skift

Då tal med inbyggt tecken ska skiftas används aritmetiska skift

ASRA, ASRB *arithmetic shift right*



Observera hur teckenbiten kopieras vid skiftet.

Uppgift 21

Undersök instruktionen ASRA. Skriv en programsekvens som utför:

```
UtPort = (InPort) >> 1
```

Ställ in följande olika värden på omkopplaren – utför det nya programmet med Run - läs av ljusdiodrampen och fyll i följande tabell. Översätt från binär form (tal utan tecken) till decimal form och fyll i respektive kolumner.

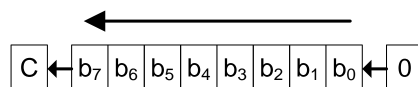
(Inport)		Avläst värde Utport	
binärt	dec.	binärt	dec.
01110000			
11110000			

Fyll också i följande tabell. Översättningen från binär form till decimal form ska nu ske för tal med tecken.

InPort		Avläst värde UtPort	
binärt	dec.	binärt	dec.
01110000			
11110000			

Slut Uppgift 21

ASLA, ASLB *arithmetic shift left*

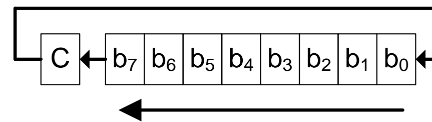


Operationen är identisk med logiskt vänsterskift men förekommer ändå som egen mnemonic av symmetriskäl. Tittar vi närmre i instruktions-listan ser vi att instruktionerna ASL och ROL i själva verket har samma operationskod.

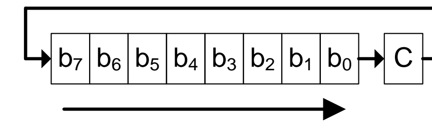
(Carry) Rotation

Rotation med "carry" kan med fördel användas för att implementera skiftoperationer på tal som är större än de tillgängliga registren. En inskiftad bit hämtas alltid från C därefter hamnar den utskiftade biten hamnar C. C-flaggan kan därför ses som en "mellanlagring" av biten som skiftas mellan två operationer.

ROLA, ROLB *rotate left*

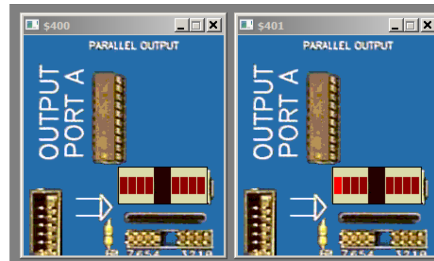


RORA, RORB *rotate right*



Uppgift 22

"Rinnande ljus", för denna uppgift är det lämpligt att använda två ML4 Parallel Output på adresserna 400_{16} respektive 401_{16}.



Undersök instruktionerna ROLA och ROLB med följande programsekvens:

```
          ORG     $1000
          CLRA
          LDAB    #1
rotate:   STAA    $400
          STAB    $401
          ROLB
          ROLA
          JMP     rotate
```

Stega igenom sekvensen, kontrollera att ljusdiodrampens segment tänds ett i taget.

Prova nu programmet med Run.

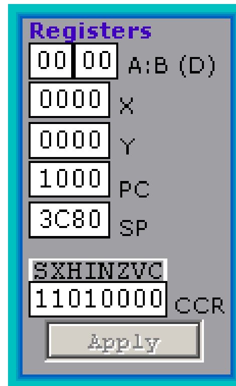
Prova slutligen med Run Fast, vad händer?

Uppenbarligen har exekveringshastigheten stor betydelse för programsekvensens funktion.

Spara programmet i en fil RinnandeLjus.s12, vi återkommer till denna i slutet av detta kapitel.

Slut Uppgift 22

Simulatorns olika sektioner



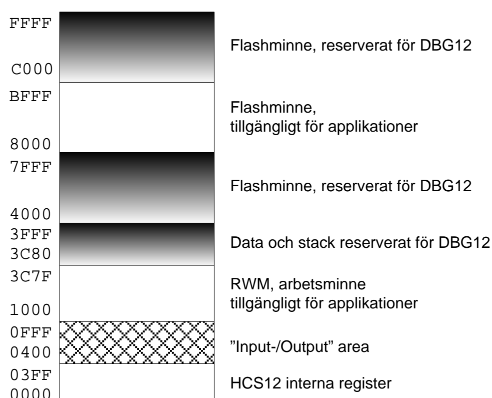
Simulatorns registersektion visar registeruppsättningen hos mikrocontrollern MC68HCS12.

Du kan ändra registrens innehåll manuellt genom att klicka på fönstret för det register du vill ändra och skriva in ett nytt värde. För register CCR tolkas värdet som på binär form, för övriga register tolkas värdet som på hexadecimal form. Då du ändrat ett registerinnehåll aktiveras knappen Apply, <u>du måste då klicka på denna</u> för att ändringarna ska registreras i simulatorn.

MC68HCS12 Registeruppsättning	
Namn	Beskrivning
A	Ackumulator A används för aritmetiska, logiska, skift-operationer etc.
B	Samma användning som ack A
D	Ackumulatorerna A och B, tillsammans bildar register D med ack A innehållande de mest signifikanta bitarna (A:B)
X	Indexregister, används för att lagra adresser
Y	Samma användning som X
PC	Programräknare
SP	Stackpekare
CCR	"Condition Code Register" – innehåller olika statusbitar.

Minnesdisposition – MC12 simulatorn

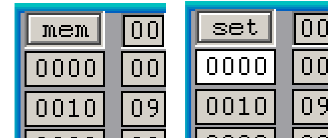
Följande figur beskriver hur minnet disponeras under standardinställningarna för MC12-simulatorn. Figuren visar det 64 kbyte stora linjära adressområdet. Figuren är inte fullständig eftersom det bank-switchade minnet här har utelämnats. En komplett beskrivning finns dock i appendix.



Adressrymd (64 kbyte) för MC12/DBG12

Simulatorns hantering av minnet

Minnesinnehåll visas i block om 64 bytes. Du kan ändra visningsintervallet med hjälp av rullningslisten till höger. Du kan också ange en startadress för blocket som ska visas. Klicka på knappen mem längst upp till vänster. Det första fönstret, visande en basadress för adressintervallet, blir nu skrivbart.



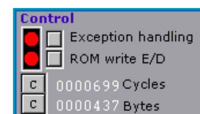
Skriv in den adress du vill visningen ska börja med och klicka sedan på samma knapp, som nu heter set.

Read Write Memory (RWM)

Följande figur visar adressutrymmet 1000_{16}-$103F_{16}$, som figuren *Adressrymd MC12* visar är detta minne av RWM-typ (*read-write memory*). I skrivbart minne kan minnesinnehållet ändras genom att skriva in ett nytt (hexadecimalt) värde. Ändringsbara minnesinnehåll har en *vit* bakgrund.

mem	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1000	B6	06	00	41	7A	04	00	20	F7	00	00	00	00	00	00	00
1010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Read Only Memory (ROM)



En annan typ är de minnesareor som är reserverade som ROM *-typ* (*read-only memory*).

I ett verkligt system kan sådana minnen normalt sett inte skrivas men i simulatorn vill vi i bland kunna initiera detta minne. Av denna anledning finns knappen ROM Write E/D (*ROM Write Enable/Disable*). Då den intilliggande dioden lyser (är ljust röd) behandlas ROM som skrivbart minne av simulatorn.

För icke-skrivbart RO-minne har fönstren för minnesinnehåll *grå* bakgrund. Du kan inte längre ändra innehållet.

mem	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
4000	F7	3D	92	27	0C	EE	82	34	CE	F5	B3	34	16	E6	78	1B
4010	84	3D	1B	9E	F7	3D	92	18	27	00	39	87	C7	6C	80	E7
4020	F3	00	04	27	18	E6	F3	00	04	B7	14	3B	16	C0	80	1B
4030	82	EE	84	08	6E	84	EE	80	08	6E	80	20	E2	EC	80	8C

IO-area

MC12's adressområde 0400_{16}-$0FFF_{16}$ har avdelats för användning tillsammans med kringenheter. Simulatorns hantering av detta intervall efterliknar hårdvaran, dvs. minnes-innehållet ser ut att vara samma som de 8 mest signifikanta bitarna av adressbussen. Detta beror av MC12's multiplexade adress/databuss men vi behöver inte gå in närmre på detta här.

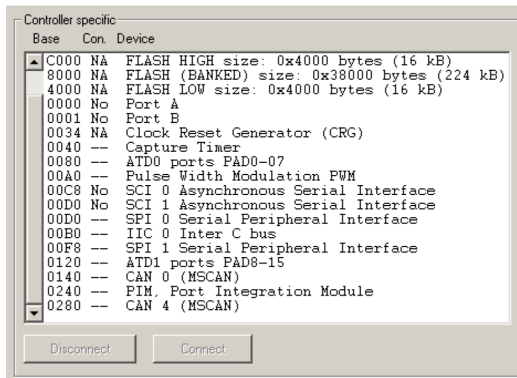
mem	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0400	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04
0410	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04
0420	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04
0430	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04	04

IO-arean är inte skrivbar genom simulatorn, däremot fungerar programstyrd in- och utmatning, som vi redan sett enkla exempel på.

HCS12 interna register

Adressområdet 0-$3FF_{16}$ upptas av register hos microcontroller'n MC68HCS12. Dessa är inte skrivbara genom simulatorn. För vissa register fungerar programstyrd in- och utmatning, alla funktioner är dock inte implementerade i simulatorn.

Sektionen Controller specific i simulatorns inställningar indikerar vilka funktioner som är tillgängliga:



Controller specific

Base	Con.	Device
C000	NA	FLASH HIGH size: 0x4000 bytes (16 kB)
8000	NA	FLASH (BANKED) size: 0x38000 bytes (224 kB)
4000	NA	FLASH LOW size: 0x4000 bytes (16 kB)
0000	No	Port A
0001	No	Port B
0034	NA	Clock Reset Generator (CRG)
0040	--	Capture Timer
0080	--	ATD0 ports PAD0-07
00A0	--	Pulse Width Modulation PWM
00C8	No	SCI 0 Asynchronous Serial Interface
00D0	No	SCI 1 Asynchronous Serial Interface
00D0	--	SPI 0 Serial Peripheral Interface
00B0	--	IIC 0 Inter C bus
00F8	--	SPI 1 Serial Peripheral Interface
0120	--	ATD1 ports PAD8-15
0140	--	CAN 0 (MSCAN)
0240	--	PIM, Port Integration Module
0280	--	CAN 4 (MSCAN)

Disconnect Connect

- NA, enheten är tillgänglig i simulatorn men kan inte anslutas som en kringenhet till IO-simulator.
- No, tillgänglig i simulatorn, för tillfället inte ansluten till kringenhet i IO-simulatorn.
- Yes, tillgänglig i simulatorn och för tillfället ansluten till kringenhet i IO-simulatorn. En enhet i simulatorn kan bara anslutas till <u>en</u> kringenhet i IO-simulatorn åt gången.
- --, ej tillgänglig i simulatorn.

Programflödeskontroll

Med "programflödesändring" menar man att det gängse arbetssättet "läs från nästa adress och tolka innehållet som instruktion", avbryts, och programflödet ändras. Någon speciell instruk-tion som utför programflödesändringen används då.

Ovillkorlig programflödesändring

Vi har tidigare sett exempel på användning av:

```
JMP      <adress>
```

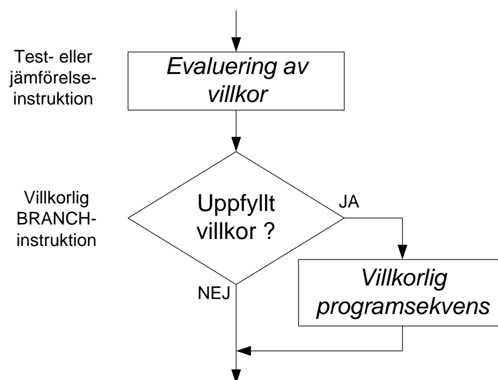
Instruktionens operand bestämmer den adress som anger var nästa instruktion ska hämtas. Det finns en annan instruktion med samma funktion, nämligen

```
BRA      <adress>
```

Instruktionerna har samma effekt men kodas annorlunda. För närvarande har detta mindre betydelse och du kan betrakta dom som likvärdiga. Vi kommer, fortsättningsvis, att använda båda varianterna `JMP` och `BRA`. (*Jump* resp. *BRanch Always*).

Villkorlig programflödesändring

Instruktionstypen *Branch* ("gren") används också för att ange så kallade "villkorliga programflödesändringar", dvs. beroende på hur någon test har utfallit så utförs antingen den ena "grenen" eller den andra. Vi ska nu titta närmare på hur detta är tänkt att användas.



Villkorsevaluering

Villkorsevaluering kan göras explicit med speciella test- eller jämförelse- instruktioner. För jämförelse av två operander kan någon av följande instruktioner (*compare*) användas:

Mnemonic	Funktion	Operation
CBA	Jämför B med A	(A)-(B)
CMPA	Jämför A med minne	(A)-(M)
CMPB	Jämför B med minne	(B)-(M)
CPD	Jämför D med minne	(A:B)-(M:M+1)
CPS	Jämför SP med minne	(SP)-(M:M+1)
CPX	Jämför X med minne	(X)-(M:M+1)
CPY	Jämför Y med minne	(Y)-(M:M+1)

Observera att även andra instruktioner sätter flaggor på samma sätt som dessa. Det är

exempelvis överflödigt att använda en jämförelseinstruktion direkt efter en aritmetisk instruktion.

För test av en operand kan någon av följande instruktioner användas:

Mnemonic	Funktion	Operation
TST	Testa minnesinnehåll	(M)-$00
TSTA	Testa register A	(A)-$00
TSTB	Testa register B	(B)-$00

Testinstruktionerna påverkar endast Z- och N-flaggorna. Även dessa kan i vissa fall utelämnas då många andra instruktioner (speciellt LOAD) påverkar flaggorna på samma sätt.

Villkorstest

14 olika villkor (hoppinstruktioner) kan anges,

Assemblersyntaxen är:

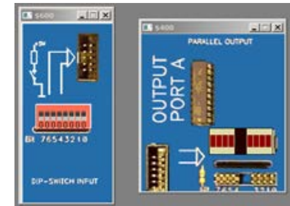
```
Bcc    <symbol>
```

Där `cc` (*condition codes*) står för något flaggvillkor givet i tabellen nedan och `<symbol>` är någon lägesangivelse i programmet. Flaggorna N,Z,V och C som används för att bilda de olika villkoren finns samlade i CCR-registret (*Condition Code Register*) i simulatorns registersektion.

Mnemonic	Funktion	Villkor
Enkla flaggtest		
BCS/BLO	"Hopp" om *carry*	C=1
BCC/BHS	"Hopp" om ICKE *carry*	C=0
BEQ	"Hopp" om *zero*	Z=1
BNE	"Hopp" om ICKE *zero*	Z=0
BMI	"Hopp" om *negative*	N=1
BPL	"Hopp" om ICKE *negative*	N=0
BVS	"Hopp" om *overflow*	V=1
BVC	"Hopp" om ICKE *overflow*	V=0
Jämförelse av tal utan tecken		
BHI	Villkor: R>M	C + Z = 0
BHS/BCC	Villkor: R≥M	C=0
BLO/BCS	Villkor: R<M	C=1
BLS	Villkor: R≤M	C + Z = 1
Jämförelse av tal med tecken		
BGT	Villkor: R>M	$Z + (N \oplus V) = 0$
BGE	Villkor: R≥M	$N \oplus V = 0$
BLT	Villkor: R<M	$N \oplus V = 1$
BLE	Villkor: R≤M	$Z + (N \oplus V) = 1$

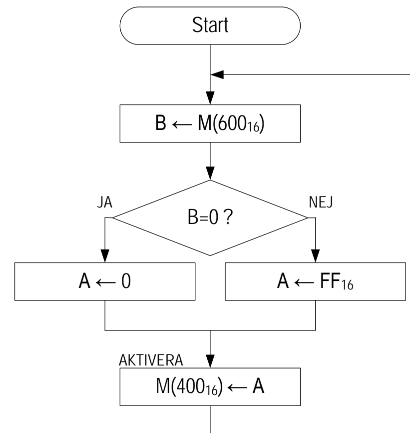
Uppgift 23

I denna uppgift implementeras en enkel strömställarfunktion. Använd IO-simulatorerna ML4 Dip-Switch Input och ML4 Parallel Output.



Skriv en programsekvens som läser från strömställaren, om någon av omkopplarna är i läge 1 ska samtliga indikatorer på ljusdiodrampen tändas. Om alla omkopplare är i läge 0 ska hela ljusdiodrampen släckas.

Funktionen beskrivs i följande flödesdiagram.



Komplettera följande ofullständiga programsekvens som en direkt implementering av flödesdiagrammet ovan.

	ORG	$1000
Start:	LDAB	$600
		JA
		NEJ
JA:	CLRA	
NEJ:		
AKTIVERA:	STAA	$400
	BRA	Start

Kontrollera att programsekvensen fungerar som avsett.

Slut Uppgift 23

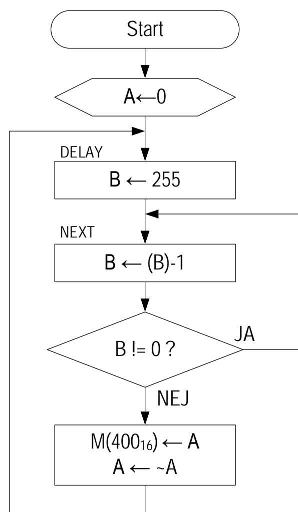
I en tidigare uppgift, "rinnande ljus", såg vi exempel på att exekveringshastigheten kan ha betydelse för funktion av en programsekvens.

Vi ska nu se hur vi på ett enkelt sätt kan fördröja programexekveringen för att på så sätt skapa en "blinkers"-funktion tillräckligt långsam för att uppfattas.

Uppgift 24

I denna uppgift ska en ”blinkers”-funktion implementeras. Här räcker det med att använda IO-simulatorern ML4 Parallel Output.

Programsekvensens funktion beskrivs av följande flödesdiagram:



1. Komplettera följande ofullständiga programsekvens som en direkt implementering av flödesdiagrammet ovan.

	ORG	$1000
Start:	CLRA	
DELAY:		
NEXT:		
	BRA	DELAY

2. Kontrollera med Run Fast att programsekvensen fungerar som avsett.

Den maximala fördröjningen bestäms här av att det största värde som kan laddas i B-registret är 255. För vissa funktioner kan denna maximala fördröjning trots allt vara för kort.

3. Modifiera programsekvensen så att register X används i stället för register B.
4. Prova med följande startvärden i X och uppskatta (grovt) fördröjningen.

Startvärde X	Fördröjning c:a (sekunder)
1000	
2000	
2500	
3000	

Slut Uppgift 24

Statisk minnesinitiering

Med ”statisk minnesinitiering” menar man att ett bestämt värde placerats på en given adress innan programmet startas.

Detta görs enkelt med assemblerdirektivet `FCB` (*Form Constant Byte*) som instruerar assemblatorn att placera ett värde, i måldatorns primärminne.

```
ORG   <adress>
FCB   <värde>
```

Flera argument (värden) kan ges med direktivet. Dessa måste då skiljas åt med kommatecken. Observera att inga blanka tecken får finnas mellan värden och kommatecken.

Uppgift 25

Följande exempel illustrerar en tabell, med start på adress 3000_{16} med de decimala värdena 1-9.

```
    ORG   $3000
    FCB   1,2,3,4,5,6,7,8,9
end:
```

1. Skapa en sådan tabell, dvs skriv de nödvändiga direktiven och assemblera din källtext.
2. Undersök listfilen, vilket värde får symbolen `end` ?

   ____

3. Ladda till simulatorn och undersök minnesinnehållet. Kontrollera att minnet initierats på rätt sätt.

Slut Uppgift 25

Om tabellen ska innehålla större tal än vad som ryms i en *byte*, kan man i stället använda `FDB` (*Form Double Byte*).

Uppgift 26

Även detta exempel illustrerar en tabell, med start på adress $(3000)_{16}$ som innehåller de decimala värdena 1-9.

```
    ORG   $3000
    FDB   1,2,3,4,5,6,7,8,9
end:
```

1. Skapa en sådan tabell, dvs skriv de nödvändiga direktiven och assemblera din källtext.
2. Undersök listfilen, vilket värde får symbolen `end` ?

   ____

3. Ladda till simulatorn och undersök minnesinnehållet. Kontrollera att minnet initierats på rätt sätt.

Slut Uppgift 26

Ytterligare adresseringssätt

Vi fortsätter nu med att undersöka ytterligare några av MC68HCS12's olika adresseringssätt.

Extended:

Data för instruktionen finns på någon adress i minnet. Vi har sett exempel på användning av detta adresseringssätt med så kallad minnesavbildad in- utmatning. Då har vi läst från och skrivit till speciella kringenheter anslutna i adressrummet.

Adresseringssättet används också för att läsa från och skriva till godtyckliga minnesceller.

Uppgift 27

Utför följande instruktionssekvens i simulatorn, observera registerinnehållen. Ange, efter varje instruktion det register som ändras samt dess nya innehåll.

```
        ORG     $1000
        LDAA    Data      A=
        LDAB    Data+1    B=
        LDX     $3000     X=
        LDY     Data      Y=
        ORG     $3000
Data    FCB     1,2
```

Slut Uppgift 27

Direct:

Detta är ett kortare sätt att adressera minnet än *Extended*. Endast 1 byte åtgår för att koda operanden. För de 8 mest signifikanta bitarna i adressen används alltid 00, dvs endast adressintervallet 0-FF_{16} kan nås med detta adresseringssätt.

Tecknet '<' framför adressen betyder *Force Direct Page*, dvs. instruerar assemblatorna att använda detta adresseringssätt.

På motsvarande sätt kan man använda tecknet '>' framför adressen för att instruera assemblatorn att använda *Extended* adressering.

Uppgift 28

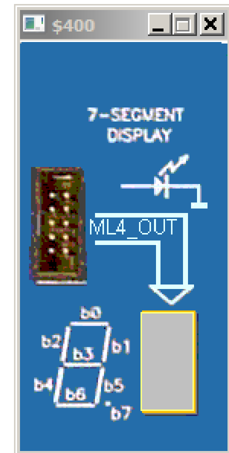
Undersök *Direct* och *Extended* adresseringssätt. Använd följande sekvens:

```
    ORG     0
sym0:
    ORG     $4000
sym4000:
    ORG     $1000
    LDAA    <sym0
    LDAA    <sym4000
    LDAA    >sym0
    LDAA    >sym4000
```

Assemblatorn kommer att generera ett felmeddelande. Undersök listfilen.

Slut Uppgift 28

Sju-sifferindikator



Sju-sifferindikatorn är en vanlig presentationsenhet i digitala system. Med en sådan kan man på ett enkelt och billigt sätt presentera tecken som kan hänföras till de välbekanta siffrorna 0-9. Namnet 'sju-segment' kommer av att det faktiskt går att representera dessa siffror, om än något kantigt, med endast sju olika streck, kallas också *segment*. Man införde också ett åttonde 'segment' vars uppgift är att tända en decimalpunkt.

Under detta moment ska ett assemblerprogram konstrueras som utför översättning och utmatning av de binära siffrorna 0 t.o.m 9 till motsvarande representationer på sju-sifferindikatorn.

Funktionsbeskrivning

Sju-sifferindikatorn på ML4 7-segment display fungerar enligt följande:

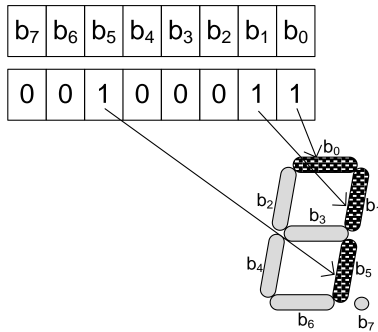
- Varje bit, i det dataord (8 bitar) som skrivs till utporten, motsvarar ett segment på sju-sifferindikatorn.
- En *etta* tänder ett segment, en *nolla* släcker segmentet.

Decimala siffror 0-9 representeras med fyra binära siffror enligt följande tabell:

Decimal-siffra	Binärkod	Decimal-siffra	Binärkod
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Översättningen från en binär siffra (0-9) till motsvarande sju-segmentskod beror naturligtvis helt och hållet på vilken typ av sju-sifferindikator man använder. I detta fall ska du först studera följande exempel på översättningen av den decimala siffran '7' till dess motsvarande sju-sifferkod. Observera speciellt att det alltid åtgår 8 bitar för en sju-sifferrepresentation, medan siffrorna 0-9 (binärt) kan representeras med 4 bitar.

För att representera siffran 7 måste vi tända de segment som (tillsammans) ger det mest ”7-lika” utseendet, i detta fallet segmenten 0, 1 och 5.



En etta tänder ett segment, en nolla släcker segmentet.

I exemplet använder vi 23_{16} detta bitmönster formar siffran sju på sifferindikatorn

Uppgift 29

Följande tabell illustrerar förhållandet mellan binära koder och sjusegmentskod. Studera speciellt föregående exempel och komplettera i tabellen med med de saknade sjusegmentskoderna.

Decimal siffra		**Sjusegmentskod**	
	Binär kod	Binär kod	Hex kod
0	0000	0111 0111	77
1	0001		
2	0010		
3	0011		
4	0100		
5	0101		
6	0110		
7	0111	0010 0011	23
8	1000		
9	1001		

Slut Uppgift 29

Ytterligare adresseringssätt

Indexerad:

I de indexerade adresseringssätten används något av registren X,Y eller SP. Innehållet i detta s.k. indexregister, används som basadress och bildar, tillsammans med någon förskjutning (*offset*), adressen till operanden.

Förskjutningen kan vara en konstant och tolkas då av assemblatorn som ett tal med tecken, dvs. såväl negativa som positiva konstanter kan användas, (*constant offset*), dvs. i intervallet:

$$-32768 \leq offset \leq 32767$$

Förskjutningen kan också utgöras av innehållet i något av ackumulatorerna A eller B respektive D. (*register offset*).

Uppgift 30

1. Skapa en tabell med start på adress 3000_{16}, som innehåller segmentskoder för siffrorna 0..9 i tur och ordning (enligt Uppgift 29).
2. Skapa programtexten, enligt följande, med start på adress 1000_{16}.
3. Använd simulatorn för att undersöka instruktionssekvensen. Fyll i tabellen.

Instruktion	ackumulator A
`LDX #$3000`	-
`LDAA 0,X`	
`LDAA 1,X`	
`LDAA 2,X`	
`LDAA 3,X`	
`LDAA 4,X`	
`LDAA 5,X`	
`LDAA 6,X`	
`LDAA 7,X`	
`LDAA 8,X`	
`LDAA 9,X`	

4. Kontrollera att kolumnen med värden i A överensstämmer med sjusegmentskoderna.

Slut Uppgift 30

Det är inte alltid lämpligt att använda konstant förskjutning. I vissa sammanhang tvingas man beräkna den eller också ges den helt enkelt som indata som i nästa uppgift.

I ett flödesdiagram kan vi symboliskt skriva:

$$A \leftarrow M(X+B)$$

Dvs.

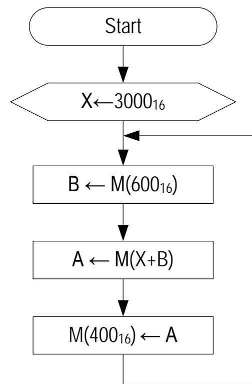
- Bestäm adressen genom att addera X och B.
- Placera innehållet på denna adress i A.

Motsvarande operation utförs av assemblerinstruktionen:

```
LDAA  B,X
```

Uppgift 31

Följande flödesschema visar ett program där vi läser ett värde från adress 600_{16}, använder detta värde för att indexera i en tabell med start på adress 3000_{16} och slutligen skriver ut det indexerade tabellvärdet till sifferindikatorn.



1. Skapa en ny källtext, `Indexed.s12`, med en tabell, med start på adress 3000_{16} som innehåller segmentskoder för siffrorna 0..9 i tur och ordning, dvs. samma tabell som i Uppgift 30.
2. Skapa programtexten, enligt ovanstående flödesschema, med start på adress 1000_{16}.
3. Koppla *ML4 Dip Switch Input* till adress 600_{16} respektive *ML4 7-segment display* till adress 400_{16}.
4. Använd simulatorn och övertyga dig om att programmet fungerar som det ska, dvs. ställ in värdena 0 t.o.m 9 (0000_2 - 1001_2) på omkopplaren och läs av sifferindikatorn.

Om du gjort allting rätt ska programmet kunna visa siffrorna 0-9 på sifferindikatorn. Om inte, felsök och rätta i programmet och tabellen.

5. Prova nu med att ställa in värden som är större än 9 på omkopplaren.

Varför är det meningslöst att använda större värden än 9?

Slut Uppgift 31

Det finns två olika sätt att lösa det uppkomna problemet.

a) Komplettera tabellen med någon speciell segmentkod för "fel", exempelvis 'E' för alla otillåtna värden hos indata.

b) Gör en kontroll (jämförelse) av indata och skriv direkt ut felkoden om det är ett otillåtet värde.

Tidigare har vi utfört tester och styrt programflödet om operanden varit noll (skild från noll). Det nya i denna programkonstruktion är 'if'-satsen där en storleksrelation används, det räcker då inte med test av operanden utan vi gör i stället en jämförelse.

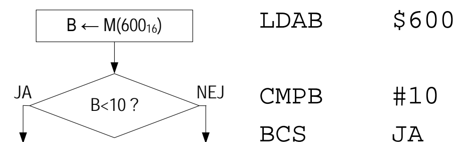
Om vi tolkar indata som ett tal utan tecken (0-255) har vi följande jämförelseoperationer att välja bland:

Jämförelse av tal utan tecken		
`BHI`	Villkor: R>M	C + Z = 0
`BCC`	Villkor: R≥M	C=0
`BCS`	Villkor: R<M	C=1
`BLS`	Villkor: R≤M	C + Z = 1

I tabellen står R för ett register och M för ett minnesinnehåll eller en konstant.

Observera speciellt att en bestämd operation, dvs. jämförelse/villkorlig flödesändring, här kan göras med vilken som av branchinstruktionerna.

Ett exempel på flödesplan och kodning är följande:

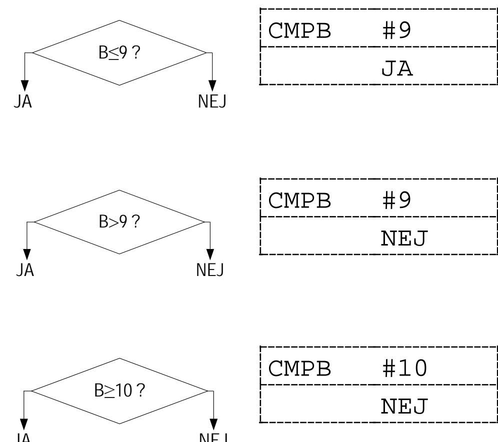



```
LDAB    $600

CMPB    #10
BCS     JA
```

dvs. "JA"-grenen utförs om indata är i intervallet 0..9.

Uppgift 32

Utgå från föregående exempel, komplettera kodningen med rätt branchinstruktion.



Slut Uppgift 32

Uppgift 33

Fortsätt i denna uppgift på programsekvensen i filen `Indexed.s12`.

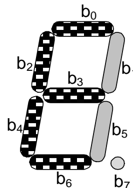
Den här gången ska programmet:

- Läsa ett värde från inporten
- Kontrollera detta värde, om det kan motsvaras av de decimala siffrorna 0..9 ska dess sjusegmentskod skrivas till sjusifferindikatorn.
- Om det är ett ogiltigt värde ska segmentkoden för bokstaven 'E' skrivas till sjusifferindikatorn.

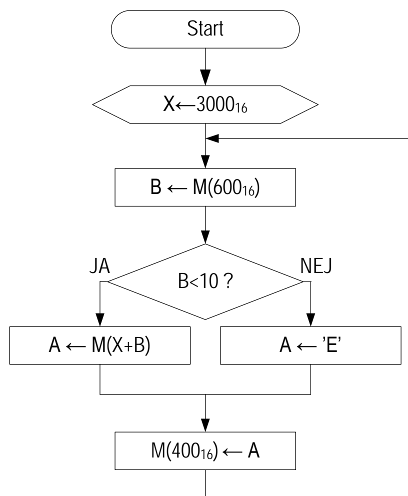
Felkoden 'E' kan visas på sjusifferindikatorn enligt figuren. Definiera den som en konstant enligt:

```
ERROR_CODE    EQU  ?
```

? = ______



Programsekvensens funktion kan beskrivas med följande flödesplan:



Konstruera programmet, testa och verifiera att det fungera korrekt.

Slut Uppgift 33

Subrutiner

Det är vanligt att man försöker organisera ett program i olika *funktioner* (*procedurer*) eller som vi vanligtvis, då det gäller assemblerspråk, kallar det *subrutiner.*

En subrutin karakteriseras av att den har ett inträde (entry) och ett utträde (exit). Inträdet anges oftast genom att man placerar en symbol i symbolfältet. Utträdet, dvs "åter från subrutin" specificeras av en speciell maskininstruktion. Följande subrutin, som vi gett det symboliska namnet "sub", uppfyller detta men utför i övrigt ingenting:

```
            ORG     $1000
sub:
            RTS
```

Ett *subrutinanrop* kan exempelvis utföras enligt

```
        ...
        JSR     sub
        ...
```

Operanden för `JSR` är en subrutins inträde. Denna kan anges med flera olika adresseringssätt men det vanligaste är en symbolisk adress.

JSR-instruktionen utför två väsentliga operationer:

- adressen till nästa instruktion sparas
- programräknaren initieras med adressen till subrutinen

RTS-instruktionen utför operationen

- adressen till nästa instruktion återställs

För att kunna "spara"/"återställa" adressen till nästa instruktion används minnesutrymme som pekas ut av den så kallade "stackpekaren" (stack: ung "hög"). Stackpekaren, i vårt fall register `SP`, måste alltså ha initierats med en lämplig adress innan `JSR` används. Observera att simulatorn ger register `SP` initialvärdet $3C80_{16}$. För att initiera stackpekaren på samma sätt i ett program används instruktionen:

```
LDS  #$3C80
```

Då `JSR` "sparar" en adress kommer denna att placeras på stacken, och instruktionen använder (implicit) processorns register `SP`. På motsvarande sätt kommer `RTS` att återställa en adress, genom att hämta denna från stacken. Formellt kan detta beskrivas som:

JSR:

- Minska stackpekaren med 2
- Placera återhoppsadress på stacken
- Placera adressen till subrutinen i `PC` (programräknaren).

RTS:

- Placera adressen som för närvarande ligger överst på stacken i `PC` .
- Öka stackpekaren med 2.

Simulatorns stackhantering

Simulatorns stacksektion används för att visa stackpekarens värde. Samtidigt ger den en överblick av innehållet på stacken. Den vita pilen indikerar såväl stackpekarens innehåll kolumn `SP` som innehållet på den adress som pekas ut, kolumn `(SP)`. Ytterligare några närliggande adresser visas. Varje gång stackpekaren ändras av någon instruktions-exekvering ändras också pilens läge och färg.

```
Stack
   SP  (SP)
  3C7A  00
  3C7B  00
  3C7C  00
  3C7D  00
  3C7E  00
  3C7F  00
→ 3C80  00
  3C81  00
```

Uppgift 34

Använd följande program för att studera stackhanteringen vid subrutinanrop.

Observera speciellt stackpekarens värde samt innehållet på toppen av stacken (SP:SP+1) i subrutinerna, komplettera tabellen. Värdena anges efter det att radens instruktion utförts.

			SP	(SP:SP+1)
	ORG	$1000	XXXX	XX XX
start:	LDS	#$3C80	3C80	XX XX
	JSR	sub1	3C7E	
	JSR	sub2		
stop:	BRA	stop		
sub1:	LDAB	#1		
	RTS			
sub2:	LDAB	#2		
	RTS			

Slut Uppgift 34

Stacken kan också användas för att tillfälligt spara registerinnehåll. Detta är många gånger användbart eftersom registren är få och samtidigt behövs för många olika saker. Instruktionerna PSH (*PuSH register*) respektive PUL (*PULl register*) används för detta ändamål.

Uppgift 35

Använd följande programsekvens för att studera stackhantering med PSH och PUL. Observera speciellt stackpekarens värde samt innehållet på toppen av stacken. Notera också hur värdet i ackumulator A sparas och återställs.

		A	SP	(SP)
ORG	$1000	XX	3C80	XX
LDAA	#$AA			
PSHA				
LDAA	#$55			
PULA				

Slut Uppgift 35

Tidigare, i Uppgift 24 skapade du en programsekvens för fördröjning av programexekveringen. Du ska nu bygga vidare på denna och skapa en subrutin Delay, med variabel fördröjning. Vi utgår hela tiden från att programmet utförs i simulatorn, med RunFast-funktionen.

Programsekvensen som utför själva fördröjningen, kallar vi fortsättningsvis för ”fördröjningssekvens”.

Vi börjar med att utforma ett ”subrutinhuvud”, dvs. text, i kommentarsform, som kortfattat beskriver subrutinen och hur den är tänkt att användas.

```
;-----------------------
; SUBRUTIN Delay
; åstadkommer fördröjning av program.
; Fördröjningen utförs i steg om 0,25
; sekunders intervall.
; Indata:
;   Register B: Antal fördröjningar om
;   0,25 s vardera.
; Registerpåverkan:
;   Register B innehåller alltid 0 efter
;   subrutinen. Inga andra register
;   påverkas.
```

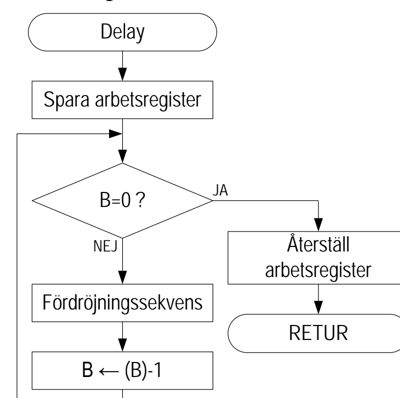
Om en fördröjning om 2,5 sekunder ska utföras blir anropet:

```
    LDAB  #10
    JSR   Delay
    ...
```

OBSERVERA

Med frasen ”Inga andra register påverkas”, menas att arbetsregistren (alla utom CCR) har samma värden efter subrutinen som före anropet. Om sådant används i subrutinen måste dess innehåll därför sparas, vanligtvis omedelbart efter inträdet (subrutinens första instruktion) för att återställas, vanligtvis omedelbart före utträdet.

Om fördröjningssekvensen utformas för att ge c:a 250 ms fördröjning beskrivs subrutinen av följande flödesplan.



Uppgift 36

Implementera nu subrutinen Delay i källtextfilen `Delay.s12` enligt anvisningarna ovan.

Skapa också ett enkelt huvudprogram som anropar subrutinen enligt följande exempel:

```
        ORG     $1000
start:  LDAB    $600
        JSR     Delay
        BRA     start

Delay:  ...
        ...
        RTS
```

Då man använder symboliska namn för att ange lägen i själva subrutinen bör man undvika generella namn som ”next”, ”loop” etc. eftersom det ganska snart skapar namnkonflikter allt eftersom fler subrutiner läggs till.

Man kan i stället döpa varje läge med subrutinens namn och lägga till någon text för att skapa ett unikt lägesnamn.

I subrutinen Delay kan man exempelvis använda lägesnamnen Delay_1, Delay_2, Delay_3 osv.

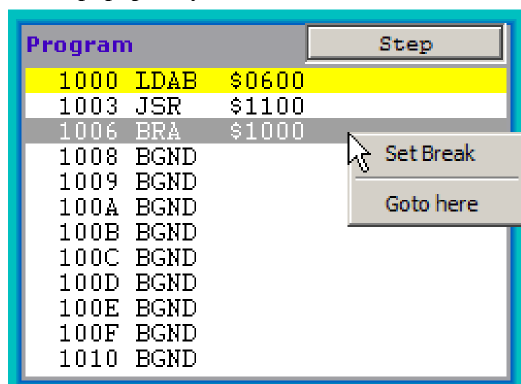
Slut Uppgift 36

För att testa fördröjningsrutinen ska vi först beskriva hur så kallade ”brytpunkter” används.

Brytpunkter

Ladda ditt program Delay till simulatorn, programsektionen uppdateras. Vi har här placerat subrutinen på adress 1100_{16} i minnet för att den inte ska vara synlig i programfönstret, det är alltså bara huvudprogrammet som syns men detta har ingen praktisk betydelse.

Om du *högerklickar* exempelvis då markören är över raden som inleds med `1006 BRA`.... får du denna popupmeny:

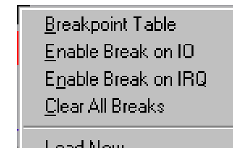


En brytpunkt är ett ställe där programexekveringen avbryts. Brytpunkten kan vara permanent (Set Break) och läggs då in i en brytpunktstabell som beskrivs nedan. Varje gång programmet ska utföra instruktionen på denna adress kommer simulatorn att avbryta exekveringen.

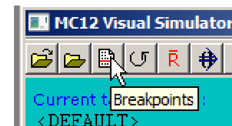
Brytpunkten kan också vara tillfällig (Goto here) programmet exekveras tills denna adress uppträder <u>nästa gång</u>. Brytpunkten tas därefter automatiskt bort av simulatorn.

Brytpunkter är bara meningsfyllda då du använder Run eller Run Fast.

Hantering av speciella brytpunktsfunktioner kan göras från simulatorns popupmeny.



- Breakpoint Table, beskrivs nedan, kan också startas från verktygsfältet.



- Enable Break on IO, brytpunkt sätts automatiskt vid skrivning/läsning till/från någon adress som reserverats för in- eller utmatning. Brytpunkterna markeras inte i programfönstret men en diod i statusfönstret indikerar att denna funktion aktiverats. Simulatorn avbryter programutförandet vid varje instruktion som utför IO.
- Enable Break on IRQ, brytpunkt sätts automatiskt vid avbrott, vi återkommer till avbrott i senare avsnitt. Simulatorn avbryter programutförandet vid någon form av avbrott.
- Clear All Breaks, återställ alla inställningar för brytpunktshantering.

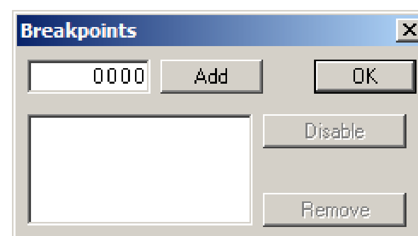
Brytpunktstabellen

I brytpunkstabellen kan du lägga in upp till 20 samtidiga brytpunkter i ditt program. För att bestämma vilka adresser som kan vara lämpliga för brytpunkter kan du granska listfilen från ditt program.

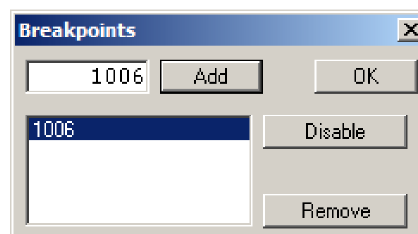
Följande uppgift illustrerar den enkla hanteringen av brytpunktstabellen.

Uppgift 37

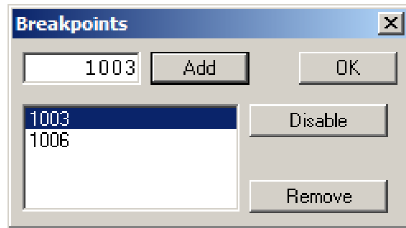
Välj Breakpoint Table på något sätt, följande dialogruta visas:



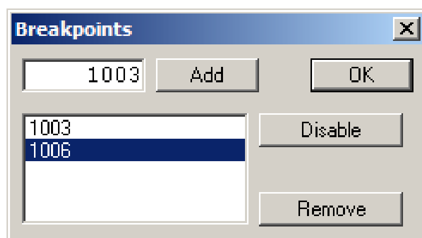
1. Skriv adressen 1006 och klicka på Add för att lägga in en ny brytpunkt i tabellen.



2. Lägg på samma sätt in ytterligare en brytpunkt, på adressen 1003.



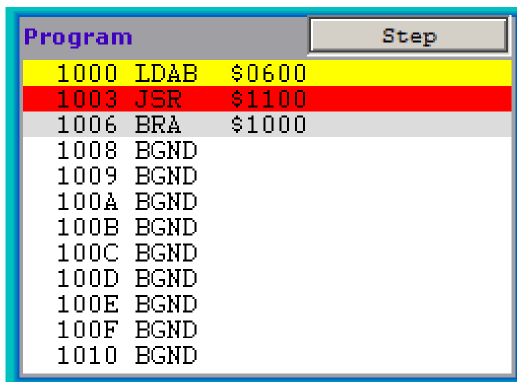
3. Välj nu brytpunkten på adress 1006 genom att att klicka på adressen.



En brytpunkt kan inaktiveras med Disable, vilket betyder att simulatorn ignorerar den, brytpunkten finns dock kvar i tabellen och kan på nytt aktiveras med Enable.

4. Inaktivera brytpunkten på adress 1006.

Klicka nu på OK för att stänga dialogrutan för brytpunkter, notera därefter hur programfönstret påverkats av brytpunkterna:



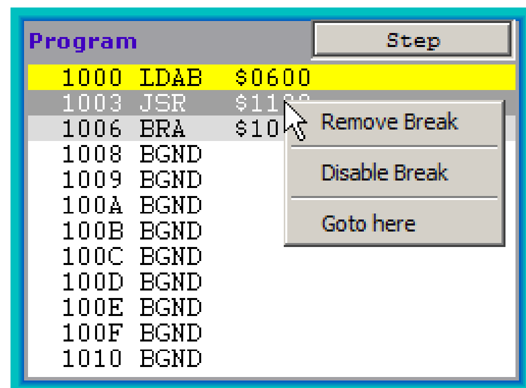
- En aktiv brytpunkt illustreras av röd bakgrund.
- Den gula bakgrunden anger vilken instruktion som kommer att utföras näst.
- En inaktiv brytpunkt illustreras av ljusgrå bakgrund.

Om den instruktion som står i tur att utföras har en brytpunkt visas denna med gul text mot svart bakgrund.

Brytpunkter synliga i programfönstret är enkla att hantera.

5. Högerklicka på raden med den aktiva brytpunkten (adress 1003).

En popupmeny visar de tillgängliga alternativen.



Om du vill ta bort en brytpunkt som inte är synlig i programfönstret måste du öppna dialogrutan Breakpoint Table på nytt. Märk den brytpunkt du vill ta bort och klicka på Remove.

Slutligen kan du ta bort alla brytpunkter på en gång genom att välja alternativet Clear All Breaks från simulatorns popupmeny

Slut Uppgift 37

Uppgift 38

Det är nu dags att testa funktionen hos fördröjnings-rutinen.

1. Låt endast brytpunkten på adress 1006_{16} vara kvar.
2. Koppla ML4 Dip Switch Input till adress 600_{16}
3. Ställ in värdet 0 på omkopplaren och välj RunFast, kontrollera att programmet stannat på adress 1006_{16} dvs. brytpunkten.
4. Ställ nu in värdet 1 på omkopplaren och kontrollera på samma sätt att subrutinen genomlöpes och programutförandet stannar vid brytpunkten igen.

Slutligen ska du ”trimma” fördröjningsrutinen så att den blir så exakt som möjligt. Det är enklast att göra detta genom att ställa in en relativt lång fördröjning, utföra programmet till brytpunkten och samtidigt mäta (med en klocka) hur lång tid det tar och därefter modifiera begynnelsevärdet i fördröjningsslingan.

5. Ställ in värdet 20 ($=14_{16} = 00010110_2$), dvs. 10 sekunders fördröjning, på omkopplaren och justera begynnelsevärdet i fördröjningsrutinen tills du känner dig nöjd med noggrannheten.

Slut Uppgift 38

Avsnitt 2

Tangentbord och Display

Syften:

I detta avsnitt ställs du inför uppgiften att skapa en sammanhållen tillämpning, ett så kallat "stoppur".

Vi introducerar två nya typer av kringenheter, ett tangentbord med 16 olika tangenter och en visningsramp för samtidig visning av upp till sex olika siffror.

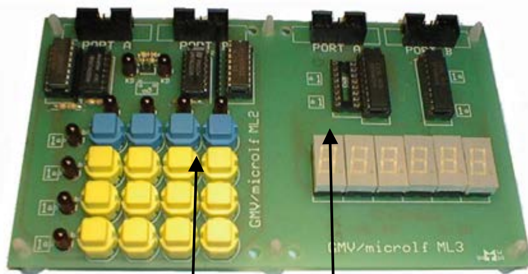
Ett viktigt syfte med avsnittet är att ge exempel på hur programsystem kan, och bör delas in i överskådliga delar (modularisering). Avsnittet beskriver också en metod att dela upp källtexten i olika källtextfiler och hur du kan utnyttja villkorlig assemblering för att göra dina program generella och därmed också återanvändbara.

Inledning

Vi introducerar nu två nya gränssnitt *ML15* och *ML5*. Båda gränssnitten är avsedda att anslutas till två olika kringenheter; en enkel inmatningsenhet i form av ett tangentbord med 16 tangenter, och en enkel utmatningsenhet i form av en sifferindikator för högst 6-siffror.

Kringenheterna vi använder kallas

- Tangentbord *ML2*
- Displaymodul *ML3*.



ML2, Ett enkelt tangentbord

ML3, En enkel displaymodul

Till *ML2* och *ML3* finns det två olika gränssnitt, *ML15* och *ML5*.

ML15 innehåller färdig styrlogik för *ML2* och *ML3* vilket innebär att programlösningarna blir enkla.

ML5 däremot innehåller enbart enkla in- och ut-portar vilket medför att det krävs avancerade program.

ML2/ML15 Funktion

I *ML15* sker all avkodning av tangentbordet i hårdvara. Resultatet av avkodningen kan alltid avläsas från ett register på den anslutna adressen. Adressen ($9C0_{16}$) avser tangentbordet *ML2* med gränssnittet *ML15*. Följande figur beskriver registret och bitarnas betydelse.

ML15, adress 09C0

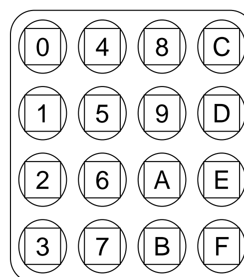
7	6	5	4	3	2	1	0
DAV	0	0	0	b_3	b_2	b_1	b_0

Bit 7, DAV: Data Valid; Statusbit som anger nedtryckt tangent
1 = Ingen tangent är för tillfället aktiverad på tangentbordet.
0 = En tangent är aktiverad

Bit 6-4, 0: Används ej

Bit 3-0, b_3-b_0: Tangentnummer;
Anger aktuell (eller senast) nedtryckta tangent som en hexadecimal siffra (0-F).

ML15's avkodning av tangentbordet, dvs. den kod som placeras i registret på adress 9C0, beroende på vilken tangent som tryckts ned, framgår av följande figur:



ML2 finns som kringenhet i IO-simulatorn.

Det är nu dags att undersöka hur tangentbordet *ML2* fungerar tillsammans med gränssnittet *ML15*.

Uppgift 39

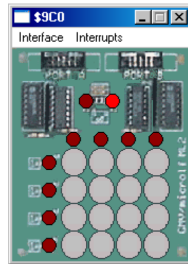
Skapa en ny fil `KeyboardML15.s12` med en enkel instruktionssekvens, enligt följande, för att undersöka tangentbordet:

```
        ORG         $1000
start:
        LDAB        $9C0
        BRA         start
```

Assemblera ditt testprogram, ladda till simulatorn och anslut IO-simulatorns enhet *ML2 Keyboard* till adress $9C0_{16}$.

Enheten har två menyalternativ, Interface och Interrupts. Kontrollera att:

- Interface = *ML15*
- Interrupts = *Disable*



Klickar du på en tangent på *ML2* märks denna grå vilket innebär en nedtryckt tangent. Klickar du en gång till på tangenten försvinner markeringen och tangenten är då inte längre nedtryckt, det är detsamma som att du släpper fingret från den nedtryckta tangenten.

Stega dig igenom programsekvensen och prova olika tangentnedtryckningar. Kontrollera speciellt följande alternativ och komplettera tabellen:

Nedtryckt tangent	*Avläst värde (ackumulator B)*
ingen tangent	
0	
7	
E	

Observera vad du läser från tangentbordet när ingen tangent är nedtryckt. Har *föregående* tangentnedtryckning någon betydelse för avläsningen? Skriv ditt svar här:

Slut Uppgift 39

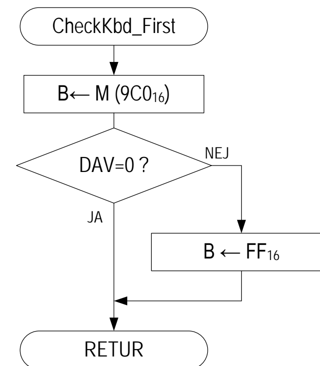
Uppgift 40

Fortsätt arbeta med källtextfilen `KeyboardML15.s12`, ersätt den enkla instruktionssekvensen med en subrutin och ett testprogram.

Konstruera en subrutin, CheckKbd_First, enligt följande specifikation:

```
; Subrutin CheckKbd_First
; Läser tangentbord via ML15
; Returvärde i B:
;  B = 0..F (hexadecimalt), tangentkod
;  B = FF om ingen tangent är nedtryckt
; Inga andra register påverkas
```

Utforma subrutinen enligt följande flödesplan. Observera speciellt vilken bit som testas och bestäm den lämpligaste villkorliga BRANCH-instruktionen.



Kontrollera att din lösning fungerar som den ska genom att använda följande testprocedur:

```
; Testprogram för CheckKbd_First
            ORG     $1000
start:      JSR     CheckKbd_First
            CMPB    #$FF
            BEQ     start
            NOP
            BRA     start
; Här följer din subrutin…
```

Sätt en brytpunkt vid instruktionen `NOP`. Då programexekveringen testas ska det stoppas, vid din brytpunkt, enbart om en tangent tryckts ned.

Kontrollera följande sekvens tangentnedtryckningar och fyll i tabellen. Tangentavkodningen är den samma som i föregående uppgift.

Nedtryckt tangent	*Avläst värde (ackumulator B)*
1	
Ingen tangent	
7	
A	
Ingen tangent	
C	

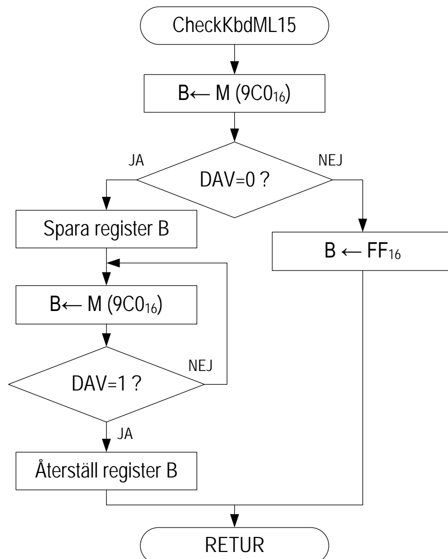
Slut Uppgift 40

I vissa sammanhang kan det vara viktigt att hålla reda på *hur många gånger* en tangent trycks ned. En nackdel med tangentbordsrutinen i föregående uppgift är då att samma tangentnedtryckning kan detekteras flera gånger vid flera anrop av tangentbordsrutinen. För att komma tillrätta med det kan tangentbordsrutinen utformas så att den efter en detekterad nedtryckning väntar på att tangenten släpps upp igen.

Uppgift 41

Modifiera subrutinen CheckKbd_First i källtextfilen `KeyboardML15.s12` från föregående uppgift.

Betrakta flödesplanen nedan. Observera att då DAV=0, innehåller ackumulator B koden för den nedtryckta tangenten. För att kunna använda samma register för att bestämma då tangenten släppts upp igen sparar vi därför tangentkoden på stacken för att återställa den i B efter vänteslingan för uppsläppt tangent. Konstruera nu en tangentbordsrutin CheckKbdML15 enligt flödesplanen.



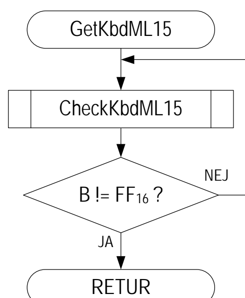
Kontrollera att din lösning fungerar som den ska genom att använda följande testprocedur:

```
; Testprogram för CheckKbdML15
            ORG     $1000
start:      JSR     CheckKbdML15
            CMPB    #$FF
            BEQ     start
            NOP
            BRA     start
;
; Här följer din subrutin…
CheckKbdML15 ...
```

Slut Uppgift 41

Uppgift 42

Skapa nu också en subrutin GetKbdML15, som alltid returnerar en giltig tangentkod, dvs. väntar på tangentnedtryckning, avvaktar därefter att tangenten släppts upp. Subrutinen ska använda CheckKbdML15 och utformas enligt följande flödesplan:



I källtextfilen `KeyboardML15.s12` ska du nu ha de två subrutinerna CheckKbdML15 och GetKbdML15- De ska vara testade och fungera efter anvisningarna.

Slut Uppgift 42

ML3/ML15 Funktion

Vi ska nu visa hur display-modulen *ML3* används tillsammans med gränssnittet *ML15*.

ML15 använder tre register för att styra display-modulen *ML3*, gränssnittet upptar dock bara två konsekutiva adresser i minnet.

ML15, adress 09C2

Display mode register

Registret används för att välja arbetssätt (”mod”) för styrkretsen. Endast bit 0 används.

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	M

Bit 7-1, används ej, alltid 0

Bit 0 M, anger arbetssätt
1 = Kretsen ställs i *styrmod.* (Styrord kan ges till styr/data- registret)
0 = Kretsen ställs i datamod. (Data kan ges till styr/data- registret)

ML15, adress 09C3

Display styr/data register

Registret har dubbelfunktion som bestäms av bit 0 i moderegistret.

Styr register, registret är endast åtkomligt när mode registrets bit 0 är ettställd.

7	6	5	4	3	2	1	0
DC	CODE	DEC	SH	0	0	0	0

Bit 7 DC; Data Coming; Anger att data ges nästa gång kretsen sätts i datamode.
1 = Data kommer.
0 = Ingen data

Bit 6 CODE; Anger kodningtyp
1 = Hexadecimal kod väljs.
0 = CODEB väljs.

Bit 5 DEC; Decoding; Anger om kodning skall användas.
1 = Ingen kodning.
0 = Hexadecimal eller CODEB (enl. CODE)

Bit 4 SH; Shut Down; Aktivera eller släck displayen.
1 = Normal mod.
0 = Släck Display

Bit 3-0, 0, används ej.

Anm. För förklaring till kodningstyp (CODE,DEC) se den tekniska beskrivningen av ML15.

Data register, registret är endast åtkomligt när ”mode”- registrets bit 0 är nollställd.

7	6	5	4	3	2	1	0
D	0	0	0	B3	B2	B1	B0

Bit 7 D; Decimal punkt för aktuellt tecken.
1 = Sätt decimalpunkt.
0 = Ingen decimalpunkt

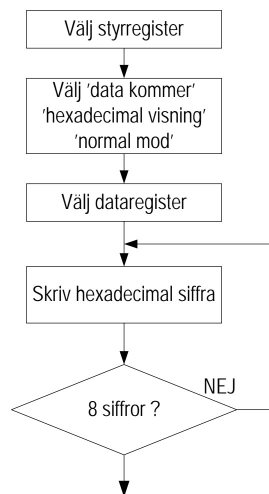
Bit 6-4 0; används ej

Bit 3-0 B3-B0; Data. Data ges på CODE B eller Hexadecimal form

För att tända sifferindikatorerna krävs:

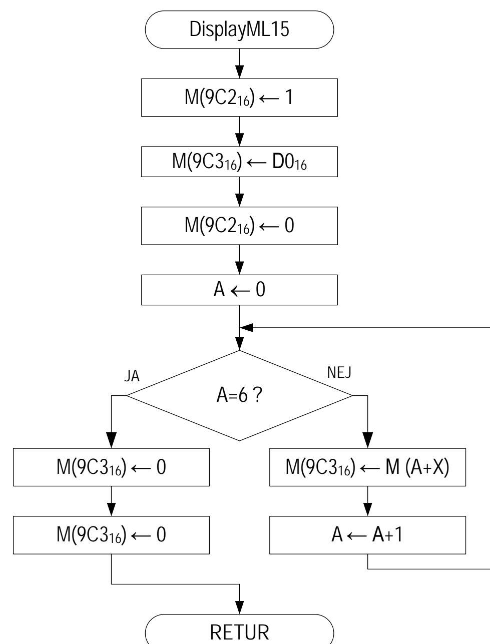
- Att visningskretsen på *ML15* initieras, dvs ett lämpligt styrord skrivs till styrregistret.
- Att 8 bytes data skrivs till dataregistret.

Därefter tänds sifferindikatorn. Notera att endast 6 siffror kan visas av *ML3*, men att styrkretsen på *ML15* är förberedd för 8 sifferindikatorer och *kräver* därför att 8 bytes data skrivs till dataregistret. Det är de första sex tecknen som skrivs till sifferindikatorn, de två sista ignoreras. Förfarandet kan beskrivas enligt följande:



Uppgift 43

Skriv en subrutin DisplayML15 som visar siffrorna `1 2 3 4 5 6` på sifferindikatorerna *ML3* via gränssnittet *ML15*. Subrutinen beskrivs detaljerat av följande flödesdiagram:



Låt siffrorna representeras av en tabell i minnet och använd register X som pekare till tabellens start (siffran 1) och låt ackumulator A hålla en räknar-variabel för att indexera i tabellen.

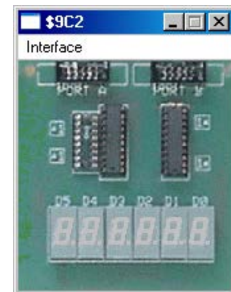
Utforma subrutinen så att register X förutsätts innehålla pekaren vid anrop. Ett huvudprogram som använder DisplayML15 kan då se ut på följande sätt:

```
; Testprogram för DisplayML15
          ORG     $1000
          LDX     #tabell
start:    JSR     DisplayML15
          BRA     start
;
tabell:   FCB     1,2,3,4,5,6
```

Subrutinens funktion kan nu sammanfattas enligt följande:

```
; Subrutin DisplayML15
; Initierar, skriver 6 tecken till
; och tänder ML15 display
;
; Indata:   Startadress till
;           896ffersträngen i register X
; Register: A ändras av subrutinanropet
```

Anslut IO-simulatorns enhet ML3 Display till adress $9C2_{16}$. Enheten har ett menyalternativ, Interface



Kontrollera att:

- Interface = *ML15*

Stega nu igenom programmet till att subrutinen genomlöpts en gång. Kontrollera att indikatorerna visar rätt siffror.

Starta nu om programmet med Run, visar indika-torerna korrekt? Beskriv kortfattat vad som händer.

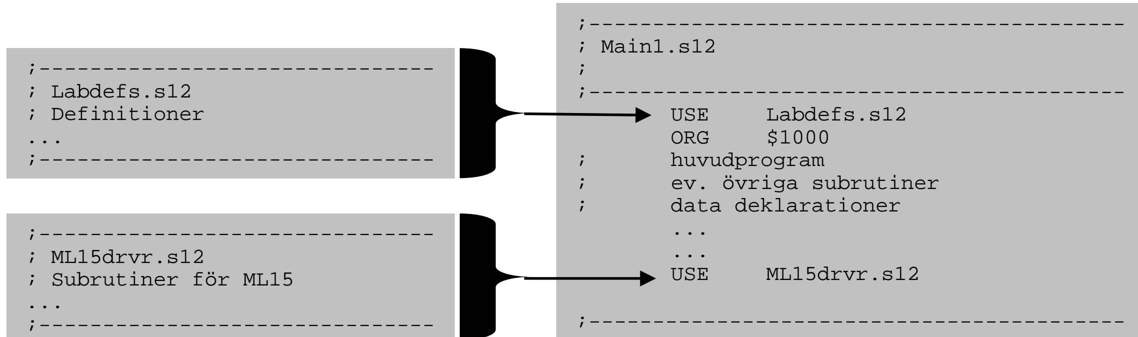
Slut Uppgift 43

Användning av USE-direktivet

Allt eftersom programmet växer blir behovet av att modularisera och skapa välordnade strukturer allt större. En typisk applikation kan som använder ML15 kan nu delas upp i tre olika källtextfiler:

- `Labdefs.s12`, här samlar vi alla konstantdefinitioner (EQU-satser). Observera att varken programkod eller datadefinitioner ska finnas i denna fil.
- `Main.s12`, här skriver vi huvud-programmet och kanske också ett fåtal subrutiner om sådana krävs, här deklareras också globala variabler (data definitioner).
- `ML15drvr.s12`, här samlas de delar av programmet som är direkt kopplade till gränssnittet *ML15* (”drivrutiner”).

Figuren överst på nästa sida visar hur man med användning av USE-direktivet låter assemblatorn sätta samman alla källtextfiler till en enda fil som sedan assembleras.



Fördelarna med en sådan här uppdelning är flera. Det blir nu enkelt att återanvända konstantdefinitioner och drivrutiner i flera olika applikationers huvudprogram.

När du använder USE-direktivet på detta sätt är det några saker du ska beakta speciellt:

- Använd ORG-direktiv *bara* i MAIN. På detta sätt kan du vara säker på att inga programdelar överlappas i minnet.
- Endast definitioner (EQU-satser) får föregå det första ORG-direktivet.
- Om du deklarerar data (direktiven `RMB`, `FCB`, `FCC`, etc.) så placera dessa sist i filen `Main1.s12`.

Tänk också på att det är filen `Main1.s12` som ska assembleras. Du kan visserligen också assemblera de andra filerna men det kommer bara att resultera i felutskrifter och den resulterande laddfilen är förstås meningslös.

Uppgift 44

Skapa en ny källtextfil `Main1.s12` med ett huvudprogram med samma funktion som "Testprogram för DisplayML15" och med USE-direktiv enligt anvisningarna ovan. Programmet ska ha ett inträde med symbolnamnet `main`.

Slut Uppgift 44

Drivrutinpaket ML15

I tidigare uppgifter har du skapat tre subrutiner för användning tillsammans med ML15. Du kommer att få användning av dessa längre fram och därför ska du nu ”snygga till” källtexterna enligt följande anvisningar.

Uppgift 45

Skapa en ny källtextfil `Labdefs.s12` med definitioner av alla portadresser för ML15 enligt:

```
; Labdefs.s12
ML15_KeyBoardRegister     EQU     $9C0
ML15_DisplayMode          EQU     $9C2
ML15_DisplayData          EQU     $9C3
```

Skapa ny källtextfil `ML15drvr.s12` med subrutinerna CheckKbdML15, GetKbdML15 och DisplayML15. Ersätt de *absoluta adresserna* du tidigare använt i dessa subrutiner med de symboliska adresserna i `Labdefs.s12`.

Slut Uppgift 45

Styrning av ML3 och ML2 via ML5

Vi ska i detta moment använda en annan, betydligt mindre komplex hårdvara, för att styra displayrampen och sedan också avkoda tangentbordet. Som vi kommer att se får vi "betala" för detta med ett mer komplicerat program.

Subrutinen Display för ML5

Programvarumässigt utgörs nu gränssnittet av två 8-bitars ut-portar ML5_Segments (kopplad till PORT A på ML3) respektive ML5_Position (kopplad till PORT B på ML3). Börja med att komplettera definitionsfilen med dessa adresser.

Uppgift 46

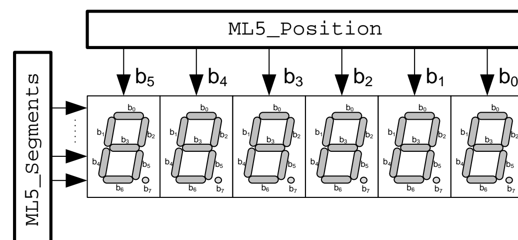
Komplettera filen `Labdefs.s12` med adress-definitioner (se även appendix A):

```
; Labdefs.s12
ML15_KeyBoardRegister     EQU     $9C0
ML15_DisplayMode          EQU     $9C2
ML15_DisplayData          EQU     $9C3
ML5_Segments              EQU     $C02
ML5_Position              EQU     $C03
```

Slut Uppgift 46

Att tända rätt sifferindikator

Det finns bara ett register för segmentmönster men det finns sex olika sjusifferindikatorer. Detta betyder att vi också måste ha något sätt att adressera, dvs märka upp vilken sifferindikator som ska tändas upp. För detta ändamål finns positionsregistret. Bit b_0 till och med b_5 används och logiknivån är aktiv låg, dvs en nolla tänder sifferpositionen för motsvarande bit medan en etta släcker sifferpositionen.



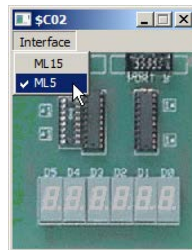
Register hos ML5

Uppgift 47

Skapa nu en enkel testsekvens för att undersöka sifferindikatorn ML3/ML5, och samtidigt kontrollera att portadresser är korrekta. Utforma sekvensen enligt följande:

```
; ML5test1.s12
    USE     Labdefs.s12
    ORG     $1000
start:
; mönster för att tända samtliga segment
    LDAA    #$FF
; mest signifikanta position b5
    LDAB    #%11011111

repetera:
; tänd segment
    STAB    ML5_Position
    STAA    ML5_Segments
; lägg ut segmentsmönster
    LDAA    #$FF
; släck alla indikatorer
    STAA    ML5_Position
; skifta nollan till nästa indikator
    ASRB
; tills samtliga positioner har visats
    BCS     repetera

; tänd alla indikatorer
    CLRA
    STAA    ML5_Position

    BRA     start
```

Anslut IO-simulatorns enhet ML3 Display till adress $C02_{16}$. Enheten har ett menyalternativ, Interface



Kontrollera att:

- Interface = ML5

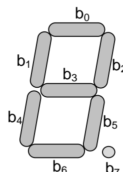
Stega igenom hela testsekvensen en gång och kontrollera funktionen.

Slut Uppgift 47

Att tända rätt segmentkod

Du har tidigare (i avsnitt 1) sett hur man enkelt översätter decimala siffror till motsvarande segmentskoder för användning med sifferindikatorn på ML4.

Sifferindikatorerna på ML3 fungerar på liknande sätt men segmentkoderna för b_1 och b_2 har här bytt plats.



Segmentkoder för ML5

Uppgift 48

Komplettera följande tabell med segmentkoder för ML5, jämför också din lösning med Uppgift 29.

Decimal siffra		Sjusegmentskod	
	Binär kod	Binär kod	Hex kod
0	0000	0111 0111	77
1	0001		
2	0010		
3	0011		
4	0100		
5	0101		
6	0110		
7	0111	0010 0101	25
8	1000		
9	1001		

Slut Uppgift 48

Att visa alla siffror "samtidigt"

Som du ser gäller det mönster du skriver till ML5_Segments för *alla* aktiva indikatorer men endast *en* indikator i taget skall lysa. För att åstadkomma detta krävs att en “nolla” roterar på ML5_Position, på samma sätt som i den enkla testsekvensen i Uppgift 47. Dessutom måste förstås respektive indikators segmentkod skrivas ut till ML5_Segments. På så sätt kommer de olika sjusegment indikatorerna att lysa en kort stund vardera med olika mönster. När den sista (den högra) indikatorn har lyst en kort stund avslutas visningen, alla segment och alla siffror ska då vara släckta.

Två saker gör dock att det inte är möjligt att få det att se ut som om alla sifferindikatorerna lyser på samma gång.

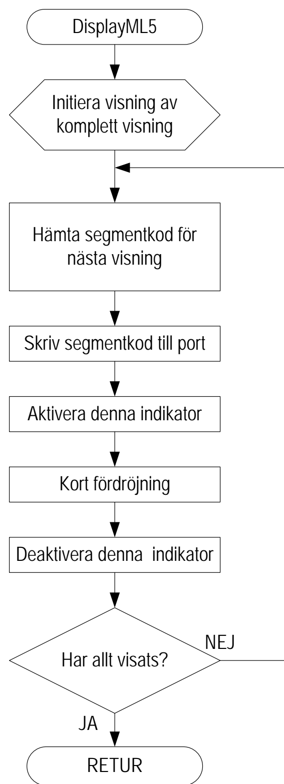
- För det första har simulatorns indikatorer ingen efterlysningstid och ...
- för det andra är simuleringen i båda fallen betydligt långsammare än det som krävs för att skapa denna vision.

Det är i själva verket mycket stora skillnader i exekveringshastigheter och följande tabell ger ungefärliga siffror som jämförelse:

EXEKVERINGS-HASTIGHET	simulator		MC12 / 8 MHz (typiskt 4 cykler/ instruktion)
	Run	RunFast	
instruktioner/ sekund	10	5 000	1 000 000

Algoritmen för att styra ML15 som beskrivs i det följande fungerar utmärkt i den verkliga MC12-datorn men i simulatorn kommer man tydligt att se hur siffrorna "sveper fram" vid Run, respektive "flimrar" vid RunFast.

Mot denna bakgrund specificerar vi nu subrutinen DisplayML15:



Villkorlig assemblering

Innan vi går vidare med fler programmerings-uppgifter ska vi presentera en praktisk funktion i assemblatorn, kallad *villkorlig assemblering*.

Villkorlig assemblering tillåter oss att använda samma källtexter för olika konfigurationer. Följande fördröjningsrutin baseras på Uppgift 24 i avsnitt 1.

```
ShortDelay:
    PSHB
    LDAB    #ShortDelayConst
ShortDelay_1:
    DBNE    B,ShortDelay_1
    PULB
    RTS
```

I vårt fall vill vi exempelvis kunna prova vårt program, med fördröjningsrutinen, både med simulatorns Run och RunFast funktion. Det som kommer att skilja konfigurationerna är det antal varv fördröjningsslingan ska genomlöpas vilket bestäms av ShortDelayConst, den konstant som används för fördröjningsslingan.

Det finns flera olika direktiv för att styra villkorlig assemblering. Samtliga direktiv inleds med tecknet '#'.

Genom att använda villkorlig assemblering kan vi med ett enda direktiv ange exempelvis en lämplig fördröjningskonstant för en given konfiguration:

```
#ifdef      RUN_FAST
; Använd stor fördröjningskonstant
ShortDelayConst    EQU     aa
#else
; Använd liten fördröjningskonstant
ShortDelayConst    EQU     bb
#endif
```

Vi har alltså *en* konfiguration då vi vill använda snabb simulering men *en annan* konfiguration då vi vill använda den långsamma simuleringen. Samma program kan nu användas i båda konfigurationer men kod som påverkas av tidsegenskaper kan utformas annorlunda med den villkorliga assembleringen.

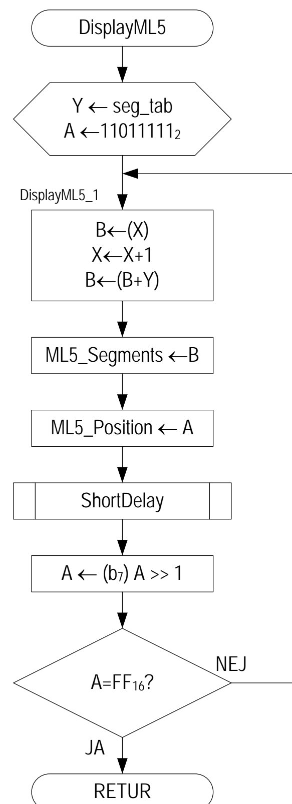
För att skapa en konfiguration som är avsedd att simuleras snabbt ("stor fördröjningskonstant"), ska följande direktiv föregå ovanstående villkorssatser:

```
#define     RUN_FAST
```

i annat fall kommer "liten fördröjningskonstant" att användas.

Uppgift 49

En lämplig registerallokering och detaljering av specifikationen för DisplayML5 leder till följande flödesplan:



Flödesdiagrammet innehåller nu anrop av subrutinen ShortDelay, enligt ovan, välj nu fördröjningskon-stanten 1 för Run-alternativet.

Skapa ny källtextfil `ML5drvr.s12` med subru-tinerna DisplayML5 och ShortDelay. Startadressen till de sex siffrorna som ska skrivas ut förutsätts, precis som tidigare, finnas i X-registret.

Deklarera en tabell seg_tab med segmentmönstren du bestämde i Uppgift 48.

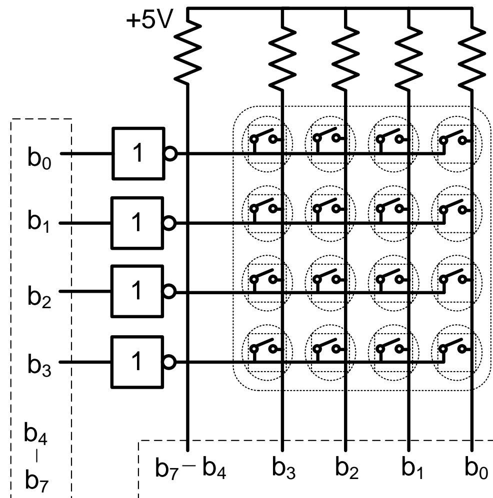
Skapa en ny källtextfil `Main2.s12` med ett huvudprogram liknande det i Uppgift 44 fast denna gång för att testa subrutinen DisplayML5.

Testa din lösning genom att stega igenom DisplayML5. Prova dig slutligen fram till en fördröjningskonstant som ger ungefär samma fördröjning vid RunFast, som fördröjningskonstanten 1 ger vid Run.

Slut Uppgift 49

Tangentbordsrutin för ML5

Studera *Principskiss ML2/ML5 tangentbord*.



Principskiss ML5/ML2 tangentbord

För att känna av en tangentnedtryckning måste någon rad hos utporten ML5_Rows aktiveras. Radernas b_0-b_3 är anslutna till vardera fyra onkopplare på tangentbordet via inverterare. Genom att sätta någon bit till '1' aktiveras motsvarande rad.

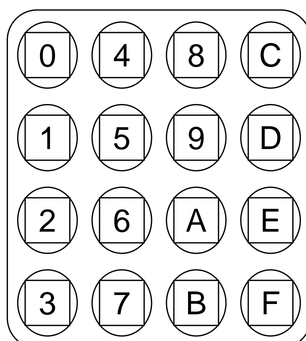
Efter att en rad aktiverats kan kolumnerna läsas av från adressen inporten ML5_Columns. Kolumnernas bitar b_3-b_0 är kopplade till 5 Volt via ett motstånd.

En tangents omkopplarfunktion är sluten då tangenten trycks ned och öppen då tangenten släpps upp.

Om någon rad aktiverats och någon av kolumnerna har logikvärdet '0' betyder det att någon tangent måste vara nedtryckt (det kan vara fler än en tangent). Om ingen tangent är nedtryckt så läses endast ettor från kolumnerna.

Med vetskap om vilken rad som aktiverats och vilken kolumn som ger indikation ('0') vet vi den nedtryckta tangentens position och kan därför också bestämma dess tangentkod.

Tangenterna ska motsvaras av koder enligt följande figur:



Tangentkoder för ML5/ML2

Uppgift 50

Komplettera filen `Labdefs.s12` med adress-definitioner (se även appendix A):

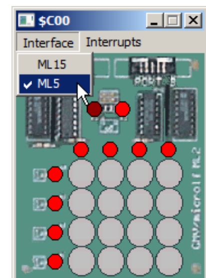
```
; Labdefs.s12
ML15_KeyBoardRegister       EQU     $9C0
ML15_DisplayMode            EQU     $9C2
ML15_DisplayData            EQU     $9C3
ML5_Segments                EQU     $C02
ML5_Position                EQU     $C03
ML5_Rows                    EQU     $C00
ML5_Columns                 EQU     $C01
```

Tangentbordets konstruktion gör det omöjligt att söka av hela bordet samtidigt. I stället måste raderna aktiveras i turordning och för varje aktiverad rad måste kolumnerna läsas av och en eventuell nedtryckt tangent detekteras.

Skapa nu en enkel testsekvens för att undersöka tangentbordet ML3/ML5, och samtidigt kontrollera att portadresser är korrekta. Utforma sekvensen enligt följande:

```
; ml5test2.s12
; Avsök tangentbord
    USE     Labdefs.s12
    ORG     $1000
start:
; radmönster för rad 4
    LDAA    #8
rept:
    STAA    ML5_Rows
    LDAB    ML5_Columns
    CMPB    #$FF
    BNE     decode
    RORA
    BCC     rept
    BRA     start
; tangent nedtryckt
decode:
    BRA decode
```

Anslut IO-simulatorns enhet ML3 Keyboard till adress $C00_{16}$.



Kontrollera att:

- Interface = ML5

Utför nu instruktionssekvensen i simulatorn och ange i följande tabell, för varje tangentkod, vilket radmönster och vilket kolumnmönster som associeras till tangenten.

Tangent	Radmönster	Kolumnmönster
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
A		
B		
C		
D		
E		
F		

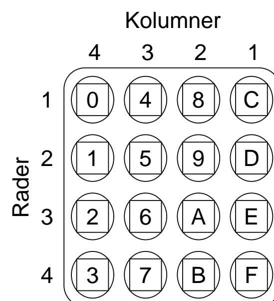
Slut Uppgift 50

En nedtryckt tangent identifieras alltså med en aktiv rad och en aktiv kolumn. Då programmet upptäckt en nedtryckt tangent ska det också bestämma koden för denna tangent.

Med programsekvensen i Uppgift 50 kommer radmönster(dvs 1,2,4 eller 8) att finnas i register A, medan det inverterade kolumnmönstret ($F7_{16}$, FB_{16}, FD_{16} eller FE_{16}) finns i register B. Att översätta detta till motsvarande tangentkoder kräver 16 olika jämförelser.

En bättre metod är att först översätta radmönster till radnummer och kolumnmönster till kolumnnummer och beräkna ett index (0-16) för den nedtryckta tangenten:

tangentkodindex = (kolumnnummer*4)+radnummer



Tangentkoder, rader och kolumner

Översättning från mönster till kod är likartad för rad och kolumn, dock måste kolumnmönstret inverteras innan översättningen som nu blir:

Rad-/Kolumn- mönster	Rad-/Kolumn- nummer
1	1
2	2
4	3
8	4

Uppgift 51

Fortsätt nu med kodsekvensen från Uppgift 50 och implementera en översättning från radmönster och kolumnmönster till tangentkod.

Definiera en konstant textsträng:

```
keycode:  FCB   tangentkoder...
```

Då avkodningssekvensen är klar ska register B innehålla tangentkoden för den nedtryckta tangenten.

Testa hela programsekvensen och övertyga dig om att den fungerar korrekt.

Slut Uppgift 51

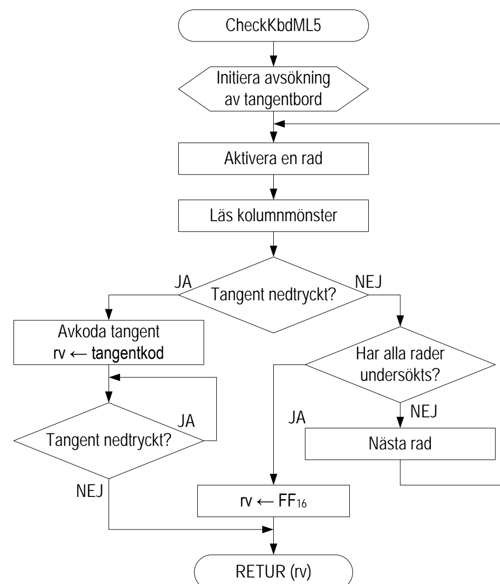
Uppgift 52

Redigera nu filen `ML5drvr.s12` och implementera subrutinerna CheckKbdML5 och GetKbdML15.

Subrutin CheckKbdML5 skall avsöka hela tangentbordet (rad för rad) en gång. Om ingen tangent är nedtryckt ska rutinen returnera FF_{16} i processorns B-register (byte). Om däremot en tangent är nedtryckt ska:

- Tangentens kod bestämmas
- Programmet skall vänta tills tangenten släpps upp igen (dvs. subrutinen `CheckKbd` skall inte lämnas innan tangenten är släppt)
- Tangentens kod returneras i processorns B register

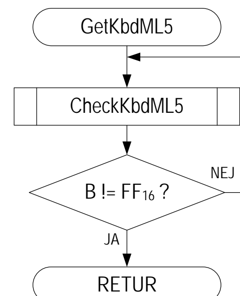
Följande flödesplan beskriver också funktionen:



Testa rutinen genom att sätta en brytpunkt på instruktionen omedelbart efter anropet till *CheckKbd* i huvudprogrammet.

- Kontrollera att *alla* tangentnedtryckningarna ger korrekt värde i register B.
- Kontrollera också att subrutinen "hänger" så länge en tangent är nedtryckt.

Implementera och testa slutligen subrutinen GetKbdML5 enligt följande flödesplan:



Slut Uppgift 52

Programpaket:Stoppur

Avslutningsvis ska du nu konstruera ett mikrodatorbaserat stoppur. Du kommer att använda såväl tangentbordet som sifferindikatorn.

Observera att vi inte är så noga med ”verklig tid” i denna övning. Då vi säger ”stoppur” menar vi att konstruktionen ska *fungera* som ett stoppur. Vi kräver inte att vårt stoppur ska visa korrekt tid.

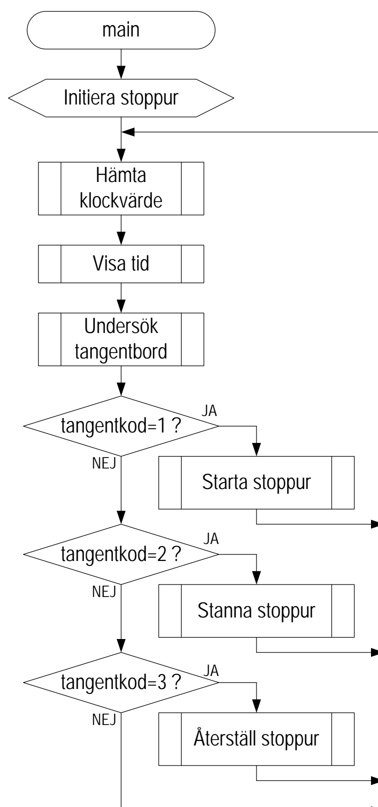
Huvudprogrammet

Huvudprogrammet och dess subrutiner skrivs i filen `Main.s12`, vi fortsätter att använda filen `Labdefs.s12` och de båda gränssnittsberoende `Ml5drvr.s12` och `Ml15drvr.s12`.

Uppgift 53

Konstruera ett huvudprogram enligt följande flödesplan. Använd *ML5* för att *visa tid* och *undersöka tangentbord*. Använd ofullständiga så kallade "dummy"-versioner för subrutinerna:

- GetTime (Hämta klockvärde)
- StartClock (Starta stoppur)
- StopClock (Stanna stoppur)
- ResetClock (Återställstoppur)



Assemblera och testa huvudprogrammet, kontrollera att det utförs enligt flödesplanen

Slut Uppgift 53

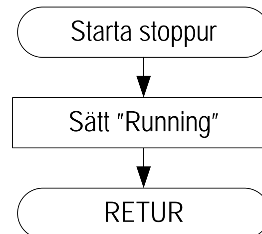
Starta stoppur (StartClock)

Stanna stoppur (StopClock)

Inför en global variabel Running som sätts till 1 då stoppuret startas och som sätts till 0 då stoppuret stannas. Running kan då sägas vara en indikator för om klockan är i gång.

Uppgift 54

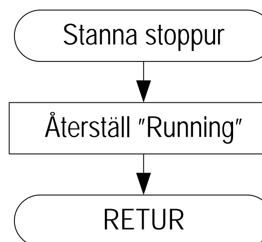
Implementera subrutinen StartClock enligt följande flödesdiagram:



Slut Uppgift 54

Uppgift 55

Implementera subrutinen StopClock enligt följande flödesdiagram:



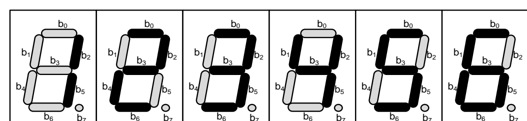
Slut Uppgift 55

Återställ stoppur (ResetClock)

Tips: Deklarera en klockvariabel *clock*, 8 bytes (ty då kan denna skrivas direkt till displayrampen av rutinen *Display*, observera dock att det då bara är de sex första positionerna som används för aktuell tid). Om klockan är startad ska en tiondels sekund läggas till klockvariabeln.

Adress	Data	Kommentar
clock	01	Timmar
clock+1	02	Tiotals minuter
clock+2	03	Minuter
clock+3	04	Tiotals sekunder
clock+4	05	Sekunder
clock+5	06	Tiondelar

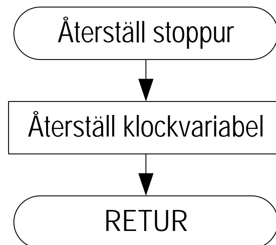
Med värden enligt figuren ovan ska då visningsenheten visa följande:



Klockan återställes genom att alla klockans fält nollställs.

Uppgift 56

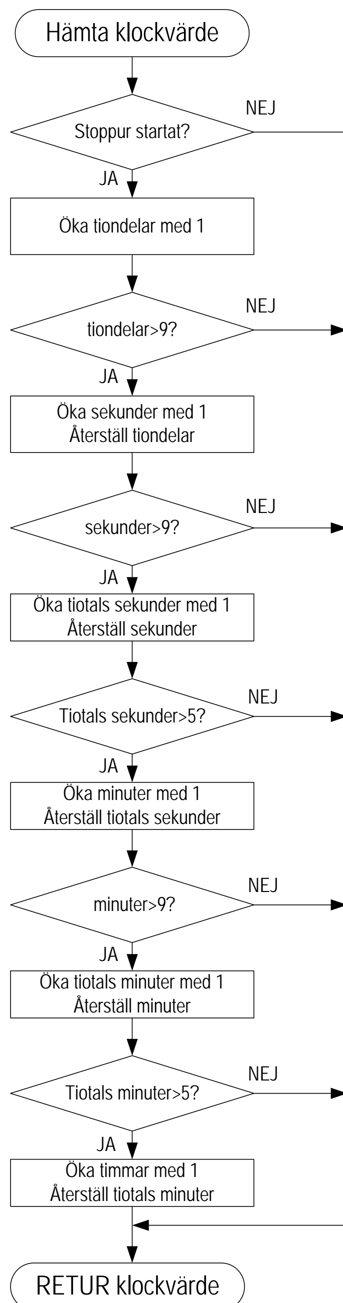
Implementera subrutinen ResetClock (dvs. nollställ klockan) enligt följande flödesdiagram:



Slut Uppgift 56

Hämta klockvärde (GetTime)

Subrutinen `GetTime` ska returnera start-adressen till en 6 bytes variabel som innehåller stoppurets tid. Rutinen ska också hantera tiden i systemet.



Vi antar att GetTime anropas 10 gånger per sekund, Då stoppuret är startat ska tiden (klockvariabeln) alltså uppdateras 10 gånger i sekunden (vi är dock inte speciellt noga med att klockan går rätt). Om stoppuret är startat ska rutinen *först* öka klockans tid med en tiondels sekund, addera ev. hel sekund, hel minut osv. Det går tio tiondelar på en sekund och 60 sekunder på en minut etc. Därefter returnera stoppurets tid.

Uppgift 57

Implementera GetTime, färdigställ programpaketet och testa funktionen.

Kontrollera att klockan kan

- startas
- stannas
- återstartas
- nollställas

Slut Uppgift 57

.

Avsnitt 3

Synkronisering och avbrott

Syften:

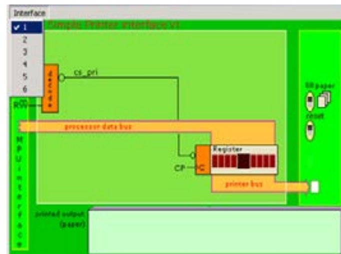
I detta avsnitt illustrerar vi hur centralenhetens arbetstakt måste synkroniseras med yttre enheter. Vi visar olika former av synkronisering, i programvara och med hjälp av speciell hårdvara. I avsnittet ges också en introduktion till "undantagshantering".

Inledning

I detta avsnitt skapar vi en anslutning till en skrivare. En sådan anslutning kräver att vi konstruerar en *port* med ett *gränssnitt* genom vilket centralenhet och skrivare kan kommunicera.

Gränssnittet är avsett för parallell in- och ut-matning. Vi bygger upp porten bit för bit, både hård- och mjukvaru- mässigt. Vi studerar problem, olika lösningar och dess fördelar och nackdelar. Allt eftersom, kommer vi att modifiera vår "skrivare", totalt har detta resulterat i sex olika varianter som vi helt enkelt benämner "Printer gränssnitt V1-V6".

Val av gränssnitt görs via menyn 'Interface':



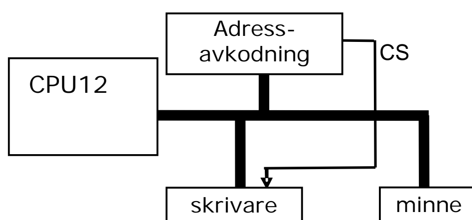
Val av printer-gränssnitt

Printergränssnitt V1

Vi börjar med en mycket enkel skrivare:

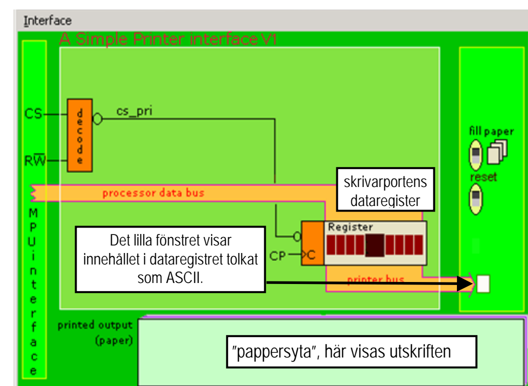
- Skrivaren kan endast arbeta med ett tecken i taget.
- Skrivaren klarar av att ta emot och skriva maximalt 4 tecken per sekund.

Gränssnittet, som fysiskt byggs samman med mikrodatorsystemet har som uppgift att anpassa arbetstakten i det snabba mikrodatorsystemet till den betydligt långsammare skrivaren.



Printer gränssnitt

Signalen CHIP SELECT (CS) aktiveras av ett adressavkodningsblock och är aktiv för adressintervallet 0800_{16}-0803_{16}. Tillsammans med en aktiv skrivsignal *Read/Write* (R/W) bildar dessa en aktiveringssignal *cs_pri* (Chip Select Printer) till ett register anslutet mellan datorsystemets databuss och skrivarens buss. Vid en klockpuls och om samtidigt *cs_pri* är aktiv, överförs alltså ett åtta bitars värde från processorns databuss till skrivaren via skrivarbussen.



Översikt, simulering av skrivare, Version 1

```
; Med portdefinition
PRINTER:    EQU     $0800
; och instruktionerna
            LDAB    #$30
            STAB    PRINTER
; överförs hexadecimala värdet 30 till
; skrivaren.
```

I simulatorns högra fält finns två "knappar" med följande funktioner.

- fill paper – då "pappersytan" fyllts med tecken kan den rensas genom att du klickar på denna knapp.
- reset – återställer simulatorn till dess begynnelsetillstånd.

Låt oss nu betrakta en första tänkbar utskrifts-rutin. I följande exempel ska texten *”Hej Du Kalle!”* skrivas ut:

```
; Printer V1_0
        ORG       $1000
; Pekare till textsträng -> X
        LDX       #Text
; Tecken -> B, peka på nästa
Loop:   LDAB      1,X+
        BEQ       Exit
; Skriv ut till port
        STAB      PRINTER
; Fortsätt med nästa tecken
        BRA       Loop

Exit:   NOP
        BRA  Exit

; så här används assemblerdirektiv för
; att skapa textsträngen med
; slutmarkering (0) på adresss 3000:
        ORG       $3000
Text:   FCS      ”Hej Du Kalle!”
        FCB       0
```

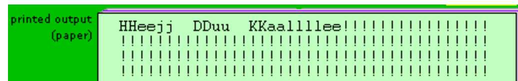
I det enkla programförslaget används register `X` för att peka ut tecknet som skall skrivas. Efter att det första tecknet laddats uppdateras `X` för att peka ut nästa tecken innan detta skickas till skrivaren, osv. Teckensträngen avslutas med tecknet 0, vilket får utskriften att avslutas.

Uppgift 58

Redigera en källtextfil enligt exemplet ovan under namnet `Printer_V1.s12`, assemblera den och ladda till simulatorn.

Anslut IO-simulatorn Simple printer till adress 800_{16}.

Testa programmet med Run. Textsträngen skrivs inte ut korrekt:



Eftersom skrivaren, då den väl startats, skriver ut 4 tecken per sekund så hinner programmet inte med, utskriften innehåller därför duplikat.

Pappret fylls dessutom med utropstecken. Observera skrivarbussens fönster och det sista värdet som placerats i skrivarens dataregister. Den enkla skrivaren läser hela tiden skrivarbussen värde och skriver detta på pappret. Eftersom registret innehåller strängens sista tecken kommer därför detta tecken att fylla ut pappret. Det finns ingen mekanism för att stoppa skrivaren då utskriften är klar.

Slut Uppgift 58

Vi ska nu succesivt ta hand om dessa problem, låt oss börja med att jämföra arbetstakten hos våra respektive delsystem, dator och skrivare.

Uppgift 59

Starta om programmet, denna gång med Run Fast och studera förloppet. Vad skrivs på skrivarens papper

Försök förklara varför texten nu inte skriv ut korrekt. Tänk på hur snabbt (hur ofta) ett tecken skickas till skrivaren. Programslingan upptar 4 instruktioner vilket inte tar speciellt lång tid jämfört med skrivaren som skriver ut 4 tecken per sekund.

Slut Uppgift 59

Följande tabell beskriver *ungefärliga* exe-kveringstider för olika användningar av simulator och verklig hårdvara:

Fall	Instruktioner/sek.
Simulator 'Run'	10
Simulator 'Run Fast'	1000
MC12 Hårdvara	1 000 000

Uppgift 60

Försök nu göra en uppskattning av *utskriftshastig-heten*, dvs. hur många tecken per sekund som kan skickas till skrivaren. Den del av programmet som skriver ut ett tecken omfattar 4 instruktioner. Bestäm först hur lång tid det tar att utföra ett sådant varv i programslingan. Bestäm därefter hur många tecken per sekund som skickas till skrivaren.

Fall	1 varv	tecken/sek
Simulator 'Run'		
Simulator 'Run Fast'		
MC12 Hårdvara		

Slut Uppgift 60

Det snabba mikrodatorsystemet måste synkroni-seras med den långsamma skrivaren. Det kan man göra på flera olika sätt, exempelvis kan man skapa en fördröjning i utskriftsrutinen för att få denna att "gå i takt" med den långsammare skrivaren. Denna lösning skulle i princip fungera om tecken matas ut till skrivaren med exakt samma hastighet som skrivaren kan ta hand om tecknen och i sin tur skriva ut dessa på pappret.

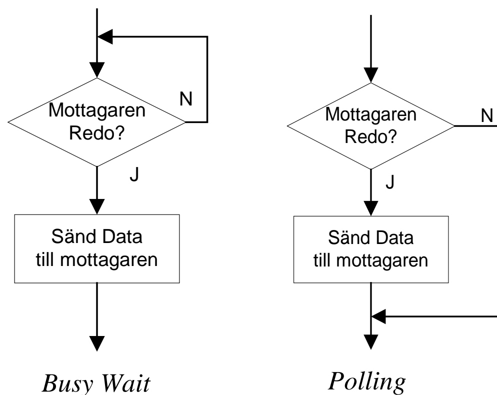
En sådan *ovillkorlig överföring* förutsätter att skrivaren är klar att ta emot ett tecken precis när utskriften inleds. Startögonblicken för dator-system och skrivare måste därför vara samtidiga. För riktigt långa utskrifter kommer det sannolikt att bli fel efter ett tag även om startögonblicket synkroniseras, eftersom det är mycket svårt att få två olika klockor, en i skrivaren och en i datorsystemet, att gå exakt lika fort. Felet kommer att uppträda som missade tecken eller duplikat. Problematiken återkommer vid alla former av ovillkorlig överföring mellan sändaren och mottagare.

Villkorlig överföring

Vi skall nu övergå till olika typer av *villkorlig överföring*. Vi behandlar två olika principer, nämligen ”upprepad statustest” (Busy Wait) och ”rundfrågning” (Polling).

För villkorlig överföring krävs någon form av signalering, ”handskakning”, mellan sändare och mottagare.

- Busy Wait innebär att sändaren inväntar att mottagaren skall bli redo att acceptera data.
- Vid Polling kontrolleras om mottagaren är redo, om den inte är det fortsätter sändaren med annat arbete (exekverar övrig kod).



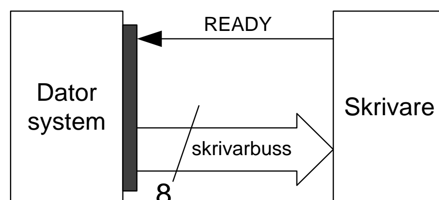
Busy Wait *Polling*

För vårt skrivargränssnitt är uppenbarligen Busy Wait-alternativet det lämpliga. Anledningen till detta är att så fort skrivaren är klar med att skriva ut ett tecken så förutsätter den att ett nytt tecken finns på skrivarbussen och därför måste då mikrodatorsystemet skicka nästa tecken omedelbart då skrivaren blivit redo. Om vi använder Polling är risken stor att programmet missar den korta tidslucka under vilken ett nytt tecken måste skickas till skrivaren.

Oavsett om vi använder *upprepad statustest* eller *rundfrågning* för villkorlig överföring måste vi nu göra ingrepp i hårdvaran.

Printer gränssnitt V2

För att möjliggöra villkorlig överföring inför vi nu en READY-signal hos vår skrivare.



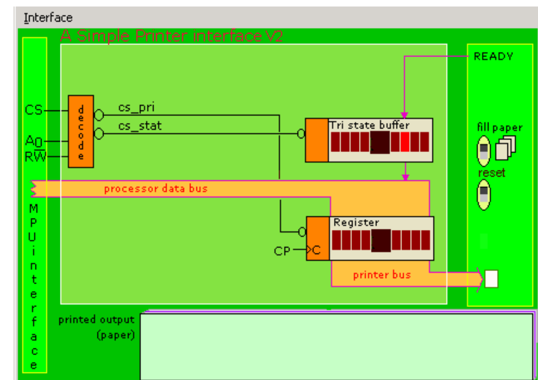
Skrivarport med READY-signal.

READY-signalen definieras enligt:

- READY = 1 (Hög nivå) indikerar att skrivaren är klar att ta emot ett nytt tecken.
- READY = 0 (Låg nivå) indikerar att skrivaren är upptagen med att skriva ut ett tecken.

För att en sändare ska kunna kontrollera skrivarens READY-signal införs nu ytterligare ett register i gränssnittet. Registret som samlar olika signaler *från* skrivaren kallas *statusregister*.

Betrakta nu ”Printer gränssnitt V2”, vi har här utökat avkodningslogiken, lagt till en ”three-state” buffer och kopplat skrivarens READY-signal till bit 2 på denna buffer. Den nya signalen cs_stat aktiveras vid en läsning på adress 0801_{16}.



Simulatorbild för Printer gränssnitt V2

För att implementera Busy Wait-slingan skapar vi en programsekvens som upprepas så länge READY-flaggan är 0, dvs. skrivaren är upptagen. Detta innebär att vi måste testa bit 2 i statusregistret. Först läses hela statusregistret till processorns register B (`LDAB`), därefter testas *enbart* bit 2 (`BITB #%00000100`). Observera att operanden ska vara en ”bitmask”, där den eller de bitar som ska ingå i testet är 1 och övriga bitar är 0. Det spelar ingen roll om vi skriver operanden på binär eller hexadecimal form, vilket leder oss till följande:

```
    ...
test:
    LDAB    $0801   ; läs status
    BITB    #$4     ; testa bit 2
    BEQ     test    ; om 0, fortsätt
    ...
```

I instruktionsuppsättningen för CPU12 ingår `BRCLR` (test and branch on clear) som utför läsning, test och villkorligt hopp, dvs. hela sekvensen ovan, som en enda instruktion. Med denna kan vi då koda hela Busy Wait-slingan:

```
    ...
test:
    BRCLR   $801,#4,test
    ...
```

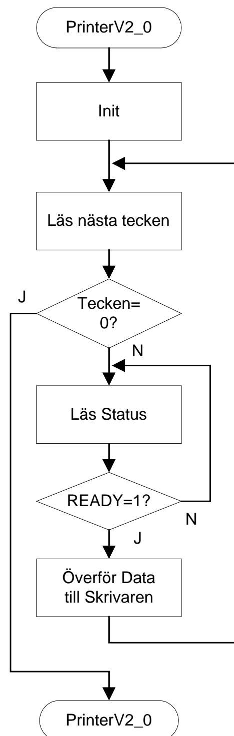
Det finns också en instruktion `BRSET` (test and branch om set) som på motsvarande sätt testar om en bit är 1.

Utgående från det enkla exemplet med föregående gränssnitt skapar vi nu en programsekvens för utskrift via gränssnittet V2.

Uppgift 61

Utforma en utskriftsekvens för Printer gränssnitt V2 enligt följande flödesplan.

I denna version har vi lagt till sekvensen där programmet läser statusregistret för att undersöka READY-biten från skrivaren. Detta upprepas tills biten indikerar att ett nytt tecken kan tas emot av skrivaren.



Skapa en källtextfil `Printer_V2_0.s12` och redigera enligt ovanstående exempel. Assemblera, rätta eventuella fel och ladda till simulatorn.

Testa din nya lösning, med Interface 2, genom att stega programmet ett antal instruktioner så att du ser hur åtminstone ett par bokstäver skrivs på pappret.

Starta om simulatorn i läget Run och testa.

Slutligen startar du om på nytt och testar Run Fast. Jämför utskrifterna med tidigare försök.

Betrakta flödesdiagrammet ovan igen. Direkt efter det att skrivaren blivit redo skickas ett tecken, processorn hämtar därefter nästa tecken och efter detta testas på nytt om READY=1.

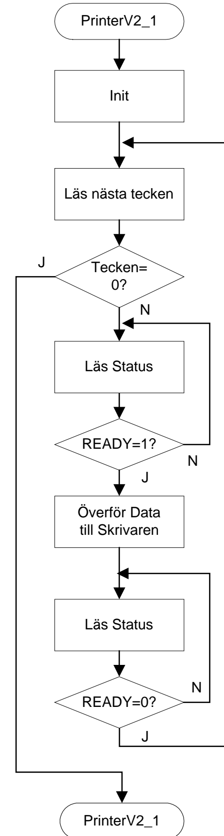
Eftersom skrivaren är betydligt långsammare än datorsystemet behåller den sin READY-signal hög tillräckligt länge så att när processorn läser statusregistret på nytt är READY fortfarande ettställd. Detta innebär att datorsystemet skickar iväg flera tecken åt gången utan att skrivaren hinner läsa dessa, dvs. tecken i skrivarens dataregister skrivs över med nya tecken. Skrivaren kommer därför att missa tecken vilket illustreras då vi simulerar med Run Fast.

Slut Uppgift 61

Uppgift 62

Orsaken till problemen med programsekvensen i föregående uppgift är att programmet inte kontrollerar att skrivaren slår om från ”redo” till ”upptagen”. En programsekvens enligt följande flödesplan kan komma tillrätta med detta.

Efter att ett tecken matats ut till skrivaren inväntas att READY går låg för att inte ett nytt tecken ska kunna skrivas ut omedelbart. Programsekvensen har därför två Busy Wait-slingor, en som testar att READY-signalen är hög (skrivare ”redo”) och en som testar att READY-signalen är låg (skrivare ”upptagen”).



Redigera en ny källtextfil `Printer_V2_1.s12` enligt flödesplanen ovan. Assemblera och rätta eventuella fel. Testa nu din lösning:

Vad händer vid 'Run Fast' ?

Eftersom det sista tecknet som skrivs till skrivarens dataregister ligger kvar kommer detta att skrivas ut på nytt varje gång skrivaren blir ”redo”. I vårt exempel kommer därför utropstecken fortfarande att fylla hela papperet.

Vad händer vid 'Run' ?

Vid långsam simulering hinner nu inte programmet med att skicka nya tecken till skrivaren och av samma anledning som ovan, dvs. senast utskrivna tecken finns kvar i skrivarens dataregister, får vi nu också dubletter av tecknen vid utskrift.

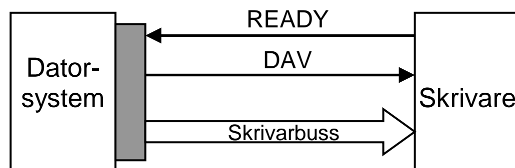
Slut Uppgift 62

Printergränssnitt V3

Vi har nu lyckats synkronisera datorsystem och skrivare så tillvida att programmet kan kontrollera att skrivaren verkligen är redo att ta emot ett tecken och först därefter överföra detta utan att förstöra ett tidigare tecken i skrivarens dataregister.

Kvar står problemet att skrivaren inte har möjlighet att skilja ett nytt tecken från ett tidigare utskrivet tecken vilket innebär att då skrivaren väl startats kommer den att fortsätta skriva ut tecken från dataregistret tills pappret tar slut eller man återställer skrivaren genom att göra reset.

En enkel och effektiv lösning är att vi nu utrustar skrivaren med en ny handskakningssignal som indikerar att det finns ett nytt giltigt tecken tillgängligt på skrivarbussen för skrivaren att trycka på pappret. Vi inför därför nya förutsättningar för skrivaren och antar att den nu även är utrustad med en insignal data available (DAV).

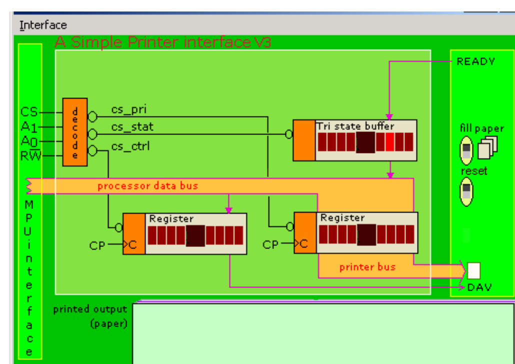


Skrivarport med READY- och Data AVailable signaler.

Signalen READY har samma funktion som tidigare och den nya handskakningssignalen DAV definieras enligt:

- DAV = 1 (Hög nivå) indikerar för skrivaren att ett nytt tecken finns att hämta på skrivarbussen.
- DAV = 0 (Låg nivå) indikerar för skrivaren att skrivarbussen har ett ogiltigt värde.

Skrivarporten måste nu förses med ett nytt register för att kunna ta in DAV-signalen. Vi kompletterar därför adressavkodningen så att denna nu också genererar signalen cs_ctrl, för adress 0802_{16} och låter denna aktivera skrivarens *styrregister*. Skrivarens signal DAV ansluts också via registret till bit 1 på systemets databuss.



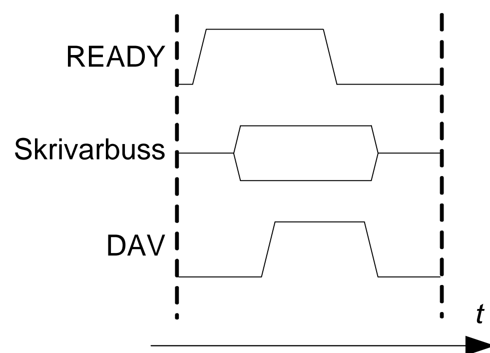
Simulatorbild för Printer gränssnitt V3

Vi har nu två handskakningssignaler READY och DAV som vi använder för att få det snabba datorsystemet och den långsamma skrivaren att kommunicera en säker överföring av data. På så sätt kommer inga dubbletter eller missade tecken att finnas i utskriften. I följande tabell ges en detaljerad beskrivning av händelseförloppet i text. Vi förutsätter då att en utskrift pågår och att skrivaren är upptagen med att skriva ut ett tecken. Detta innebär att READY = 0 från början.

Händelser i Datorsystemet	*Händelser i skrivaren*
Inväntar READY=1	Skrivaren är upptagen med att skriva ut ett tecken. READY=0.
	Skrivaren är redo för nästa tecken och sätter READY=1
När READY=1 skrivs nästa tecken till skrivarens dataregister. Sätter DAV=1	Inväntar DAV=1
Inväntar READY=0	Ser att DAV=1. Läser nytt tecken från skrivarbussen. Signalerar upptagen, READY=0.
När READY=0 nollställs DAV som indikation på att det inte finns giltigt tecken på skrivarbussen	Skrivaren är upptagen med att skriva ut ett tecken. READY=0.

Händelser i datorsystem och skrivare vid handskakning

Samma förlopp illustreras av följande tidsdiagram:

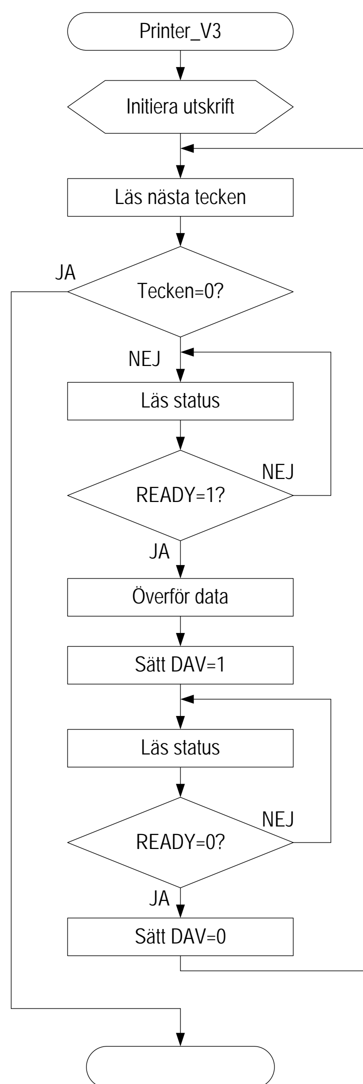


Tidsdiagram över handskakningsförloppet

I tidsdiagrammet, som visar överföring av ett tecken mellan datorsystem och skrivare, visas förhållanden mellan de olika signaler som ingår för överföringen. Överföringen är asynkron.

Uppgift 63

Följande programförslag kan nu utformas för version 3 av vår skrivare.



Redigera en källtextfil `Printer_V3.s12` enligt förslaget ovan. Assemblera och rätta eventuella fel.

Testa programmet med Run respektive Run Fast och kontrollera att alla tecken skrivs ut på rätt sätt.

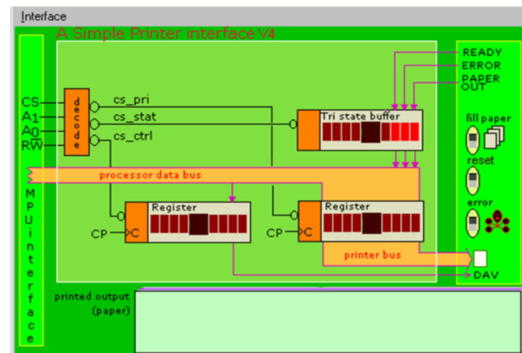
Slut Uppgift 63

Vi har nu konstruerat ett hårdvarugränssnitt för en mycket enkel skrivaranslutning. Vi har också konstruerat och testat programsekvenser för att kontrollera att gränssnittet fungerar som det ska. Vår konstruktion kan skicka en godtyckligt lång textsträng till skrivaren för utskrift.

I ett mer realistiskt fall är detta inte tillräckligt, händelser som gör att utskriften måste avbrytas, exempelvis då pappret tar slut eller om det uppstår något annat fel som måste åtgärdas handgripligt ska också kunna hanteras. Mot denna bakgrund studerar vi nästa variant av vårt gränssnitt.

Printergränssnitt V4

I simulatorns gränssnitt V4 har skrivaren utökats med två nya statussignaler, PAPER OUT som anger att hela utskriftsytan fyllts ut (slut på pappret) och ERROR en signal som anger något godtyckligt fel i skrivaren. Det finns också en "knapp", error, där vi kan framkalla detta godtyckliga fel.



Simulatorbild för Printer gränssnitt V4

Vi måste nu komplettera definitionen av READY-signalen från skrivaren tillsammans med de nya signalerna:

- READY = 1 (Hög nivå) indikerar att skrivaren är klar att mottaga ett nytt tecken.
- READY = 0 (Låg nivå) indikerar att skrivaren är upptagen, har slut på papper eller att ett fel har uppstått.
- ERROR = 1 (Hög nivå) indikerar att skrivaren fungerar korrekt.
- ERROR = 0 (Låg nivå) indikerar att ett fel uppstått i skrivaren
- PAPER OUT = 1 (Hög nivå) indikerar att skrivaren är fylld med papper
- PAPER OUT = 0 (Låg nivå) indikerar att skrivaren har slut på papper

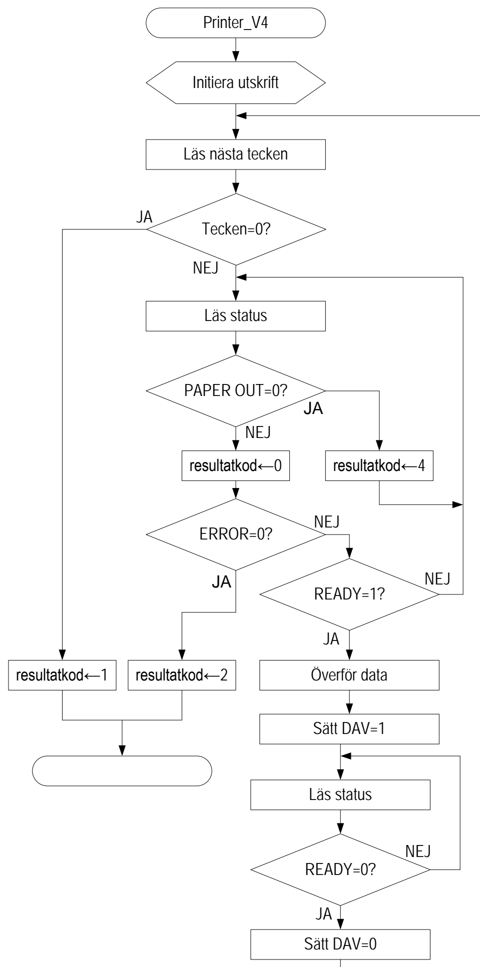
Prova felsignalen genom att klicka på error-knappen och observera hur felsignalen aktiveras (nollställs). Återställ genom att klicka på reset-knappen.

Uppgift 64

Programsekvensen ska modifieras för det nya gränssnittet. Vi ska här också använda ML4 Parallel output för att skriva ut en resultatkod som indikerar eventuella fel eller att utskriften är korrekt.

- Efter att felfritt ha skrivit ut hela textsträngen ska programmet skriva resultatkoden 1 till ljusdioderna.
- Om fel uppstår ska utskriften avbrytas och felet indikeras genom att resultatkoden 2 skrivs till ljusdioderna.
- Då pappret tar slut skall utskriften stanna och statuskoden 4 skrivas till ljusdioderna. Därefter ska programmet invänta påfyllning av papper (fill paper), för att då släcka resultatkoden och fortsätta utskriften.

Flödesplan för programmet:



Redigera en källtextfil `Printer_V4.s12` enligt förslaget ovan. Assemblera och rätta eventuella fel.

Anslut även ML4 Parallel output till adress 400_{16}.

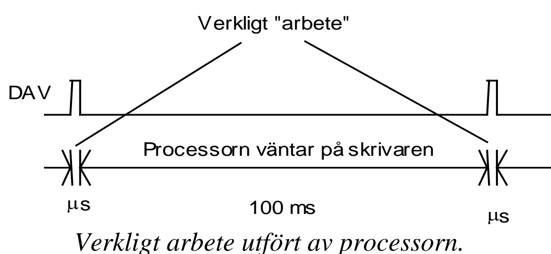
Kontrollera (RunFast) att utskriften blir riktig under normala betingelser.

Upprepa utskriften flera gånger så att pappret fylls ut fullständigt, kontrollera att felhanteringen för ”slut på papper” fungerar som avsett.

Kontrollera slutligen funktionen för ”godtyckligt fel”.

Slut Uppgift 64

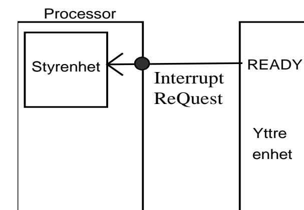
En naturlig fråga vi bör ställa oss nu är hur effektivt processorn utnyttjas under den tid en utskrift pågår. Ur vår synvinkel utför processorn "nyttigt arbete" endast när den skickar ett tecken till skrivaren men däremot inte under den tid den inväntar att skrivaren skall bli redo att ta emot nästa tecken.



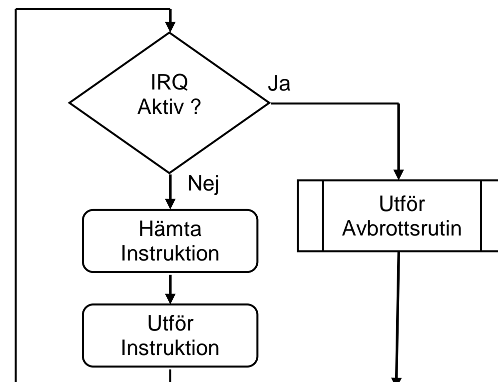
Verkligt arbete utfört av processorn.

Introduktion till "avbrott"

Vi skall nu övergå till att studera en metod där processorn avbryts i sitt normala arbete för att under en kortare tid utföra någon programrutin innan den därefter återupptar sitt normala arbete. Vi skall se mekanismer där en hårdvarusignal används som en signal till processorn som får den att omedelbart avbryta sitt arbete och starta en annan *fördefinierad* programsekvens, ofta kallad *avbrottsrutin*.



Låt oss anta att vi har en signal, det skulle exempelvis kunna vara READY-signalen från skrivaren. Vi kopplar nu denna till en *avbrotts-ingång* hos processorn. Ingången, som vanligtvis kallas IRQ (*Interrupt ReQuest*) är internt ansluten till processorns styrenhet som ansvarar för att hämta instruktioner från minnet och därefter utföra dom. Processorn undersöker denna signal *mellan* utförandet av *varje* instruktion. Betrakta följande flödesdiagram som, generellt, beskriver styrenhetens arbete:

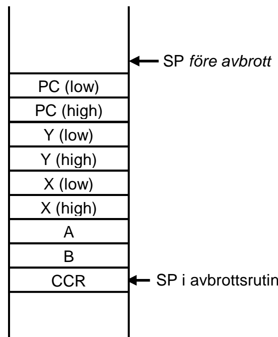


Man skulle kunna jämföra avbrottsrutinen med en subrutin som startas av en signal, snarare än av en instruktion `JSR`.

Hur lokaliseras då avbrottsrutinen för en speciell avbrottsignal? Till varje enskild avbrottsignal associeras en *avbrottsvektor*. Avbrottsvektorn har en förutbestämd adress i minnet och avbrottsvektorns innehåll tolkas som startadress för avbrottsrutinen. Initieringar som placerats först i huvudprogrammet måste skriva in startadressen för avbrottsrutinen (initiera avbrottsvektorn) på den förutbestämda adressen innan det första avbrottet initieras.

Avbrottsvektorernas adressutrymme är normalt sett placerat i ROM (adresser FF80-FFFF) men i laborationssystemet MC12 och i simulatorn har dessa omdirigerats med placering i RWM. Detta innebär i praktiken att vektorns första siffra (F_{16}) alltid ska bytas mot 3 i program för MC12. Se även appendix F.

Vid avbrott sparas alla registerinnehåll på stacken enligt följande figur:



Notera speciellt att RTS (*ReTurn from Subroutine*) inte kan användas vid utträde från avbrottsrutin. Minns att RTS endast återställer PC från stacken. I stället finns instruktionen RTI (*ReTurn from Interrupt*) för detta ändamål. RTI återställer *samtliga* registerinnehåll från stacken.

För att processorn ska acceptera avbrottet måste dess avbrottsmask vara 0. Hos CPU12 finns en bit (I) i CCR som utgör denna avbrottsmask. Om biten sätts till 1 kommer processorn *inte* att utföra några andra avbrott än XIRQ. Om avbrottsmasken sätts till 0 kommer alla avbrott att betjänas. Processorn ändrar automatiskt denna avbrottsmask vid avbrottsbetjäning.

Uppgift 65

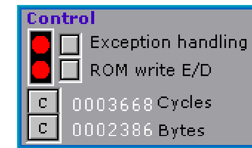
Följande programexempel består av nödvändiga initieringar, huvudprogram och en avbrottsrutin för att illustrera avbrott.

```
* Irq1.s12

Port1:  EQU     $0400
Port2:  EQU     $0401

        ORG     $1000
main:
; Nollställ variabler
        CLR     Var1
        CLR     Var2
; Initiera avbrottsvektor IRQ
        LDX     #IrqR
        STX     $3FF2
; Nollställ I-bit, tillåt avbrott
        CLI

; I huvudprogrammet skrivs
; variabelvärdena till olika utportar.
; Endast 'Var1' ökas dock för varje
; varv i slingan

main_loop:
        INC     Var1
        MOVB    Var1,Port1
        MOVB    Var2,Port2
        BRA     main_loop

; Avbrottsrutin
; Ökar 'Var2' med 1
IrqR:
        INC     Var2
        RTI

; Variabler
Var1:   RMB     1
Var2:   RMB     1
```

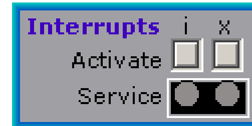
Redigera en källtextfil Irq1.s12 enligt ovan, assemblera och rätta eventuella fel.

Koppla Port1 till adress 400_{16} och Port2 till adress 401_{16} till var sin ML4 Parallel Output.

Kontrollera att Exception handling är aktiverad (ljust röd).



Här kan du aktivera olika avbrott.
OBS Använd endast 'i' för detta exempel eftersom programmet inte initierar för andra avbrott. Då du klickar på knappen 'i' genereras ett avbrott IRQ i simulatorn och motsvarande avbrottsrutin utförs. Notera speciellt att under avbrottsrutinen är dioden 'Service i' tänd.

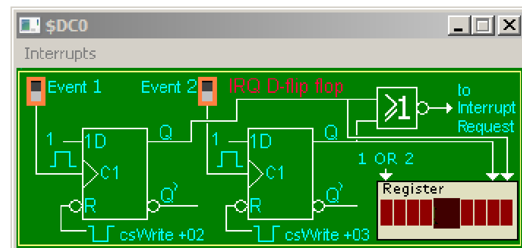


- Välj först Run och observera hur värdet hos den första utporten räknas upp.
- Klicka på knappen 'i' och observera beteendet.

Observera hur ljusdioderna på den andra parallellporten räknas upp för varje avbrott. Studera även hur det blinkar till i indikatorn 'Service' då processorn exekverar en avbrottsrutin.

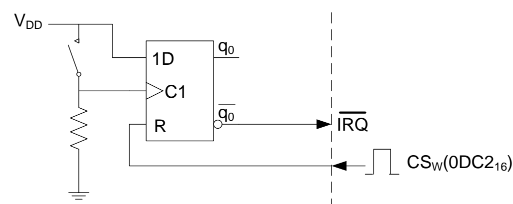
Slut Uppgift 65

Efter att ha använt simulatorns inbyggda funktion för att "generera" (simulera) avbrott går vi nu vidare och exemplifierar en yttre enhets signal genom att införa en enkel tryckknapp utanför processorn. Laborationsmodulen *ML19* innehåller två tryckknappar (här kallade Event 1 respektive Event 2).



ML19 (IRQ Flip Flop) i simulatorn

Principen för kopplingen av en tryckknapp visas i följande figur:



Koppling för yttre avbrottssignal

Vi kan inte ansluta tryckknappen direkt till processorns IRQ-ingång eftersom en enda nedtryckning skulle åstadkomma tusentals avbrott (vi kan inte hålla nere knappen tillräckligt kort tid för att bara generera *en* avbrottsbegäran)

Vi måste därför använda en "vippa" som slår om när vi trycker ner knappen. Därutöver måste vi

programvarumässigt ha möjlighet att återställa vippan.

Förloppet blir då att vi trycker på knappen för att generera avbrott. Programmet (egentligen avbrottsrutinen) återställer vippan så att avbrottsbegäran in till processorn avlägsnas. Med en sådan mekanism spelar det ingen roll hur länge vi håller knappen nertryckt, endast en ny nedtryckning kommer att generera nästa avbrott.

När vi trycker på knappen (sluter den åter-fjädrande strömställarenn) klockas en etta in från D-ingången på vippan. Detta medför att vippan sätts och q_0' blir noll, därmed aktiveras också processorns IRQ-ingång.

Efter att ha tagit hand om avbrottet i avbrotts-rutinen måste också avbrottsbegäran in till processorn avlägsnas. Vi måste därför se till att ge vippan en återställningssignal (en signal in på R-ingången) så att q_0' blir ett.

Detta åstadkoms med hjälp av adressavkod-ningslogik som skapar en puls på vippans R-ingång vid en skrivning (CS_W) på adress $DC2_{16}$.

Observera att värdet som skrivs till adress $DC2_{16}$ är betydelselöst, det är skrivoperationen i sig som genererar CS-signalen och därmed R-pulsen.

Uppgift 66

Lägg till följande portdefinitioner i filen `Labdefs.s12`:

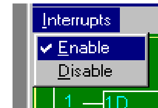
```
;   ML19 definitioner
ML19_Stat           EQU     $0DC0
;   Kvittera händelse 1
ML19_AckIrq_1       EQU     $0DC2
;   Nollställ händelse 2
ML19_AckIrq_2       EQU     $0DC3
```

Redigera en källtextfil `Irq2.s12`, i denna uppgift ska tryckknappen Event1 på ML19 användas för att generera avbrott. Programmet blir praktiskt taget identiskt med föregående uppgifts men avbrottsrutinen måste nu också kvittera avbrottet genom en skrivning på adress $0DC2_{16}$.

```
; Irq2.s12
        use     Labdefs.s12
Port1:  EQU     $0400
Port2:  EQU     $0401

        ORG     $1000
main:
; Nollställ variabler
        CLR     Var1
        CLR     Var2
; Initiera avbrottsvektor IRQ
        LDX     #IrqR
        STX     $3FF2
; Nollställ I-bit, tillåt avbrott
        CLI

; Huvudprogram
main_loop:
        INC     Var1
        MOVB    Var1,Port1
        MOVB    Var2,Port2
        BRA     main_loop

; Avbrottsrutin
; Ökar 'Var2' med 1 och kvittera avbrott
IrqR:   INC     Var2
        CLR     ML19_AckIrq_1
        RTI

; Variabler
Var1:           RMB     1
Var2:           RMB     1
```

- Assemblera och testa programmet.
- Anslut IO-simulator IRQ Flip flop (dvs motsvarigheten till ML19) till adress $DC0_{16}$.
- Aktivera ML19's avbrottsfunktion:

OBSERVERA: Du **måste** försäkra dig om att enhetens avbrottsmekanism är aktiverad (Enable)



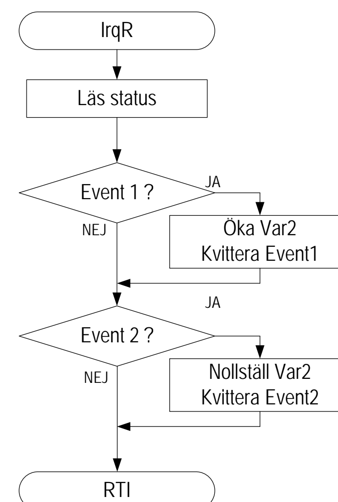
- Sätt en brytpunkt i avbrottsrutinen (högerklicka och välj Enable break on IRQ).
- Starta programmet med Run kontrollera att ljusdioderna för parallellport 1 ändras som tidigare.
- Klicka nu på knappen Event 1 för att simulera avbrott. Observera hur programmet stannas i avbrottsrutinen, Stega nu instruktionsvis genom hela avbrottsrutinen.
- Ta slutligen bort brytpunkten, återstarta, RunFast, klicka upprepade gånger på Event 1 och övertyga dig om att programmet fungerar som det ska.

Slut Uppgift 66

Uppgift 67

Redigera en källtextfil `Irq3.s12`. I denna uppgift ska programmet kunna hantera avbrott genererade från två olika tryckknappar, Event 1 och Event 2. Tryck på Event 1 ska precis som tidigare medföra en ökning av Var1 medan tryck på Event 2 ska nollställa variabeln.

Avbrottsrutinen ska nu modifieras enligt följande flödesdiagram, du måste här kontrollera bitarna i statusregistret för att ta reda på om det är Event 1 eller Event 2 som är orsaken till avbrottet.

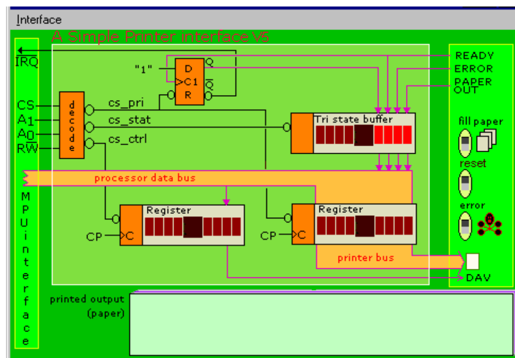


Assemblera, testa och rätta eventuella fel tills programmet fungerar som det ska.

Slut Uppgift 67

Printer gränssnitt V5

Betrakta figuren med *Printer gränssnitt V5* där gränssnittet modifierats så att READY-signalen kan användas för att generera *avbrott*.



Simulator för printer gränssnitt V5

Då READY går hög klockas en etta in på D-vippan vilket medför att Q′ går låg. Q′ är direkt ansluten till processorns IRQ-ingång och på så sätt styr skrivarens READY-signal när avbrott skall ske.

Avbrottssignalen till processorn avlägsnas genom att cs_pri-signalen används för att påverka RESET-ingången på vippan. Detta medför att när avbrottsrutinen skriver ett nytt tecken till skrivaren kommer vippan att nollställas och därmed ettställs Q′ och IRQ till processorn.

Uppgift 68

Vi ska nu se hur en avbrottsdriven tillämpning för detta skrivargränssnitt kan se ut. I applikationen låter vi huvudprogrammet ändra utdata till ML4 Parallel Output så att vi ser att det hela tiden händer något och samtidigt låter vi en avbrottsrutin ta hand om utskriften till skrivaren.

```
; Printer V5_0
        ORG     $1000
main:
; Initieringar
        CLR     count
        CLR     TextP
        LDX     #IrqR
        STX     $3FF2
        CLI

; Huvudrutin
main_loop:
        INC     count
        MOVB    count,$400
        BRA     main_loop

; Avbrottsrutin
IrqR:   LDX     #Text
        LDAB    TextP
        LDAA    B,X
        CMPA    #0
        BEQ     Done
        INC     TextP

        STAA    $800
        MOVB    #2,$802  ; sätt DAV
        CLR     $802     ; nollställ DAV
Done    RTI

Text    FCS     "Hej Du Kalle!"
        FCB     0

count:  RMB     1        ; räknarvariabel
dummy:  RMB     1        ; aktuellt tecken
```

Redigera i en ny källtextfil `PrinterV5.s12` nu ett program enligt exemplet Printer V5_0. Assemblera, och rätta eventuella fel.

För att testa programmet, anslut som tidigare Simple Printer till adress 800_{16}, aktivera gränssnitt 5. Anslut också ML4 Parallel Output till adress 0400_{16}.

Testa nu programmet genom att starta en utskrift och invänta att den tar slut. Klicka därefter på error-knappen på skrivaren. Vad händer?

> Inga avbrott genereras. Huvudprogrammet exekveras normalt.

Klicka nu på "reset"-knappen hos skrivaren och observera förloppet. Vad händer?

> Skrivaren återställs och börjar på nytt generera avbrott.

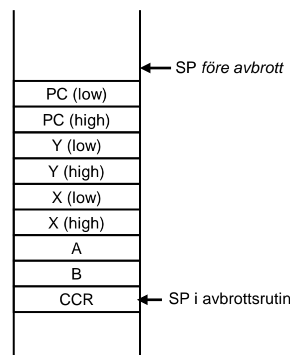
Slut Uppgift 68

Antag att vi skrivit ut ett helt dokument. Efter utskriften är skrivarens READY-signal hög samtidigt som I-flaggan i processorns CC-register är nollställd, följaktligen fortsätter skrivaren att generera avbrott även då utskriften är klar.

Uppgift 69

Efter en utskrift måste programmet ettställa I-flaggan i CC-registret, detta görs då i avbrottsrutinen direkt efter det att textsträngens slutmarkering påträffats.

Eftersom det registerinnehåll som återställs vid RTI-instruktionen ser ut på följande sätt:



kan den befintliga avbrottsrutinen modifieras exempelvis enligt följande:

```
; Avbrottsrutin
IrqR:   LDX     #Text
        LDAB    TextP
        LDAA    B,X
        CMPA    #0
        BEQ     Done
        INC     TextP

        STAA    $800
        MOVB    #2,$802  ; sätt DAV
        CLR     $802     ; nollställ DAV
        RTI

Done:   LDAB    0,SP     ; "CCR" -> B
        ORAB    #$10     ; I-flagga = 1
        STAB    0,SP     ; tillbaks
        RTI
```

När processorn nu utför `RTI` kommer den att återställa `CCR` där avbrottsmasken är ettställd och därmed accepteras inga fortsatta avbrott.

- Ändra i programmet i `PrinterV5.s12` enligt ovan och spara under filnamnet `PrinterV5_1.s12`.
- Testa lösningen, kontrollera att systemet uppför sig på ett bättre sätt.

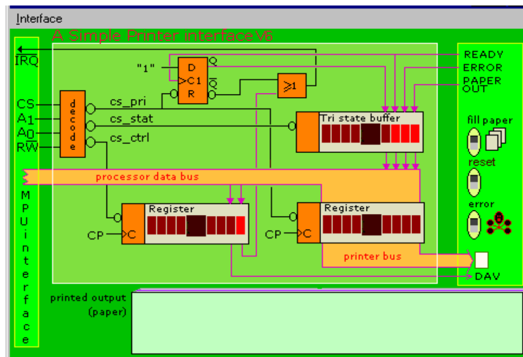
Slut Uppgift 69

Lösningen ovan fungerar alltså så länge vi bara har en avbrottskälla. Vi kan då manipulera avbrottsmasken i `CCR` och utestänga vidare avbrott utan problem. Vanligtvis finns det dock finns ett antal olika yttre enheter som kan generera avbrott. I sådana system kan vi därför inte enkelt manipulera avbrottsmasken på samma sätt.

Det krävs ett nytt gränssnitt där vi har möjlighet att utestänga avbrottet i själva printerporten.

Printer gränssnitt V6

Printer gränssnitt V6 visar vår färdiga lösning i hårdvara.



Simulator för printer gränssnitt V6

Notera hur styrregistret utökats till två bitar. Den nya biten är ansluten till en OR-grind. Den andra ingången på OR-grinden är den tidigare IRQ-signalen. Om programmet nu skriver en etta till b_0 i styrregistret så kommer IRQ aldrig att kunna gå låg. Detta är alltså ett sätt att programvaru-mässigt stänga av enhetens avbrottsmekanism. Vi kallar den nya styrsignalen IrqDisable. Vid RESET sätts denna bit automatiskt till 1, dvs Interrupt Disable. Nu kan sekvensen

```
Done:
    LDAB    0,SP
    ORAB    #$10
    STAB    0,SP
```

bytas ut mot:

```
Done:
    MOVB    #1,$802     ; ettställ bit 0
```

Uppgift 70

I denna avslutande uppgift ska du sammanställa utskrifsrutinen för skrivargränssnitt med avbrottsmekanism.

- Skapa en fil `Printerdefs.s12` med symboliska namn för gränssnittets register och bitar enligt följande:

```
; Printerdefs.s12
pi_printer      EQU     $800
pi_status       EQU     $801
pi_control      EQU     $802
PI_PaperOut     EQU     1
PI_Error        EQU     2
PI_Ready        EQU     4
PI_IrqDisable   EQU     1
PI_Dav          EQU     2
```

- Redigera en ny källtextfil `PrinterV6.s12` där du utgår från lösningen PrinterV4. Definitionsfilen ovan *ska* användas i denna lösning.
- Anpassa lösningen för avbrott med det nya gränssnittet. ML4 Parallel output används för utskrift av resultatkoden precis som i PrinterV4.
- En strömbrytare (Dip switch input, b_0) , ska användas för att starta utskrift. Om bit 0 är 1 så ska utskriften startas, och hela textsträngen skrivas ut.
- Assemblera och rätta eventuella fel.
- Anslut Simple Printer till adress 800_{16}, välj gränssnitt 6.
  Anslut ML4 Parallel output till adress 400_{16}.
  Anslut även ML4 Dip-switch input till adress 600_{16}.
- Testa programmet och kontrollera att det fungerar som avsett.

Slut Uppgift 70

Avsnitt 4

Programmeringsprojekt: Borrmaskinen

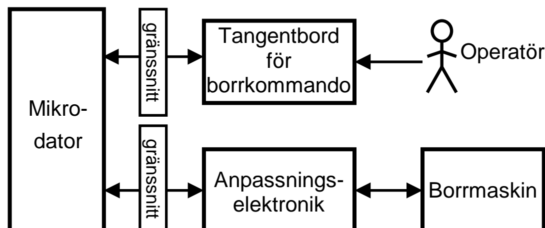
Syften:

I detta avsnitt ska du konstruera en "arbetsrobot" som borrar hål i ett arbetsstycke enligt ett givet schema. Roboten består av en borrmaskin, en styrpanel (ett tangentbord), anpassningselektronik och ett mikrodatorsystem. Genom att självständigt utföra uppgifterna i detta avsnitt förutsätts du få de grundläggande färdigheter som krävs för att själv genomföra ett mindre programmeringsprojekt i assemblerspråk.

Inledning

Ett mikrodatorsystem ska användas för att övervaka och styra en borrmaskin på så sätt att enstaka hål eller kompletta hålmönster kan borras. En operatör skall kunna ställa in önskat borrmönster och utföra borrningen "manuellt" i ett antal deloperationer eller "automatiskt" i en enda komplett operation. Detta sker via styrpanelen (ett litet tangentbord).

Din uppgift är att skriva programvaran till mikrodatorsystemet. Systemet (operatör och hårdvara) beskrivs av följande illustration.



Ett datorsystem som byggts samman med något mekaniskt delsystem på detta sätt kallas ofta för "inbyggt system" (*eng: embedded system*). Eftersom <u>hela</u> systemet består av såväl elektroniska som mekaniska delsystem kallas också hela systemet för "mekatroniskt".

Systemets funktion realiseras med hjälp av programvara. Större delen av detta avsnitt kommer att handleda dig då du konstruerar denna programvara, vi ska dock först bekanta oss lite bättre med systemet.

Borrmaskinen består av ett antal delar, alla placerade på en basplatta. Ett arbetsstycke som skall borras fästs på en axel som vrids av en stegmotor. Efter det att ett hål är borrat kan stegmotorn aktiveras och därmed vrida (rotera) arbetsstycket fram till nästa position där ett nytt hål kan borras. Borrhålen hamnar därför på en cirkellinje på arbetsstycket.

Här följer en kortfattad beskrivning över de olika delarna som tillhör borrmaskinen:

- En vridmagnet används för att sänka borrmotorn. När magneten inte är aktiverad pressar en tryckfjäder borrmotorn uppåt.
- Motor, för att rotera borret (borra).
- Givare som anger borrets läge, bottenläget (borrat i genom arbetsstycket) eller toppläge (viloläge).
- Stegmotor för att kunna vrida arbetsstycket i steg om 7,5° vridning.
- Givare som anger att arbetsstycket är i referensposition (startläge).
- "Alarmindikator" kan användas exempelvis som larm vid fel eller som indikering på att arbetsstycket är färdigborrat.

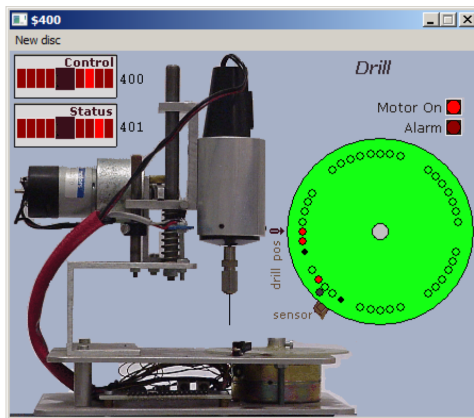
Verklighetens borrmaskin

Systemet skall i allt väsentligt vara operatörsstyrt, och det måste finnas kommandon för att utföra följande operationer:

- starta borrmotorn
- stoppa borrmotorn
- sänk borret
- höj borret
- vrid (stega) arbetsstycket ett steg
- vrid (stega) arbetsstycket till referenspositionen
- borra ett hål
- borra hål enligt ett förutbestämt mönster

Borrmaskinsimulatorn

Simulatorn innehåller förutom borrmaskinen, ett cirkulärt arbetsstycke och indikatorer för styr- och statusregister.



Arbetsstycket i bilden visar ett exempel på ett delvis felaktigt genomfört borrprogram.

- De svarta cirklarna markerar var hål ska borras i arbetsstycket.
- De cirklar som fyllts med röd färg har borrats korrekt.
- En svart punkt utan cirkel visar var hål borrats felaktigt i arbetsstycket.

Observera att när du arbetar med simulatorn kommer vissa realtidsegenskaper att variera. Detta innebär att exempelvis fördröjningsrutiner måste ändras beroende på om du utnyttjar simulatorn i läget Run eller Run Fast. Vi återkommer senare speciellt till detta.

Låt oss nu inledningsvis undersöka hur borrmaskinsimulatorn fungerar och ge dig möjlighet att bli bekant med den.

Uppgift 71

- Komplettera filen `Labdefs.s12` med portadresser enligt följande:

```
; Adress till omkopplare och display
DipSwitch       EQU     $0600
HexDisplay      EQU     $0700
; Adress till borrmaskinens styrregister
DrillControl    EQU     $0400

; ... borrmaskinens statusregister
DrillStatus     EQU     $0401
```

- Redigera också en fil `Drilltest1.s12` enligt följande:

```
; Drilltest1.s12
    USE     Labdefs.s12
    ORG     $1000
Dtest1:
    LDAA    DipSwitch       ;Läs styrord
    STAA    DrillControl    ;Skriv styrord
    BRA     Dtest1
```

- Assemblera `Drilltest1.s12` och ladda till simulatorn.

Slut Uppgift 71

Uppgift 72

- Anslut IO-simulatorns ML4 Dip switch input till adress 600_{16} och Drill till adress 400_{16}.

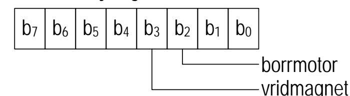
Med borrmaskinsimulatorns menyval New Disc kan du skapa ett nytt arbetsstycke. Det finns fyra olika att välja mellan och arbetsstycke 4 används för slutuppgiften i detta avsnitt.

- Klicka upprepade gånger på ett och samma arbetsstycke exempelvis '4' och observera hur markören för referensposition (svart fylld cirkel i arbetsstyckets periferi) hamnar slumpmässigt på olika ställen.

Du kan läsa mer om IO-simulator's borrmaskin i ETERM's hjälpsystem.

- Kontrollera att alla strömbrytarna på omkopplaren är nollställda.
- Starta programmet genom att välja Run i simulatorn.
- Studera indikatorerna för borrmaskinens styrregister.

Control (styrregister)

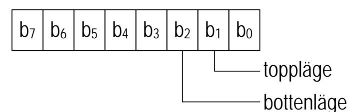


b_3, vridmagnetens funktion
- b_3=1, borr sänks
- b_3=0, borr höjs

b_2, borrmotorns funktion
- b_2=1, borr roterar
- b_2=0, borr stillastående

- Starta borrmotorn genom att ettställa b_2, dvs. ställa in styrordet 00000100_2 på omkopplaren. Kontrollera att indikatorn för Motor On tänds i borrmaskinsimulatorn.
- Observera borrmaskinens statusregister och notera att b_1 är 1 vilket indikerar att borret är i toppläge.

Status



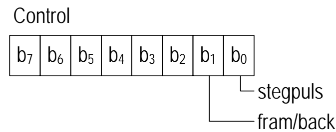
b_2=1, borr i absolut bottenläge
b_1=1, borr i absolut toppläge

- Ettställ nu även b_3 på omkopplaren för att ge styrordet 00001100_2 $(0C)_{16}$.
  Observera hur borret sänks och indikatorn för "'toppläge" släcks. När borret arbetat sig genom arbetsstycket tänds indikatorn för "bottenläge".
- Ge slutligen styrordet 00000000_2 för att ställa passiva styrsignaler till borrmaskinen. Notera hur borret höjs och att borrmotorn stannar men även hur en markering för ett borrat hål finns på arbetsstycket.

Slut Uppgift 72

Uppgift 73

För att vrida arbetsstycket krävs en positiv flank på b_0 i styrordet till borrmaskinen. Utnyttja samma testförfarande och testprogram som i förra uppgiften och starta simulatorn med Run. Ge styrorden 00_{16} respektive 01_{16} genom att upprepade gånger klicka på b_0 på omkopplaren . På så sätt ger du ett antal stegpulser till stegmotorn.

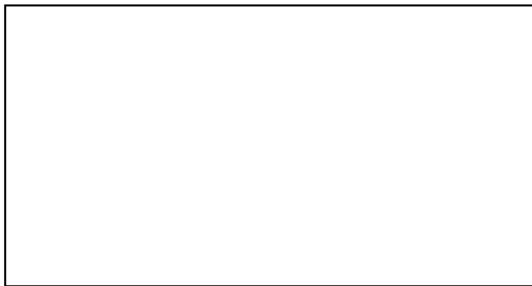


b_1=1, medurs vridning vid stegpuls
b_0, stegpuls för transition 0⟶1

Vrid nu arbetsstycket ett par steg till och försök på nytt att borra ett hål genom att ge styrordssekvensen 04_{16}, $0C_{16}$ och 00_{16}. Du ska nu att få en ny markering på arbetsstycket som indikerar ett hål.

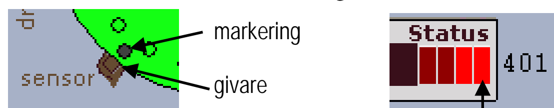
Arbetsstycket kan vridas både med- och moturs och vridningsriktningen styrs via b_1 i borrmaskinens styrord. Ge styrordet 02_{16} , dvs ettställ b_1 och klicka därefter upprepade gånger på b_0 på omkopplaren för att generera ett antal stegpulser. Observera hur arbetsstycket nu roterar medurs.

Testa slutligen att vrida arbetsstycket när borret är i bottenläge. Vad händer?



Slut Uppgift 73

Avslutningsvis skall du undersöka hur man bestämmer var på arbetsstycket det första hålet skall borras. På arbetsstycket finns en markering som anger referensposition (startposition), borrmaskinen är utrustad med en sensor (givare) som känner av denna markering.



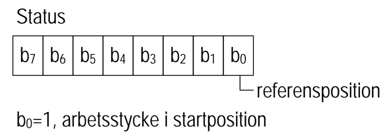
markering under sensor (i referensposition)

Välj New disc upprepade gånger och observera hur referenspositionen hamnar på olika ställen.

När arbetsstycket vrids upprepade gånger kommer markeringen förr eller senare att passera sensorn som är ansluten till b_0 i borrmaskinens statusregister.

Uppgift 74

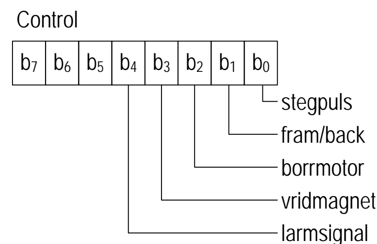
Starta åter igen testprogrammet (Run). Ge ett antal stegpulser genom att klicka på b_0 på omkopplaren och observera hur b_0 i status-registret blir 1 då arbetsstyckets referensposition är mitt för givaren.



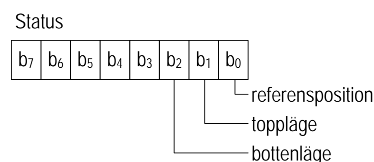
b_0=1, arbetsstycke i startposition

Slut Uppgift 74

Du har nu undersökt de flesta av borrmaskinens styr- och statussignaler. En sammanställning av borrmaskinens styrregister och statusregister visas i följande figur:



b_4=1, larm på
b_3=1, borr sänks
b_2=1, borr roterar
b_1=1, medurs vridning vid stegpuls
b_0, stegpuls för transition 0⟶1



b_2=1, borr i absolut bottenläge
b_1=1, borr i absolut toppläge
b_0=1, arbetsstycke i startposition

Den enda signalen vi hittills inte behandlat är b_4 larmsignal i styrregistret. Ett automatiskt borrprogram kan exempelvis fås att ge larm för att påkalla operatörens uppmärksamhet för olika typer av händelser.

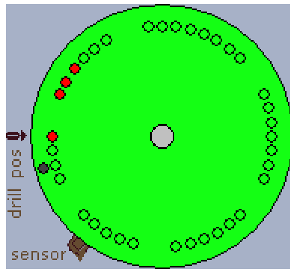
Uppgift 75

Använd samma testprogram och testförfarande som tidigare och:

- Ändra b_4 ett antal gånger och observera hur indikatorn Alarm ändras i simulatorn.

Slut Uppgift 75

Det är nu dags att konstruera ett enkelt program som utför ett automatiskt borrprogram för något arbetsstycke. Vi ska använda arbetsstycke 3 i simulatorn. Efter avslutat borrprogram ska arbetsstycket se ut på följande sätt:

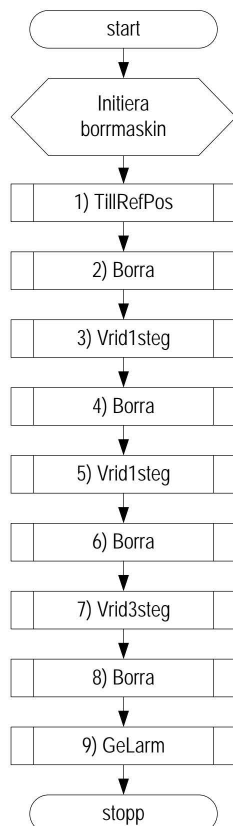


Borrschemat föreskriver här att första hålet ska borras då arbetsstycket är i referensposition, därefter ska ytterligare två hål borras med ett stegs intervall. Efter detta ska tre steg ignoreras och därefter ska det sista hålet borras. Efter avslutat borrprogram ska larmsignalen aktiveras.

För testprogrammet får vi därför följande specifikation:

1) Arbetsstycket vrids till referensposition.
2) Hål borras
3) Arbetsstycket vrids medurs ett steg
4) Hål borras
5) Arbetsstycket vrids medurs ett steg
6) Hål borras
7) Arbetsstycket vrids medurs tre steg
8) Hål borras
9) En larmsignal ges som indikation på att uppgiften är klar.

Följande flödesplan beskriver också uppgiften:



En direkt implementering av specifikationen och flödesplanen kan se ut på följande sätt:

```
; Drilltest2.s12
        USE     Labdefs.s12
        ORG     $1000
start:  LDAA    #0          ; Reset
        STAA    DrillControl
        JSR     TillRefPos
        JSR     Borra
        JSR     Vrid1steg
        JSR     Borra
        JSR     Vrid1steg
        JSR     Borra
        JSR     Vrid1steg
        JSR     Vrid1steg
        JSR     Vrid1steg
        JSR     Borra
        JSR     GeLarm
stopp:  BRA     stopp

Vrid1steg:      RTS
TillRefPos:     RTS
Borra:          RTS
GeLarm:         RTS
```

Vi har här implementerat funktionen Vrid3steg i flödesplanen som tre sekventiella anrop av funktionen Vrid1steg.

Uppgift 76

Redigera ett program i en ny källtextfil `Drilltest2.s12` enligt anvisningarna ovan, spara källtextfilen.

Observera hur de nödvändiga subrutinerna implementerats som ”dummy”-rutiner. Detta gör vi för att definiera respektive symbol och kunna assemblera programmet. Vi ska sedan, efter hand, färdigställa subrutinerna.

Assemblera programmet, rätta eventuella fel.

Slut Uppgift 76

För att färdigställa det automatiska borrprogrammet måste nu även följande subrutiner implementeras.

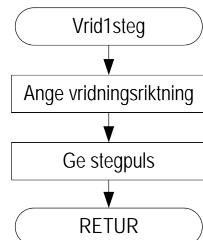
- `Vrid1steg`
- `TillRefPos`
- `Borra`
- `GeLarm`

Uppgift 77

Du ska nu implementera subrutinen Vrid1steg

Arbetsstycket fås att vridas ett steg då vi ger styrsignalsekvensen:

1. Ange vridningsvinkel
2. Ge stegpuls



Implementera enligt flödesplanen i filen `Drilltest2.s12`.

Testa funktionen med simulatorns Run kommando.

Slut Uppgift 77

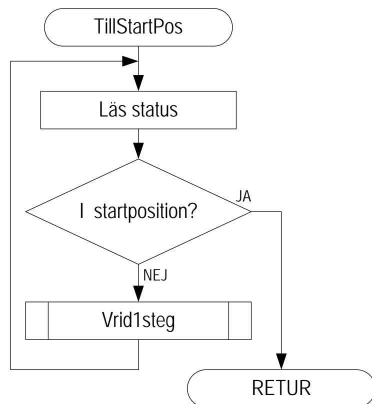
Låt oss nu specificera subrutinen TillRefPos som används för att ställa arbetsstycket i referenspositionen. Vi såg tidigare att b_0 i statusregistret anger om arbetsstycket är positionerat i referensposition eller inte.

Att söka upp referenspositionen på arbetsstycket, kan alltså göras genom att följa denna algoritm:

1. Undersök om indikatorn för referenspositionen är satt (om $b_0 = 1$)
2. Om $b_0 = 1$ avsluta subrutinen
3. Utför subrutin Vrid1steg
4. Upprepa från 1)

Uppgift 78

Redigera i filen `Drilltest2.s12`, rutinen TillStartPos enligt följande flödesplan:



- Assemblera, rätta eventuella fel och ladda till simulatorn.
- Testa subrutinen med simulatorns Run kommando.

När du testar denna rutin kan det vara lämpligt att sätta en brytpunkt på instruktionen där programmet läser status (se flödesplanen) för att därefter undersöka om den följande villkorliga BRANCH-instruktionen fungerar som avsett.

Slut Uppgift 78

Vi tar oss nu an algoritmen för att borra ett hål. Följande styrsignalsekvens måste åstadkommas:

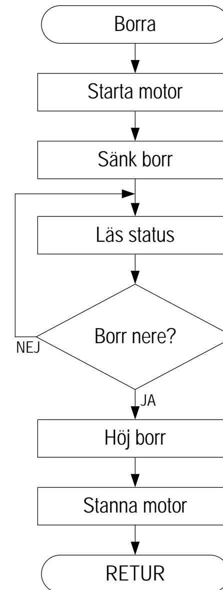
- Starta borrmotor
- Sänk borr
- Invänta borr nere
- Höj borr
- Stanna borrmotor.

Observera att algoritmen inte är optimal eftersom vi vill borra flera hål i ett arbetsstycke. Det är därför onödigt att stanna borrmotorn efter varje borrat hål.

Vi nöjer oss dock för tillfället med detta enkla angrepssätt.

Uppgift 79

Redigera i filen `Drilltest2.s12`, rutinen Borra enligt följande flödesplan.



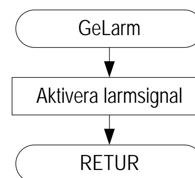
Assemblera, ladda till simulator och testa funktionen. Rätta eventuella fel. Testa med simulatorns Run kommando.

Slut Uppgift 79

Larmsignalen används här för att indikera att hela programmet utförts och implementeringen tillhör de enklare uppgifterna.

Uppgift 80

Redigera i filen `Drilltest2.s12`, rutinen GeLarm enligt följande flödesplan.



Assemblera och rätta eventuella fel.

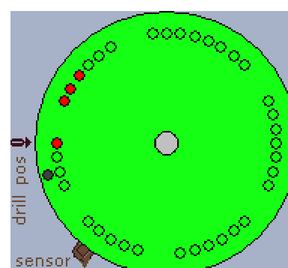
Slut Uppgift 80

Uppgift 81

Utför nu en avslutande test av programmet. Tänk på att använda arbetsstycke 3 (New Disc).

Testa med simulatorns Run kommando.

Efter avslutat borrprogram ska arbetsstycket se ut så här:



Slut Uppgift 81

Du ska nu ha skrivit ett program som borrar efter ett förutbestämt borrschema. För att ge styrord till borrmaskinen har du förmodligen använt kodsekvenser liknande:

```
*  Ge Styrord till borrmaskinen
    LDAA    #Styrord       ;Funktion
    STAA    DrillControl   ;Skriv styrord
```

eller kanske

```
    BSET    #StyrOrd,DrillControl
```

Sådana sekvenser fungerar dock bara om programmet alltid följer samma flödesplan, och därmed använder en förutbestämd sekvens av styrord som kontrollerar borrmaskinen.

För vår kommande applikation, där en operatör kan ge kommandon för att styra borrmaskinen, måste vi konstruera ett generellt program.

Eftersom olika sekvenser av styrsignaler ska kunna ges till borrmaskinen måste vi ha kontroll över vilka styrsignaler som är aktiva respektive passiva till borrmaskinen när nästa kommando ges. Vanligtvis behöver man bara ändra en bit i taget i borrmaskinens styrregister för någon given funktion, samtidigt måste dock övriga bitar måste vara opåverkade.

Problemet kan lösas genom att vi manipulerar enstaka bitar i styrordet till borrmaskinen. Det kan vi åstadkomma med hjälp av AND- och OR-instruktioner (för att nollställa respektive ettställa bitar i borrmaskinens styrregister) och vi kan också använda speciella bit-manipuleringsinstruktioner som BSET (Bit SET) respektive BCLR (BitCLeaR).

Betrakta nu följande kodsekvenser,
för att nollställa en bit används:

```
* Läs nuvarande styrord
    LDAA    DrillControl
* Nollställ lämplig bit
    ANDA    #xx
* Skriv nytt styrord
    STAA    DrillControl
```

sekvensen är funktionellt likvärdig med:

```
    BCLR    #~xx,DrillControl
```

för att ettställa en bit används i stället:

```
* Läs nuvarande styrord
    LDAA    DrillControl
* Ettställ lämplig bit
    ORAA    #xx
* Skriv nytt styrord
    STAA    DrillControl
```

denna sekvens är funktionellt likvärdig med:

```
    BSET    #xx,DrillControl
```

Ingen av kodsekvenserna ovan fungerar dock i vårt fall eftersom styrregistret `DrillControl` är en utport. Vid läsningen av styrregistret kan vi inte räkna med att läsa samma värde som det som senast skrevs till utporten. (Det finns även läsbara utportar och man måste alltid, från fall till fall utreda vilken typ man har att göra med). Utporten på adress 400_{16} (`DrillControl`) är av typen "write only" och då vi försöker läsa från den med instruktionen (`LDAA DrillControl`) får vi ett nonsensresultat.

Det krävs därför att vi använder en variabel som hela tiden innehåller en kopia av det senaste styrordet. Ibland kallas denna metod för att använda ”skuggregister”.

Ge variabeln symbolnamnet DCShadow enligt:

```
DCShadow   RMB  1  ;DrillControl shadow
```

Följande kodsekvenser visar exempel på hur vi kan ändra enstaka bitar i borrmaskinens styrregister utan att samtidigt ofrivilligt ändra övriga bitars inställningar.

Låt `xx` beteckna ett bitmönster med ettor för de bitar som ska påverkas och nollor för de bitar som ska lämnas opåverkade. För att nollställa bitar används nu:

```
    LDAA        DCShadow
    ANDA        #~xx
    STAA        DCShadow
    STAA        DrillControl
```

för att ettställa en bit används:

```
    LDAA        DCShadow
    ORAA        #xx
    STAA        DCShadow
    STAA        DrillControl
```

Observera hur vi använder det inverterade bitmönstret då vi vill nollställa bitar.

Du ska nu isolera dessa kodsekvenser i två subrutiner, `OUTONE` respektive `OUTZERO`. Dessa rutiner är centrala och kommer att användas av åtskilliga subrutiner framöver. Det är därför viktigt att du noggrant kontrollerar funktionen då du testar dessa subrutiner.

Uppgift 82

Subrutinen Outzero sköter utmatningen av styrsignaler (styrordet) till borrmaskinen Endast en av styrsignalerna kan nollställas åt gången. En kopia av styrordet lagras i vaiabeln `DCShadow`. Subrutinen Outzero specificeras av följande:

```
; Subrutin Outzero.
; Läser kopian av borrmaskinens styrord
; på adress 'DCShadow'. Nollställer en
; av bitarna och skriver det nya
; styrordet tillbaka till kopian
; 'DCShadow' samt till utporten
; 'DrillControl'.
; Biten som nollställs ges av innehållet
; i B-registret (0-7) vid anrop.
; Om (B) > 7 utförs ingenting.
;
; Anrop:       LDAB    #bitnummer
;              JSR     Outzero
;
; Bitnumrering framgår av följande:
```

b7	b6	b5	b4	b3	b2	b1	b0
7	6	5	4	3	2	1	0

```
; Utdata:      Inga
; Register:    Ingen
; Anropar:     Inga
```

Skapa en ny källtextfil `Subroutines.s12` med subrutinen Outzero. I nästa uppgift ska du testa subrutinen.

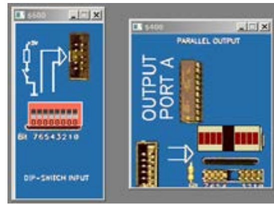
Slut Uppgift 82

Uppgift 83

Det är dags att testa Outzero.

Först skriver du ett litet huvudprogram som anropar din subrutin. Du måste initiera `DCShadow` innan du gör subrutinanropet. Huvudprogrammet beskrivs utförligt nedan.

För att testa subrutinen är det lämpligt att använda IO-simulatorerna Dip-Switch Input och Parallel Output.




```
;
; Subroutines.s12
        USE     Labdefs.s12

        ORG     $1000
; Startvärde till DCShadow
start:  LDAB    #$FF
        STAB    DCShadow

; Läs bit som skall nollställas
Loop:   LDAB    InPort
; .. och nollställ
        JSR     Outzero
; Läs kopian och ...
        LDAA    DCShadow
; .. visa denna på lysdioderna
        STAA    OutPort
; Börja om
        JMP     Start

;
Outzero:
; Implementering av Outzero
        ...
        RTS
```

Ställ in värden 0 t,o,m 7 på omkopplaren för att ange vilken bit du vill nollställa med Outzero. Läs av resultatet på lysdioderna.

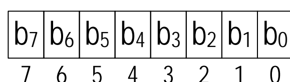
Testa och rätta eventuella fel i Outzero så att subrutinen fungerar som den ska.

Slut Uppgift 83

Uppgift 84

Subrutinen Outone sköter utmatningen av styrsignaler (styrordet) till borrmaskinen, då en av styrsignalerna skall ettställas. Outone specificeras på följande sätt:

```
; Subrutin Outone.
; Läser kopian av borrmaskinens styrord
; på adress 'DCShadow'. Ettställer en
; av bitarna och skriver det nya
; styrordet till 'DCShadow' samt
; utporten 'DrillControl'.
; Biten som nollställs ges av innehållet
; i B-registret (0-7) vid anrop.
; Om (B) > 7 utförs ingenting.
; Anrop:        LDAB    #bitnummer
;               JSR     OUTONE
;
; Bitnumrering framgår av följande:
```




```
; Utdata:       Inga
; Register:     Ingen
; Anropar:      Inga
```

Implementera rutinen Outone i filen `Subroutines.s12`. Du kan testa Outone med nästan samma metod som då du testade Outzero. Med skillnaden att nu börja med att nollställa `DCShadow`. Testa Outone och övertyga dig om att den fungerar enligt specifikationen.

Slut Uppgift 84

När subrutinerna Outone och Outzero fungerar som de ska är det dags att ta bort testsekvenserna, endast själva subrutinerna ska vara kvar i filen `Subroutines.s12`.

Realtid och fördröjningsrutiner

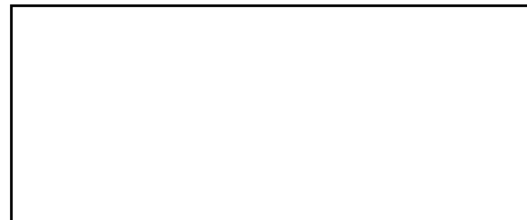
I detta avsnitt ska vi titta närmre på olika realtidsegenskaper hos vår borrmaskin. Vi ska konstruera så kallade fördröjningsrutiner för synkronisering mellan dator och borrmaskin.

Du har sett hur du kan köra simulatorn på tre olika sätt, nämligen:

- Step, där du själv bestämmer när nästa instruktion skall utföras
- Run, långsam kontinuerlig exekvering (c:a 10 instruktioner/sekund)
- Run Fast, snabb kontinuerlig exekvering (c:a 1000 instruktioner/sekund)

Uppgift 85

Testa på nytt din lösning från Uppgift 81. Välj Run Fast. Vad händer?



Slut Uppgift 85

Borrmaskinssimulatorn tar c:a 250 ms att utföra en vridning. Simulatorn utför c:a 10 instruktioner/sek. vid Run och betydligt fler vid Run Fast. Vid Run Fast kommer därför pulserna till stegmotorn med så pass korta mellanrum att den inte hinner att vrida sig (och arbetsstycket). Det krävs alltså fördröjningar i vårt program för att anpassa arbetstakten hos den snabba processorsimulatorn till den långsamma borrmaskinsimulatorn.

Flera operationer med borrmaskinen kräver någon fördröjning innan man kan utföra nästa moment, exempelvis:

- Starta borrmotorn (vänta tills den är uppe i varv)
- Vrid arbetsstycket ett steg (vänta tills det har vridits till rätt position)

Inför arbetet med att skriva det riktiga programmet för borrmaskinen så måste vi beakta de realtidsfördröjningar som kan komma att krävas.

I avsnitt 1 Uppgift 36, konstruerade du en fördröjningsrutin Delay, som det nu är dags att återanvända.

För att kunna använda samma fördröjningsrutin i olika exekveringsmiljöer (Run respektive RunFast) är det lämpligt att ange startvärdet i fördröjningssekvensen under villkorlig assemblering:

```
#ifdef  RUNFAST
DelayConst  EQU xx
#else
DelayConst  EQU yy
#endif
```

Uppgift 86

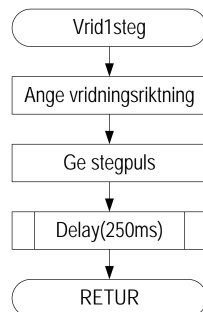
Komplettera din Delay funktion i filen `Delay.s12` med villkorlig assemblering.

Repetera eventuellt Uppgift 36 och bestäm konstanten för Run så att fördröjningarna blir ungefär lika långa oavsett om du använder Run eller RunFast.

Slut Uppgift 86

Uppgift 87

Komplettera den befintliga subrutinen Vrid1steg med anrop av fördröjningsrutinen Delay enligt följande flödesplan:



Redigera en ny fil `Drilltest3.s12` enligt följande:

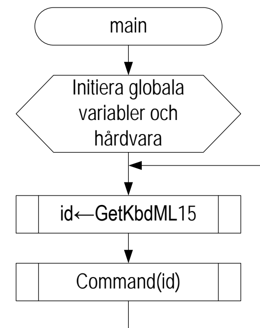
```
; Drilltest3.s12
        USE     Labdefs.s12
        ORG     $1000
start:
        JSR     Vrid1steg
        BRA     start

        USE     Delay.s12

Vrid1steg:
        ...
        RTS
```

Testa subrutinen för både Run och RunFast, verifiera att arbetsstycket vrider sig i båda fallen.

Slut Uppgift 87

Operatörsstyrd borrmaskin

Borrmaskinens operationer styrs från ett tangentbord av typen ML2/ML15. Följande flödesplan visar huvudprogrammets struktur. När systemet väl startats, kommer det kontinuerligt att bestämma och utföra operatörens kommando.



Initiera:

Här ges initialvärden till de globala variabler som används av applikationen. Vi har sedan tidigare variabeln `DCShadow` och ytterligare eventuella variabler ska också ges sina respektive initialvärden här.

Här ska också all hårdvara försättas i väldefinierat tillstånd. För vår borrmaskin innebär detta att den måste ges passiva styrsignaler för alla funktioner till styrregistret.

Kontinuerlig funktion

Operatörens instruktion hämtas som en tangentkod från tangentbordet. Vi använder här den tidigare, i avsnitt 2, konstruerade tangentbordsrutinen. Returvärdet från denna kallar vi symboliskt för ”id” i flödesplanen ovan.

Vi fortsätter med att skicka ”id” som en parameter till subrutinen Command, som är den egentliga arbetshästen i programmet.

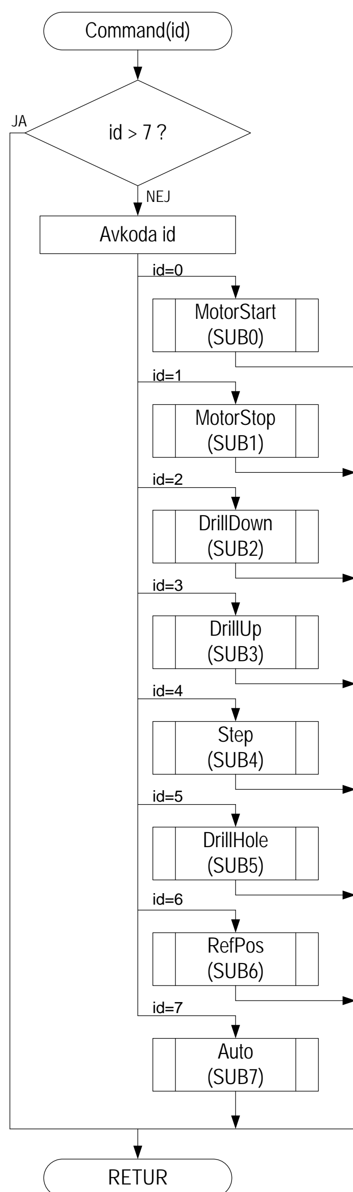
Själva huvudprogrammet (main) och rutinen Command redigeras i en källtextfil med följande struktur:

```
;   Main.s12
;   Operatörsstyrd borrautomat

;   Definitioner
    USE     Labdefs.s12

    ORG     $1000
main:
    ---     Initiera borrmaskin
;
; Huvudprogram, invänta vald operation
main_loop:
    JSR     GetKbdML15
* Tangentkod nu i register B...
* Utför vald operation
    JSR     Command
    BRA     main_loop
;-----------------------------------
    ---     Här följer rutinen Command
    ---     med pekartabell
    ---     och Dummy-subrutiner
;-----------------------------------
;   Placera USE-direktiv här
    USE     ML15drvr.s12
    ---
; Placera alla globala variabler här
DCShadow:  RMB     1
```

Subrutinen Command

De olika operationer som kan utföras, respektive tangentkod och deras respektive implementering (subrutin) ges i följande tabell:

tangent kod	Operation	subrutin
0	starta borrmotorn	MotorStart
1	stoppa borrmotorn	MotorStop
2	sänk borret	DrillDown
3	höj borret	DrillUp
4	rotera arbetsstycket medurs ett steg	Step
5	borra ett hål	DrillHole
6	stega arbetsstycket till referensposition	RefPos
7	borra hål längs cirkeln enligt mönster	DoAuto



Flödesplan för subrutinen Command

I flödesplanen för subrutinen Command, ser vi att parametern (id) först kontrolleras så att den representerar ett tillåtet kommando.

Efter detta följer en flervalskonstruktion. Den kan implementeras med upprepade jämförelse/test operationer men en sådan implementering blir snabbt svårhanterlig då antalet fall (olika val) ökar. Vi ska här därför ge exempel på en effektiv implementering baserad på användande av en tabell innehållande pekare till subrutinerna som ska kunna anropas.

Vi börjar med att konstruera själva flervals-konstruktionen i Command. De olika sub-rutinerna som ska finnas i tabellen representerar var sin operation som operatören kan utföra med borrmaskinen. Eftersom de olika subrutinerna för styrning av borrmaskinen ännu ej är slutgiltigt implementerade inför vi för teständamål ”dummy-subrutiner” kallade `SUB0`, `SUB1` osv. Subrutinnamnen `SUB0` och `SUB1` i hopptabellen byter vi senare ut mot de riktiga rutinerna MotorStart, MotorStop etc, allt eftersom dessa implementeras.

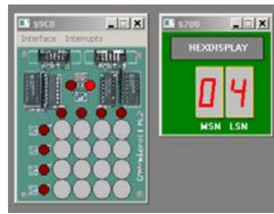
```
;------------------------------------------
; SUBRUTIN - Command
; Beskrivning: Rutinen avgör vilken
; kommandosubrutin som skall
; utföras och anropar denna.
; Anrop:   JSR Command
; Indata:  Kommandonummer i reg B
; Utdata:  Inga
; Register:B,X ändras
; Anrop:   SUB0 ... SUB7
;------------------------------------------

Command:
; giltigt värde?
           CMPB    #7
           BHI     CommandExit
; pekartabellens basadress
           LDX     #JUMPTAB

; offset är 2 bytes per adress
           ASLB
; hämta subrutinens startadress
           LDX     B,X
; utför subrutin
           JSR     ,X
; återvänd från kommandorutin
CommandExit:
           RTS

;-----------------------------------------
;   Tabell med subrutinadresser (pekare)
JUMPTAB     FDB     SUB0,SUB1,SUB2,SUB3
            FDB     SUB4,SUB5,SUB6,SUB7
;-----------------------------------------
; subrutiner för test

SUB0     MOVB    #0,OutPort
         RTS
SUB1     MOVB    #1,OutPort
         RTS
SUB2     MOVB    #2,OutPort
         RTS
SUB3     MOVB    #3,OutPort
         RTS
SUB4     MOVB    #4,OutPort
         RTS
SUB5     MOVB    #5,OutPort
         RTS
SUB6     MOVB    #6,OutPort
         RTS
SUB7     MOVB    #7,OutPort
         RTS
```

Uppgift 88

Redigera en fil med namnet `Main.s12` enligt de tidigare anvisningarna.

Assemblera filen `Main.s12`. Rätta eventuella fel och ladda denna sedan till simulatorn.

För test är det lämpligt att använda HexDisplay som utmatningsenhet. Koppla därför de nödvändiga IO-simulatorerna HexDisplay till adress 700_{16} och ML2 Keyboard till adress $9C0_{16}$.



Starta huvudprogrammet genom att välja RunFast i simulatorn. När du nu trycker ner (och släpper upp) tangenter på ML2 ska du kunna utläsa tangentens nummer på displayenheten.

Prova olika kombinationer av kommandon och kontrollera att ditt huvudprogram med tillhörande Command rutin fungerar korrekt.

Slut Uppgift 88

Rekapitulera nu huvudprogrammet. Initialt måste borrmaskinen initieras så att vi vet att förutsättningarna för att börja ge kommandon är uppfyllda. Detta görs här genom att ställa ut passiva styrsignaler för borrmaskinens samtliga funktioner, exempelvis enligt följande:

```
; Sätt passiva styrsignaler till
; borrmaskinen ...
    MOVB    #$00,DrillControl
; ... och kopian
    MOVB    #$00,DCShadow
```

Subrutiner till borrmaskinen

Vi börjar nu med de subrutiner som är nödvändiga för att kunna styra borrmaskinen. Vi kommer att använda de båda subrutinerna Outzero och Outone som vi redan implementerat. Vi börjar med rutinerna MotorStart och MotorStop. Detta är alltså de två första rutinerna som kan anropas från Command.

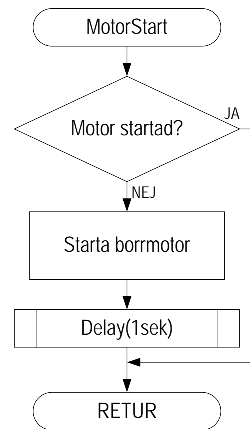
Uppgift 89

Rutinen MotorStart används för att starta borrmotorn då tangent med kod 0 tryckts ned på tangentbordet och specificeras enligt:

```
; SUBRUTIN MotorStart.
; Subrutinen startar borrmotorn och
; väntar därefter i 1 sekund för att
; borret skall uppnå rätt hastighet
;
; Anrop:    JSR     MotorStart
; Indata:   Inga
; Utdata:   Inga
; Register: Ingen
; Anropad:  Outone, Delay
```

Efter att biten borrmotor i styrregistret getts värdet 1 tar det c:a 1 sekund innan borren är uppe i fullt varvtal. Borren får inte sänkas innan fullt varvtal uppnåtts eftersom detta kan skada borrmotorn.

Om motorn redan startats ska ingen fördröjning utföras. Det finns ingen statusbit som indikerar om motorn är i gång eller inte så det är därför lämpligt att använda `DCShadow` för att kontrollera om så är fallet, se följande flödesplan:



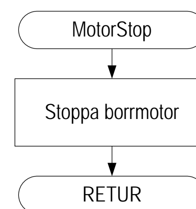
Skriv subrutinen MotorStart enligt ovanstående specifikation. Lägg till den i `Subroutines.s12`

Slut Uppgift 89

Uppgift 90

Subrutinen MotorStop som stannar borrmaskinens motor, specificeras enligt följande:

```
; SUBRUTIN MotorStop.
; Subrutinen stoppar borrmotorn.
;
; Anrop:    JSR     MotorStop
; Indata:   Inga
; Utdata:   Inga
; Register: Ingen
; Anropade: Outzero
```



Skriv subrutinen MotorStart enligt specifikationen. Lägg till den i `Subroutines.s12`

Slut Uppgift 90

Det är nu dags att bygga vidare på vår applikation med att testa rutinerna MotorStart och MotorStop.

Uppgift 91

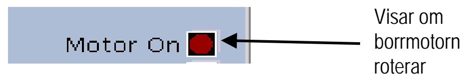
- I slutet av `Main.s12` lägger du nu till ytterligare direktiv för att inkludera de nya subrutinerna.

```
USE Delay.s12
USE Subroutines.s12
```

- Korrigera hopptabellen i Command genom att ändra `SUB0` till `MotorStart` och `SUB1` till `MotorStop` enligt följande. Du kan därefter ta bort dummy-rutinerna `SUB0` och `SUB1`.

```
;   Tabell med subrutinadresser
JUMPTAB     FDB     MotorStart,MotorStop
            FDB     SUB2,SUB3
            FDB     SUB4,SUB5,SUB6,SUB7
```

- Assemblera filen `Main.s12`. Rätta eventuella fel och ladda till simulatorn för test. Tänk också på att definiera `RUNFAST` så att fördröjnings-rutinen assembleras med rätt fördröjnings-konstant.
- Anslut IO-simulatorns ML2 Keyboard till adress $9C0_{16}$ och Drill till adress 400_{16}.
- Starta RunFast och kontrollera att borrmotorn kan startas och stoppas från tangentbordet.



Visar om borrmotorn roterar

Spara den fungerande lösningen.

Slut Uppgift 91

Rutinerna DrillDown och DrillUp är enkla och beskrivs av följande:

```
; SUBRUTIN DrillDown
; Rutinen sänker borret genom att
; aktivera drivenheten för vridmagneten
;
; Anrop:    JSR     DrillDown
; Indata:   Inga
; Utdata:   Inga
; Register:Ingen
; Anropade:Outone
```

```
; SUBRUTIN DrillUp
; Rutinen höjer borret genom att
; deaktivera vridmagneten
;
; Anrop:    JSR     DrillUp
; Indata:   Inga
; Utdata:   Inga
; Register:Ingen
; Anropade:Delay,Outzero
```

Uppgift 92

- Skriv subrutinerna DrillDown och DrillUp enligt specifikationen. Lägg till dem i `Subroutines.s12`
- Korrigera hopptabellen i Command genom att ändra `SUB2` till `DrillDown` och `SUB3` till `DrillUp` enligt följande. Du kan därefter ta bort dummy-rutinerna `SUB2` och `SUB3`.

```
;   Tabell med subrutinadresser
JUMPTAB     FDB     MotorStart,MotorSTOP
            FDB     DrillDown,DrillUp
            FDB     SUB4,SUB5,SUB6,SUB7
```

Övertyga dig om att subrutinerna fungerar som de ska.

Slut Uppgift 92

Uppgift 93

Du skall nu kunna starta, sänka borret och detektera en genomborrning av arbetsstycket, höja borret och slutligen stanna borrmotorn med kommandon från tangentbordet.

Då du borrat ett hål i arbetsstycket så kan du välja New Disc för att välja ett nytt, oborrat, arbetsstycke.

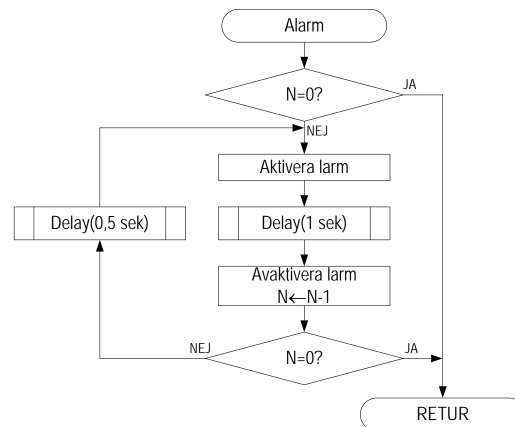
Prova nu på nytt samtliga funktioner genom att borra ett hål i arbetsstycket genom att ge kommandon från tangentbordet.

Slut Uppgift 93

Uppgift 94

Subrutinen Alarm används för att implementera en funktion som påkallar uppmärksamhet av operatören. Funktionen kan ge variabelt antal larmsignaler som ljuder i en sekund med en halv sekunds mellanrum. Den specificeras på följande sätt:

```
; SUBRUTIN Alarm
; ger N larmsignaler med
; längden 1 s och med 0,5 s mellanrum.
;
; Anrop:    LDAB    #N
;           JSR     Alarm
; Indata:   Antal larmsignaler, N,
;           i B-registret.
; Utdata:   Inga
; Register:A,B kan ändras
; Anropade:Delay,Outzero,Outone
```



Implementera rutinen Alarm enligt specifikation. Lägg till dem i `Subroutines.s12`.

Du testar Alarm i samband med nästa uppgift.

Slut Uppgift 94

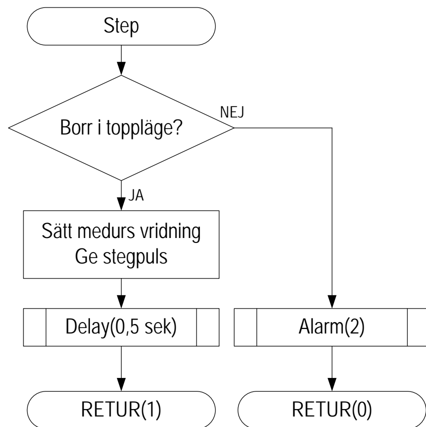
Uppgift 95

Subrutinen Step anropas för att vrida arbetsstycket en position.

Innan arbetsstycket vrids ska programmet kontrollera att borren står i toppläget, om så inte är fallet ska Step ge två larmsignaler och därefter avslutas.

Efter att ha gett stegpulsen måste en fördröjning om c:a 0,5 sekunder göras för att arbetsstycket ska hinna stabiliseras i det nya läget.

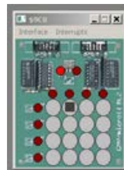
```
; SUBRUTIN Step
; Rutinen kontrollerar först att
; borret är i "toppläge", vrider sedan
; arbetsstycket ett steg medurs.
; Om borret är sänkt vid anrop av Step
; utförs ingen vridning utan i stället
; ges tre larmsignaler via Alarm innan
; rutinen avslutas.
;
; Anrop:    JSR     Step
; Indata:   Inga
; Utdata:   Register B innehåller 1 om
;           arbetsstycket vridits, annars
;           innehåller B värdet 0
; Register:A,B kan ändras
; Anropade:Alarm,Delay,Outzero,Outone
```



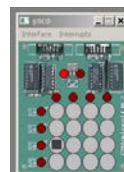
Observera att kontrollen av om borret är sänkt måste göras genom att undersöka värdet av givaren borr uppe. Man kan inte använda givaren för borr nere vid denna kontroll eftersom en borrning kan pågå utan att borret har hunnit igenom arbetsstycket. I ett sådant fall har vi alltså varken indikation på borr nere eller borr uppe.

- Implementera rutinen Step enligt specifikation. Lägg till den i `Subroutines.s12`.
- Korrigera hopptabellen i Command genom att ändra `SUB4` till `Step` enligt följande. Du kan därefter ta bort dummy-rutinen `SUB4`.

```
;   Tabell med subrutinadresser
JUMPTAB     FDB     MotorStart,MotorSTOP
            FDB     DrillDown,DrillUp
            FDB     Step,SUB5,SUB6,SUB7
```



Testa nu Step genom att starta programmet (RunFast) och därefter använda tangentbordet. Kontrollera först att Step fungerar då borren är i toppläget genom att klicka på tangent med koden 4.



Använd sedan DrillDown (tangent med kod 2) för att få borren att ligga an mot arbetsstycket. Försök nu på nytt att vrida arbetsstycket genom att klicka på tangent med kod 4. Kontrollera såväl felhanteringen i Step som subrutinen Alarm.

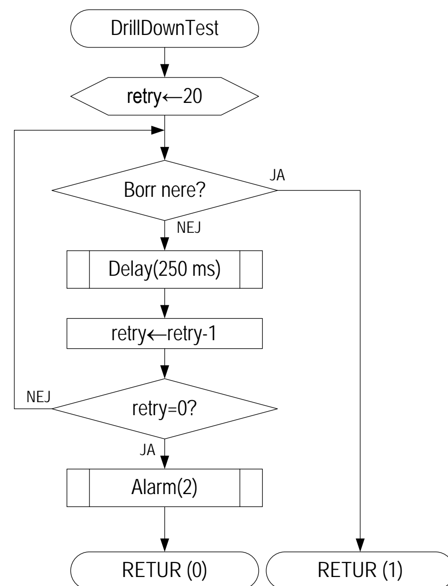
Slut Uppgift 95

Uppgift 96

För att ett hål skall kunna borras i en sammansatt operation behövs också en test som avgör om borret har arbetat sig igenom hela arbetsstycket eller inte. Subrutinen DrillDownTest skall sköta detta.

```
; SUBRUTIN DrillDownTest
; Väntar maximalt 5 sekunder på att
; borret nått sitt bottenläge.
; Indikatorn ska läsas av fyra gånger
; per sekund.
; Om borret ej har nått bottenläget inom
; denna tid ges två larmsignaler och
; därefter avslutas rutinen
;
; Anrop:    JSR DrillDownTest
; Indata:   Inga
; Utdata:   Register B innehåller 0 om
;           larm gavs, B innehåller
;           annars 1
; Register:A och B kan ändras
; Anropade:Alarm,Delay
```

Följande flödesplan ger förslag på hur DrillDownTest bör implementeras:



Indikatorn undersöks med 250 ms intervall, och "timeout" är 5 sekunder, alltså ska maximalt 20 försök göras innan larm ges.

- Implementera rutinen DrillDownTest enligt specifikation. Lägg till den i `Subroutines.s12`.

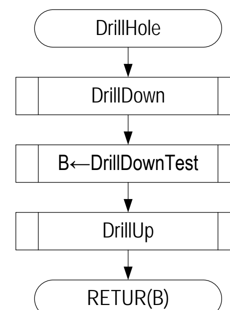
Assemblera och rätta eventuella fel i källtexten, funktionen hos DrillDownTest ska vi testa efter nästa uppgift.

Slut Uppgift 96

Vi ska nu realisera funktionen "borra ett hål" för tangent med kod 5.

Uppgift 97

Rutinen som i en sammansatt operation borrar ett hål i arbetsstycket kallas DrillHole och specificeras enligt följande flödesplan.



Notera att funktionen förutsätter att borrmotorn har startats. Om så inte är fallet ska larm ges från DrillDownTest. Notera också att returvärdet från DrillHole är det samma som för DrillDownTest.

- Implementera rutinen DrillHole enligt specifikation. Lägg till den i `Subroutines.s12`.
- Korrigera hopptabellen i Command genom att ändra `SUB5` till `DrillHole` enligt följande. Du kan därefter ta bort dummy-rutinen `SUB5`.

```
;   Tabell med subrutinadresser
JUMPTAB     FDB     MotorStart,MotorSTOP
            FDB     DrillDown,DrillUp
            FDB     Step,DrillHole
            FDB     SUB6,SUB7
```

Slut Uppgift 97

Det kan vara dags att påminna om specifikationen av vår borr-robot, så här ser tabellen med kommandon och motsvarande funktioner ut, de funktioner vi ännu inte implementerat är kursiva och nedtonade här.

tangent kod	Operation	subrutin
0	starta borrmotorn	MotorStart
1	stoppa borrmotorn	MotorStop
2	sänk borret	DrillDown
3	höj borret	DrillUp
4	rotera arbetsstycket medurs ett steg	Step
5	borra ett hål	DrillHole
6	*stega arbetsstycket till referensposition*	*RefPos*
7	*borra hål längs cirkeln enligt mönster*	*DoAuto*

Uppgift 98

Du ska nu testa funktionen DrillHole och subrutinen DrillDownTest. Detta omfattar två testfall, du måste dels kontrollera att funktionen är korrekt under de givna förutsättningarna, men också att programrutinerna beter sig korrekt även om inte förutsättningarna är uppfyllda.

För att kontrollera korrektheten under givna förutsättningar, dvs. borrmotorn har startats innan DrillHole anropas:

Starta programmet (RunFast) och ge följande operatörskommandon från tangentbordet:

1. Starta borrmotor (tangent 0)
2. Borra ett hål (tangent 5)
3. Stoppa borrmotor (tangent 1)

Kontrollera att du har en hålmarkering och att inga larm gavs under operationen.

4. Försök nu att borra ett hål (tangent 5) utan att borrmotorn startats. Eftersom det redan finns ett hål här ska även detta hanteras utan felindikering.
5. Rotera arbetsstycket genom att ge tangentkod 4.
6. Försök nu på nytt att borra ett hål (tangent 5) utan att borrmotorn startats. Denna operation ska resultera i fel och ge larm.

Rätta eventuella felaktigheter och upprepa testförfarandet tills subrutinerna fungerar som de ska.

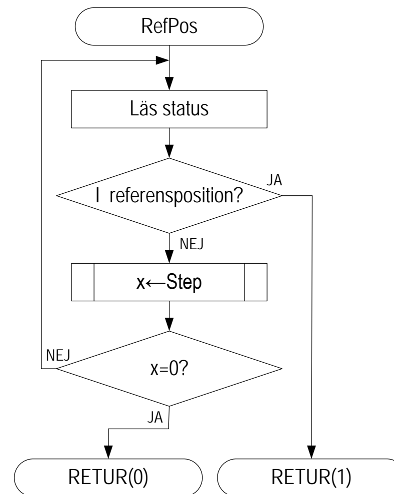
Slut Uppgift 98

Arbetsstycket skall kunna roteras till referenspositionen genom anrop av subrutinen RefPos.

Detta innebär att vrida arbetsstycket ett steg i taget tills givaren för referenspositionen aktiveras. Vi måste dock vara observanta på att det kan bli fel i subrutinen Step, om borren inte befinner sig i toppläget. Alltså måste returvärdet från Step kontrolleras och om ingen vridning har utförts så ska också RefPos avbrytas även om givaren inte indikerar referensposition.

Uppgift 99

Följande flödesplan och specifikation visar hur RefPos ska implementeras:




```
; SUBRUTIN RefPos
; Roterar arbetsstycke tills det
; befinner sig i referensposition eller
; avbryter vid feltillstånd
; Anrop:    JSR RefPos
; Indata:   Inga
; Utdata:   B är 0 om feltillstånd
;           B är 1 annars
; Register:A och B kan ändras
; Anropade:Step
```

- Implementera rutinen RefPos enligt specifikation. Lägg till den i `Subroutines.s12`.
- Korrigera hopptabellen i Command genom att ändra `SUB6` till `RefPos` enligt följande. Du kan därefter ta bort dummy-rutinen `SUB6`.

```
;   Tabell med subrutinadresser
JUMPTAB     FDB     MotorStart,MotorSTOP
            FDB     DrillDown,DrillUp
            FDB     Step,DrillHole
            FDB     RefPos,SUB7
```

Testa slutligen RefPos, kontrollera såväl korrekt beteende, dvs. då Step kan utföras utan fel, som upptäckt av feltillstånd i Step.

Slut Uppgift 99

Vi har nu skapat praktiskt taget allt vi behöver för att uppfylla den ursprungliga specifikationen av vår "borr-automat". Det är dags att ge sig i kast med den avslutande uppgiften där du implementerar denna robot. Hålmönstret för ett arbetsstycke (ett varv) specificeras i en tabell:

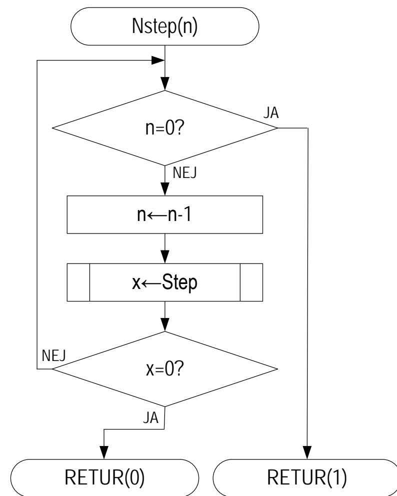
hål	1	2	3	4	5	6	7	8	9	10
antal steg	0	1	1	1	1	1	1	1	2	1
11	12	13	14	15	16	17	18	19	20	21
5	2	2	2	2	4	4	3	8	2	FF

Tabell för hålmönster

I denna tabell anges antalet steg till nästa hål. Talet FF_{16} används för att markera mönsteravslut. När arbetsstycket befinner sig i referenspositionen eller när ett hål har borrats i någon position skall arbetsstycket kunna roteras ett givet antal steg till positionen för nästa hål.

Uppgift 100

Konstruera en subrutin Nstep som utför ett bestämt antal steg. Nstep ska använda subrutinen Step och vi får, precis som i RefPos, kontrollera returvärdet och avbryta vid eventuellt fel i Step. Antalet steg som ska utföras anges i flödesdiagrammet av n.




```
; SUBRUTIN Nstep.
; Roterar arbetsstycket n steg medurs.
;
; Anrop:   LDAB    #n
;          JSR     Nstep
; Indata:  Antalet steg n (0-255)
;          i B-registret.
; Utdata:  B är 0 om feltillstånd
;          B är 1 annars
; Register:A och B kan ändras
; Anropade:Step
```

- Implementera rutinen Nstep enligt specifikation. Lägg till den i `Subroutines.s12`.

Du testar Nstep i samband med att du utför nästa uppgift.

Slut Uppgift 100

Subrutinen Auto implementerar en automatisk borr-robot. Rutinen borrar ett antal hål i arbetsstycket enligt en given tabell för hålmönster som beskrivits ovan.

Uppgift 101

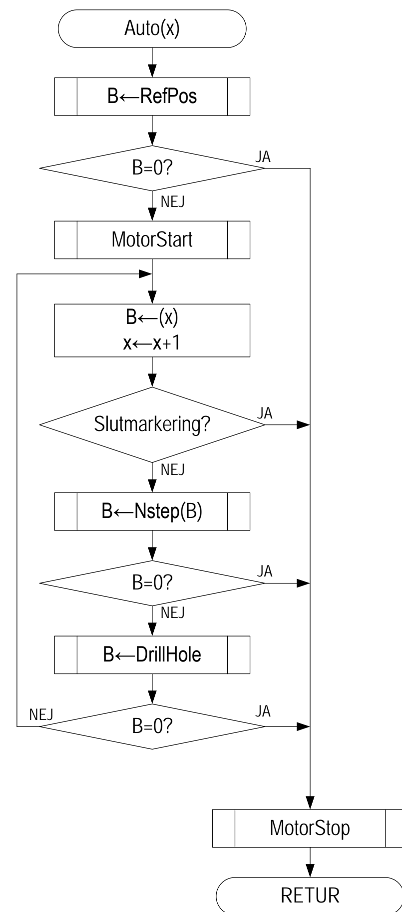
- Komplettera i `Main.s12` med en tabell `Pattern` som innehåller hålmönstret.

```
; Placera alla globala variabler här
DCShadow:  RMB     1
Pattern:   FCB     0,1,1,1    etc
```

- Subrutinen DoAuto används för att först ladda register X med parametern och därefter anropa Auto. Implementera rutinen DoAuto och lägg till den i `Subroutines.s12`.
- Korrigera hopptabellen i Command genom att ändra `SUB7` till `DoAuto` enligt följande. Du kan därefter ta bort dummy-rutinen `SUB7`.

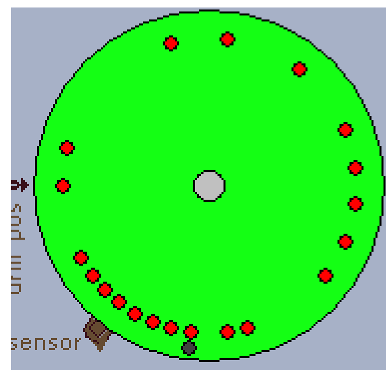
```
;   Tabell med subrutinadresser
JUMPTAB    FDB     MotorStart,MotorSTOP
           FDB     DrillDown,DrillUp
           FDB     Step,DrillHole
           FDB     RefPos,DoAuto
```

- Implementera nu även rutinen Auto enligt följande specifikation. Lägg till den i `Subroutines.s12`.




```
; Subrutin Auto.
; Borrar ett antal hål enligt ett givet
; mönster i ett arbetsstycke
;
; Anrop:   LDX     #Pattern
;          JSR     Auto
; Indata:  Adressen till tabell med
;          hålmönster i register X
; Utdata:  Inga
; Register:A,B och X kan ändras
; Anropade:MotorStart, MotorStop, RefPos
;          Nstep, DrillHole
```

Välj 'New Disc' = 4 då du testar doAuto. Om din lösning är riktig ska hål ha borrats i samtliga ringar, inga hål ska ha borrats utanför ringarna, se följande figur:



Slut Uppgift 101

Avsnitt 5

Maskinnära programmering i C och assemblerspråk

Syften:

I detta avsnitt kommer du att introduceras till maskinnära programmering i 'C'. Du får lära dig att hantera ett modernt verktyg för utveckling och test av applikationer för MC68HC12-baserade mikrodatorer. Du får se exempel på hur kodgenerering utförs, du får samtidigt en rad tips om hur du programmerar maskinnära i C. Avslutningsvis har du möjlighet att testa dina nyvynna kunskaper och kombinera dom för att lösa enklare programmeringsproblem

Inledning

XCC12 är ett integrerat program-utvecklingsverktyg för MC68HC12 - baserade mikrodatorer. *XCC12* har från början utvecklats som en s.k "korskompilator", dvs ett program-utvecklingssystem där man använder en speciell värddator för att utveckla program till en annan maskin, den s.k måldatorn. *XCC12* har också anpassats speciellt för utbildningsändamål och det är därför mycket enkelt att skapa applikationsprogram för exempelvis laborationsdatorn *MC12*.

De primära syftena med *XCC12* är:

- Tillhandahålla ett IDE (*Integrated Development Environment*) som på ett enkelt och intuitivt sätt ger möjlighet att utföra de olika momenten vid programutvecklingen.
- Tillhandahålla en ANSI-C kompilator för MC68HC12-baserade mikrodatorer.
- På ett tydligt och enkelt sätt illustrera hur en modern utvecklingsmiljö är uppbyggd.

Efter att ha arbetat dig igenom momenten ska du inte bara ha lärt dig att utveckla program (i 'C' och assembler) du ska också ha bekantat dig med en rad nya viktiga begrepp i dessa sammanhang. Dessa kunskaper är allmängiltiga och du kommer att ha mycket stor nytta av dem i fortsatt arbete med programutveckling för så kallade "inbyggda system".

Introduktion – det första projektet

XCC tillhandahåller *Project Manager* för att hjälpa dig organisera källtexter, bibliotek och så kallade 'projektfiler' (anvisningar om hur att kompilera och länka..) på ett enkelt sätt.

Ett 'Projekt' utgörs av minst en källtextfil och en projektfil. Projektfilen skapas och underhålls av XCC. Det är inte meningen att du ska redigera denna själv. Ett projekt syftar till att skapa en applikation och det finns tre olika typer av applikationer:

- Exekverbar fil
- Programbibliotek
- Länkad objektfil

Projektets typ anges då du skapar det, typen kan därefter inte ändras. Du kan däremot lägga till nya källtextfiler efter hand, ändra en rad olika projektinställningar etc.

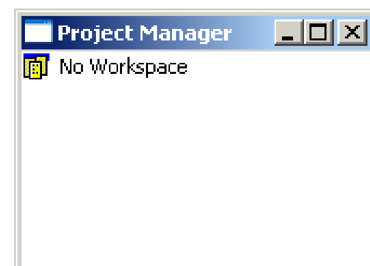
Projekten organiseras i ett 'Workspace'. Workspace kan innehålla obegränsat antal projekt och ett projekt kan samtidigt finnas upptaget i flera olika Workspace. Du kan bara ha ett Workspace öppet åt gången i XCC. Låt oss illustrera begreppen med följande övning.

Skapa ett arbetsbibliotek (exvis c:\xcc). Under detta arbetsbibliotek, skapa biblioteken:

xcc\src   här skapar du alla källtextfiler.

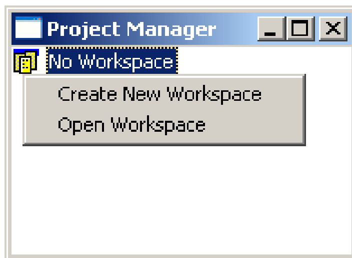
xcc\ws   här skapas 'Workspace' och projekt

Starta XCC om du inte gjort det tidigare, det enda fönstret du ser är projekthanteraren:



Projekthanteraren har ett fönster som alltid ligger ovanför andra fönster. Du kan gömma det genom att klicka på 'Close', visa det igen via menyvalet. 'Project | View Project Manager'

Högerklicka en gång på texten 'No Workspace'.



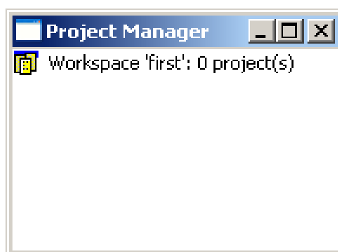
Härifrån kan du skapa ett nytt, eller öppna ett befintligt 'Workspace'. Du kan göra samma saker via menyn:

'File | New Workspace' eller

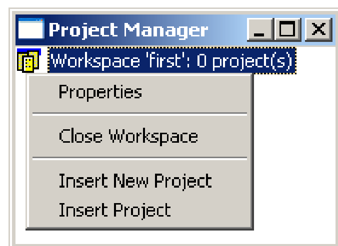
'File | Open Workspace'

Skapa nu ett nytt workspace '`first`' i biblioteket 'C:\xcc\ws'.

Observera hur projekthanteraren nu visar ett aktivt workspace:



Högerklicka på texten som tidigare, följande popup mey visas:

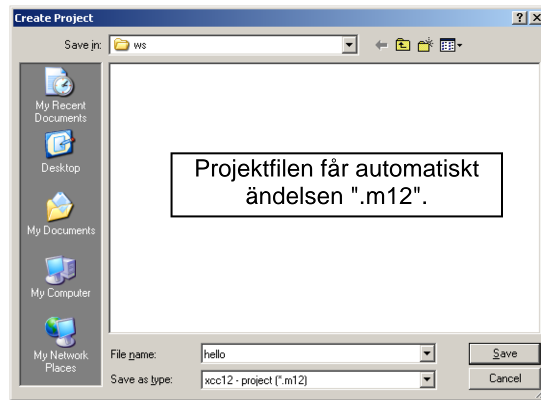


Alternativ:

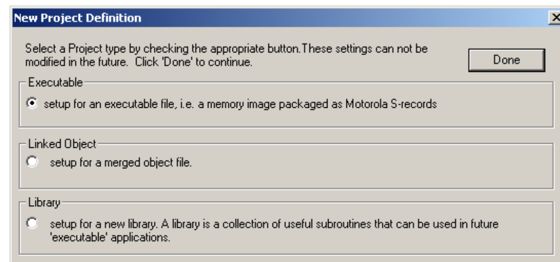
- *Properties* - visar information om filens egenskaper
- *Close Workspace* - stänger aktivt workspace
- *Insert New Project* - Skapa ett nytt projekt och inför detta i aktivt workspace
- *Insert Project* - Inför projekt som skapats tidigare (eventuellt under ett annat workspace) i aktivt workspace.

Du kan också skapa nya/sätta in gamla projekt via menyvalet 'Project'

Välj Insert New Project från menyn, du får då en dialogbox "Create Project", välj arbetsbiblioteket och skapa projektet '`hello`':



Klicka på "Save" , du får nu följande dialogruta:



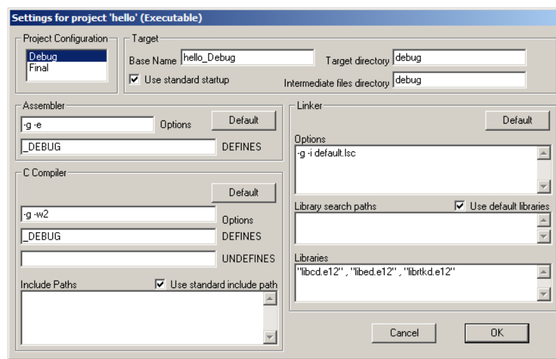
Vi vill nu skapa en exekverbar fil enligt förslaget, klicka därför på 'Done'.

Nästa bild är 'Project Settings'-dialogen (se figur nedan). Du kan öppna denna när som helst om du behöver ändra projektinställningar. Notera inställningar i 'Target'-sektionen. Här anges:

- 'Base Name' - applikationens namn. XCC lägger till korrekt filnamnstillägg.
- 'Target directory' - här anges det underbibliotek (relativt projektfilen) där *resultatfiler* placeras av XCC.
- 'Intermediate files directory' - här anges det underbibliotek (relativt projektfilen) där *temporärfiler* placeras av XCC.
- 'Use standard startup' – Här anges om en generell startfil ska användas. I de allra flesta fall är denna tillräcklig. Om inte denna ruta är märkt måste du tillhandahålla en egen startfil med de nödvändiga funktionerna. Vi återkommer till detta.

För varje projekt definieras två konfigurationer. 'Debug'-konfigurationen används för att skapa applikationer med Debug-information. Då applikationen är färdigutvecklad kan man växla till 'Final'-konfiguration, den resulterande koden blir då i allmänhet bättre än med 'Debug'-konfiguration.

För vårt inledande exempel är det lämpligt att använda de föreslagna inställningarna.



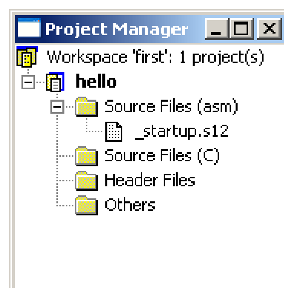
I vårt fall kommer således applikationen

'hello\debug\hello_Debug.s19'

så småningom att skapas.

Klicka 'OK' för att stänga dialogen.

Observera hur Projekthanteraren uppdateras och aktiverar det nya projektet.



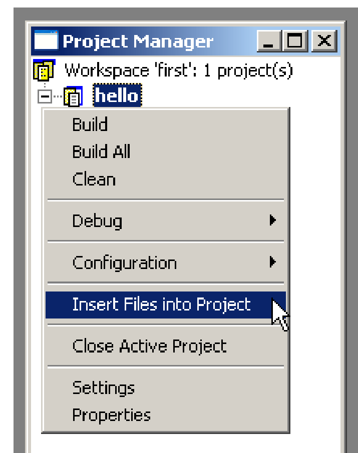
Projekthanteraren sorterar filerna med avseende på filnamnsändelser under separata flikar. Detta har ingen annan praktisk betydelse än att de ska vara mer överskådliga.

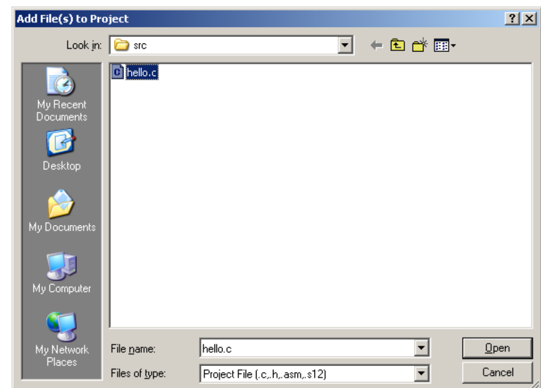
Välj nu, från menyn, File | New och skapa filen 'hello.c' i biblioteket xcc\src. En editor startas, skriv in följande program:

```
hello.c

#include        <stdio.h>

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

Välj File | Save för att spara den nya filen.

Märk det aktiva projektet och högerklicka, du får en ny pop-up-meny. Välj 'Insert Files into Project'.



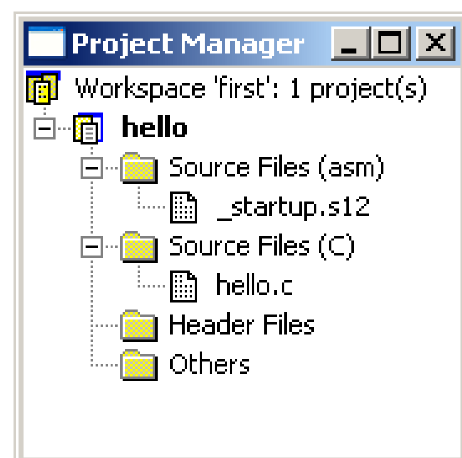


Tips:

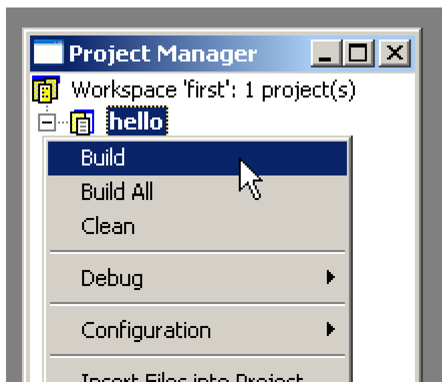
Du kan använda 'multiple select' här om du vill lägga till fler filer från samma bibliotek samtidigt. Håll ned 'Ctrl'-tangenten samtidigt som du märker de filer du vill ska ingå i projektet.

Välj den nya filen 'hello.c' och klicka på 'Open'.

Projekthanterarens fönster uppdateras....



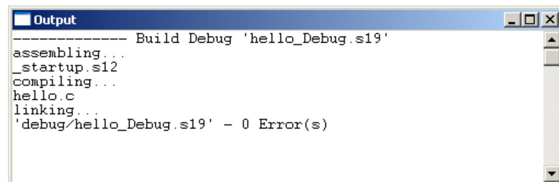
Du skapar nu den exekverbara filen 'hello_Debug.s19' genom att välja Build.



Anmärkning:
Alternativet 'Build' innebär att endast de filer som ändrats sedan applikationen skapades förra gången kommer att behandlas. 'Build All' innebär att alla filer behandlas oavsett om de ändrats eller ej. Alternativet 'Clean' får XCC att ta bort alla objektfiler och själva applikationen.

XCC startar nu kompilering och länkning...

Du kommer att få meddelanden i "Output"-fönstret:



XCC assemblerar filen '_startup.s12', kompilerar filen 'hello.c', länkar därefter samman modulerna med nödvändiga biblioteksrutiner och skapar laddfilen 'hello_Debug.s19'.

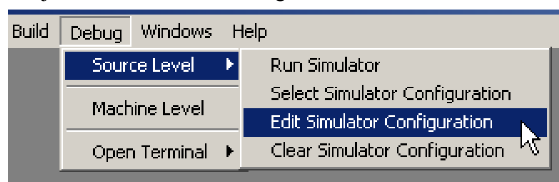
Debugging

Du kan testa ditt nya program med hjälp av XCC's källtext-debugger (SLD, *Source Level Debugger*). För att kunna se utskriften från programmet måste du också koppla en serie-enhet till IOsimulatorns konsollfönster.

Uppgift 102

Anslut ett konsollfönster:

Välj 'Edit Simulator Configuration':

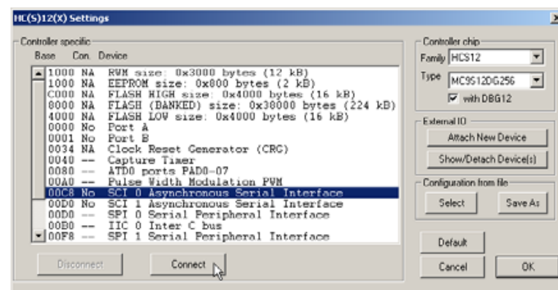


Fönstret med inställningar för simulatorn öppnas.

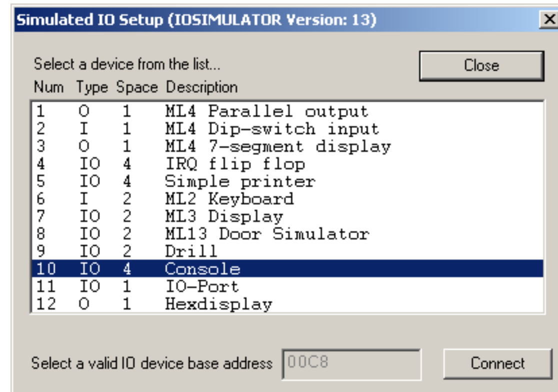
Märk nu raden:

`'00C8 No SCI 0 Async...'`

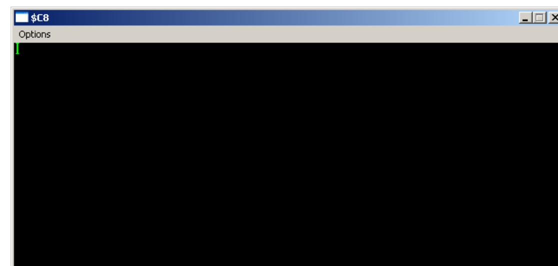
Denna motsvarar den inbyggda seriekretsen på adress 00C8. Vi ska koppla den till ett så kallat "terminalfönster".



Klicka på 'Connect'. Fönster med tillgängliga enheter öppnas, välj enheten 'Console'.

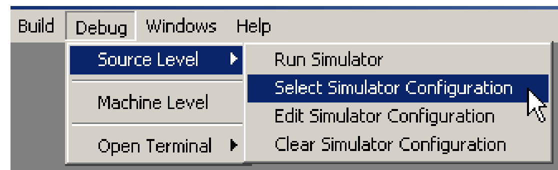


Klicka på 'Connect', konsollfönstret öppnas.



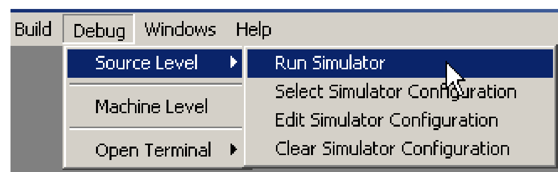
Stäng nu fönstret med "tillgängliga enheter". Spara därefter inställningarna under namnet 'console'. Stäng fönstret för simulatorinställningarna.

Vi kopplar nu dessa simulatorinställningar i debuggern och på så sätt laddas de automatiskt varje gång vi startar debuggern med detta projekt:



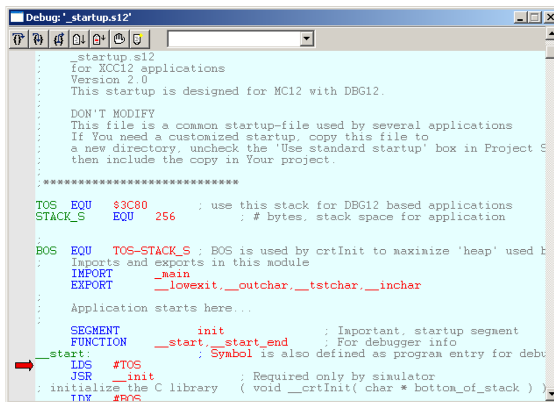
Välj: 'Select Simulator Configuration' och ange därefter den fil du nyss sparade.

Välj nu 'Run Simulator' för att starta debuggern och den inbyggda simulatorn.



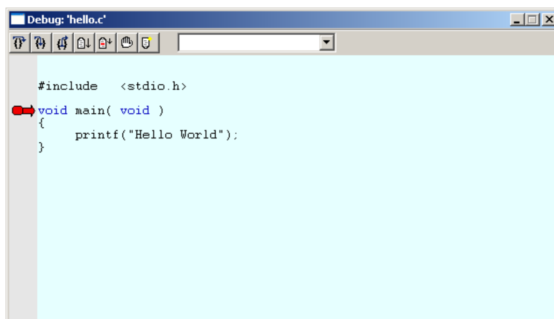
Välj därefter filen: `hello_Debug.s19`.

När debuggern startas skapas också flera nya fönster, ett känner du igen sedan tidigare, det är simulator-fönstret. Simulatorn är den samma som du tidigare använt under ETERM. Via program-fönstret kan du kontrollera debuggern.

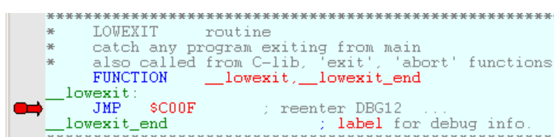


Programmets första exekveringspunkt finns i filen '_startup.s12' som länkats in först av XCC. Vi ska inte nu fördjupa oss i själva uppstarten av programmet. Debuggern sätter alltid ut en brytpunkt på adressen för symbolen '_main' (som ju är det egentliga applikationsprogrammet). Det är därför lämpligt att starta programmet med 'Run' från verktygsfältet. Observera att det finns två varianter. En variant ignorerar brytpunkter, den vill vi *inte* använda nu, en annan variant respekterar brytpunkter. Vi väljer denna.

Debuggern startar programmet men det stannar omedelbart vid funktionen 'main'.



Det finns nu flera alternativ om vi vill testa och ”avlusa” programmet. I detta fall är det allt för enkelt, endast en biblioteksrutin används och det är rimligt att bara utföra programmet. Välj på nytt 'Run' från debuggerns verktygslist. Programmet återstartas och exekveras tills det är färdigt, i debug-fönstret ser du hur debuggern nu stannat programmet i startup-modulen.



I konsollfönstret ska du dessutom se 'printf'-satsens utskrift.

Om du inte ser texten 'Hello World' här måste du gå tillbaka och i första hand kontrollera anslutningen av IO-simulatorn. Testa programmet och kontrollera funktionen.

Det exempel vi använt förutsätter en *MC12* måldator. Mycket av detta beroende finns i vår 'startup'-modul. Vi kommer att behandla 'startup' moduler för generell användning längre fram.

Slut Uppgift 102

Uppgift 103

Skapa ett nytt projekt – ”hexnum” – med ett program som:

skriver de hexadecimala talen 0 – 0x2F till konsollen. Talen ska separeras med blanksteg.

Ledning: 'printf' kan formatera utskrift på hexadecimal form med:

```
printf(" %x", ........ );
```

Använd hjälpsystemet, läs mer om biblioteks-funktioner.

Testa programmet och kontrollera funktionen.

Slut Uppgift 103

Debuggerns funktioner

Du har nu sett ett inledande exempel på hur du kan använda XCC med färdiga programbibliotek och enkelt skapa en komplett applikation.

Utveckling av nya maskinnära program kräver alltid speciella kunskaper men innan vi går vidare med mer intrikata detaljer så som kompilatorns kodgenerering och olika programbibliotek, är det lämpligt att belysa olika möjligheter med debuggern.

Programmets *exekveringspunkt* indikeras med en röd pil i debug-fönstret. Exekveringspunkten motsvaras oftast (inte alltid) av en enstaka rad i en källtext.

EXEMPEL

```
if( a < b ) return 1;
```

har två exekveringspunkter:

if-satsen och

return-satsen

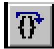
om du skriver samma sak enligt:

```
if( a < b )
    return 1;
```

kommer det att bli lättare att följa programflödet i debuggern.

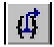
För assemblerkälltexter gäller att varje rad motsvarar en unik exekveringspunkt.

Som du kanske redan märkt, ändras menyalternativen då du startar debuggern. Själva 'Debug'-menyn omfattar då följande alternativ, observera att flera av dom kan du också välja från debug-fönstrets verktygslist.



- **Next** - Exekvera förbi - dvs utför alla satser/instruktionen på den aktuella raden. För C-källtexter kan detta innebära flera exekveringspunkter (alla på raden). För assemblerkälltexter gäller det vanligtvis en instruktion, untantaget är dock programflödesinstruktioner. Exempelvis kan en hel subrutin exekveras.

- **Step Into** - Stega in, används för att stega in i funktioner och subrutiner. För C-källtexter innebär detta att programmet utförs till nästa verkliga exekveringspunkt och man kan därför följa programflödet in i funktioner vid anrop. För assemblerkälltexter innebär det samma sak som instruktionsvis exekvering.



- **Step Out** - För att snabbt utföra programmet till funktionens sista rad. För C-källtexter innebär detta att programmet exekveras fram till den aktuella funktionens 'epilog' (beskrivs under 'kodgenerering' nedan) dvs ett väldefinierat block med kod som avslutar funktionen. För assemblerkälltexter innebär det exekvering fram till nästa "return from subroutine"-instruktion.

- **Run Nobreak** - Exekvera programmet och ignorera eventuella brytpunkter. Programmet startas av debuggern som därefter ignorerar samtliga brytpunkter. Debuggern stoppar programmet först då det når punkten 'exitlabel'.

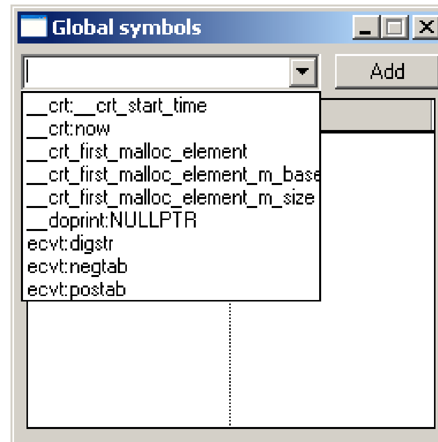
- **Run** - Exekvera programmet till nästa brytpunkt. Programmet startas av debuggern och exekveras fram till nästa brytpunkt (se brytpunktshantering nedan).

- **Breakpoints** - Öppna dialogruta för brytpunktstabellen. Används för att sätta, aktivera/deaktivera och ta bort brytpunkter.

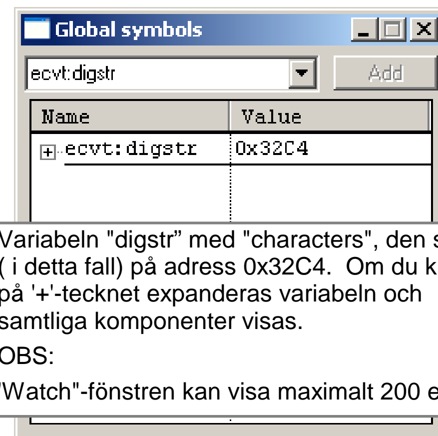
- **Restart** - Starta om programmet från början. Förbereder omstart av programmet. Eventuella brytpunkter och 'Watch'-variabler behålls.

Globala variabler

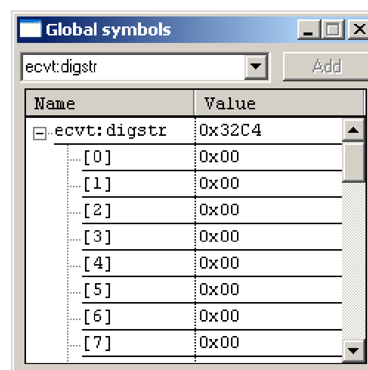
Du kan inspektera variabler med hjälp av 'watch'-funktionen. Skriv namnet på den variabel du avser *eller* välj från listan du får då du fäller ut 'combo'-boxen. Klicka därefter på 'Add'.



Prova funktionen genom att lägga till "ecvt:digstr"...



Klicka på '+'-tecknet för att expandera en strängvariabel...



Du kan ändra variabelvärde, klicka i 'Value'-fältet för den variabel du vill påverka...

Name	Value
ecvt:digstr	0x32C4
[0]	0x00
[1]	0x00
[2]	0x00
[3]	0x00

Skriv in något nytt värde och tryck 'Enter'. Tänk på att värdet tolkas beroende på datatyp. Använd prefix för andra talbaser än decimalt.

Lokala variabler

Visningen av lokala variabler sker automatiskt, dvs så fort programmet når en exekveringspunkt där lokala variabler (eller parametrar) är 'synliga' så aktiveras detta fönster.

I övrigt fungerar detta fönster på samma sätt som för globala variabler. Tänk på att lokala variabler tilldelas i den första exekveringspunkten i en funktion. Detta innebär att du måste "stega in" i en funktion som har parametrar och/eller lokala variabler för att dessa ska visas i fönstret.

Brytpunkter

Brytpunkter hanteras dels med hjälp av den gråa listen i 'debugfönstret', dels med hjälp av en brytpunktstabell.

Den röda markeringen anger att det finns en aktiv brytpunkt på denna raden.

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

Placera markören i det grå fältet, på raden med brytpunkten och högerklicka...

Du får nu alternativen

- Disable Breakpoint, brytpunkten sparas i tabellen men är inte längre aktiv. Du kan aktivera den igen om du vill
- Remove Breakpoint, ta bort brytpunkten.

Välj 'Disable Breakpoint'...

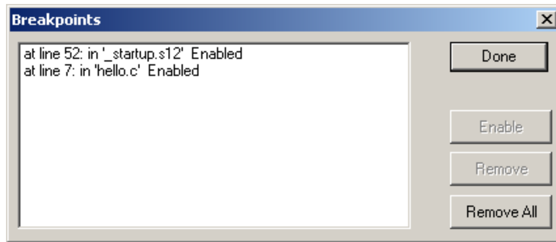
```
void main( void )
{
    printf("Hello World");
}
```

Den blå markeringen anger att det finns en inaktiverad brytpunkt på raden.

På motsvarande sätt kan du sätta ut en ny brytpunkt genom att placera markören på en rad och högerklicka.

Observera att detta bara fungerar för rader som har exekveringspunkter. Om du placerar markören på en rad som inte har exekveringspunkt händer ingenting då du högerklickar.

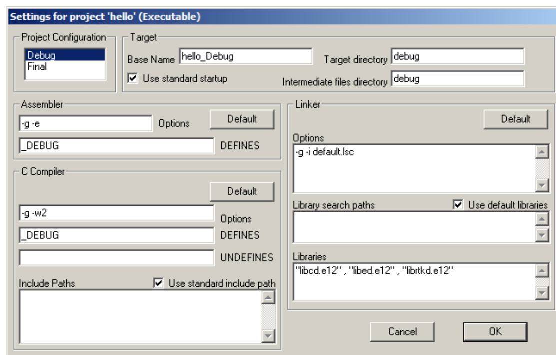
Slutligen, kan brytpunkter hanteras med hjälp av brytpunktstabellen.



Tabellen ger dig en översikt av samtliga brytpunkter. Du kan aktivera/inaktivera (beroende på aktuellt tillstånd). Du kan ta bort enstaka brytpunkter (märk en brytpunkt och klicka på 'Remove') och du kan ta bort samtliga brytpunkter på en gång ('Remove All').

Projekt-dialogen

Låt oss nu ta ytterligare en titt på dialog-fönstret Project | Settings. Här kontrollerar du projektets generella inställningar.



Project Configuration

Här visas projektets konfiguration, du kan välja mellan 'Debug' för att låta kompilator, assembler och länkare skapa extra information för debuggern. Då du vill skapa din slutgiltiga applikation väljer du konfiguarationen 'Final'. Nu skapas ingen debuginformation, filerna blir därför mindre.

Target

Här anges namnet på projektets resultatfil (Target Base Name). Till namnet läggs automatiskt rätt tillägg

`.s19` - för exekverbara filer
`.qld` - för länkade objektfiler
`.e12` - för programbibliotek

Här visas också var alla filer placeras ('Target directory' för resultatfil och 'Intermediate Files Directory' för exempelvis objektfiler som krävs för att skapa resultatfiler). Sökvägen är relativ det bibliotek där projektfilen finns, dvs det bibliotek där du skapade projektet. Biblioteken skapas automatiskt. Här anger du också om du vill använda den generella startupfilen ('Use standard startup'). Om du inte väljer detta måste du tillhandahålla en egen fil med de nödvändiga rutinerna i `'_startup.s12'`.

Assembler

Options - här anges de flaggor som används för assemblerkälltexter, du kan också sätta individuella flaggor för olika källtexter, mer om detta nedan.

DEFINES - här kan du definiera makron för assemblerkälltexter.

C-Compiler

Options - här anges de flaggor som används för C-källtexter, du kan också sätta individuella flaggor för olika källtexter, beskrivs nedan.

DEFINES - definiera makron för kompilatorn. Jämför med preprocessordirektivet `#define`

UNDEFINES - avlägsna makrodefinition för preprocessorn. Jämför med direktivet `#undef`.

Use standard include path – Standard-bibliotek för "header-filer" är `INSTALL/include` - där INSTALL är det bibliotek där XCC installerats. Denna ruta ska vara ifylld om du vill att preprocessorn ska söka efter "include"-filer i detta bibliotek.

Include Paths - Här inför du ytterligare sökvägar för "include"-filer om du vill.

Linker/Loader

Options - här anges flaggor för länkningen. Flaggornas funktion och användning beskrivs detaljerat i XCC's hjälpsystem.

Use default libraries - Standardbibliotek för XCC's programbibliotek är `INSTALL/lib/xcc12` - där INSTALL är det bibliotek där XCC12 installerats. Denna ruta ska vara ifylld om du vill att länkaren ska söka efter funktioner i XCC12's standardbibliotek.

Library search paths - Här anger du bibliotek där länkaren ska söka efter XCC12's programbibliotek (.e12-filer) . Om en sökväg inleds med '\' innebär detta relativt biblioteket INSTALL, annars ska fullständig sökväg anges.

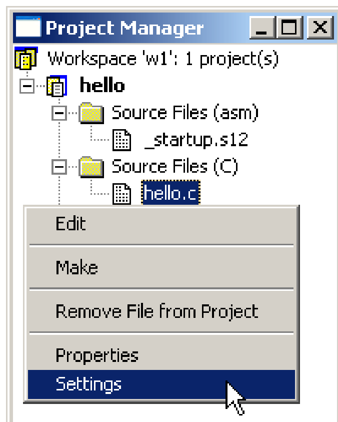
Libraries - Här anges de programbibliotek (endast namnen, ingen sökväg) du vill att länkaren ska söka i efter eventuella oupplösta symbolnamn. Detta är typiskt extra bibliotek som exempelvis standard c-biblioteket (libc), "extended library" (libe), eller "real time kernel library" (librtk) men det kan självfallet vara bibliotek du själv konstruerat.

Du kan modifiera dessa projektinställningar när som helst. Med 'Default'-knapparna återställes alla inställningar till standardinställningarna för projekttypen, dvs de inställningar som visas ovan.

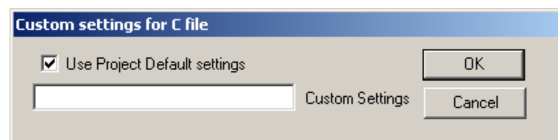
Individuella inställningar

De inställningar som anges under *Assembler* och *C-Compiler* antas gälla för alla projektets källtextfiler. I bland är det önskvärt att använda andra inställningar för någon fil. Du kan göra detta med individuella inställningar.

Om du märker en fil i projekthanteraren och därefter högerklickar får du följande pop-up-meny:



Väljer du 'Settings' öppnas en dialogbox där du kan göra inställningar som gäller endast för denna källtext.



Om du vill att standardinställningarna ska användas ska rutan 'Use Project Default settings' vara ikryssad. Du kan också ge extra flaggor till C-kompilatorn på raden 'Custom Settings'.

De flaggor ('Options') som kan anges kan du läsa om i XCC's hjälpsystem.

XCC, arbetssätt

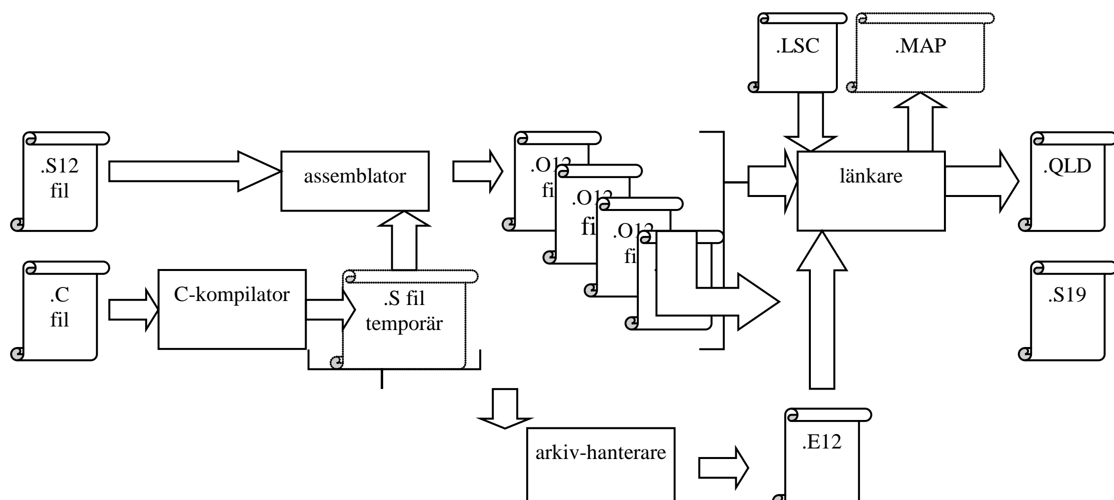
XCC består i själva verket av flera programdelar:

- Projekthanterare, Debugger
- C-kompilator
- Relokerande assemblator
- Länkare/Arkivhanterare

Projekthanteraren, som vi delvis har behandlat, används för att ange alla källtextfiler som ingår i projektet. Dessa kan vara källtexter i programspråket C och ska då ha filnamnsändelsen .C, men de kan också vara källtexter i assemblerspråk, de ska då ha filnamnsändelsen .asm eller .s12. För filnamn med ändelser .s12, .c eller .h aktiveras automatiskt editorns färgade syntax. Vissa av Projekthanterarens funktioner kan också utföras från menyn Project.

När du arbetar med ett projekt kan du välja Build All, Projekthanteraren kommer då att kompilera alla C-filer, assemblera alla källtexter och därefter länka samman dessa till en färdig modul (exekverbar, länkad objekt eller bibliotek). Om du i stället väljer Build kommer projekthanteraren endast att kompilera/assemblera de källtexter som du ändrat i sedan projektet byggdes förra gången. För att utföra detta krävs alltså att projekthanteraren har kännedom om de beroenden som existerar. Exempel på ett sådant beroende är mellan den objektmodul (filnamnsändelse .O12) som skapas av kompilator/assembler och källtexten. Om projekthanteraren upptäcker att en fil *file.c* har sparats senare än den motsvarande *file.o12* kommer filen att kompileras om automatiskt före länkningen. *XCC* kan automatiskt bestämma sådana beroenden. Du kan se dessa genom att välja 'Properties' från pop-up menyn för en märkt källtextfil.

Om du endast vill kompilera/assemblera den fil du för tillfället arbetar med kan du välja Make, detta kräver att du samtidigt har filen öppen i någon editor eller väljer alternativet från en pop-up meny.



En källtext med filnamnsändelsen .C bearbetas av C-kompilatorn. Här sker först så kallad preprocessing, dvs alla preprocessordirektiv utförs, därefter vidtar översättningen av programmet till assemblerkod. Hanteringen av `#include`-direktivet innebär att preprocessorn söker filer antingen i projektbiblioteket (”current directory”) eller i något fördefinierat bibliotek.

En källtextfil med filnamnsändelse .S12 eller .asm dirigeras direkt till assemblatorn som assemblerar filen och skapar en objektmodul, .O12. Objektmodulen innehåller relokerbar kod, dvs inga absolutadresser är ännu bestämda.

Länkaren sammanfogar flera objektmoduler till en enda (.QLD). Denna modul kan användas för vidare länkning med andra programdelar men den kan också, om den innehåller ett färdigt program, användas för att generera en laddfil (.S19). Förutom att länkaren sammanfogar alla objektmoduler från filer som ingår i projektet, kan den också instrueras att söka i speciella programbibliotek.

Följande tabell ger en översikt av XCC’s filtyper och deras respektive filnamnstillägg:

.W12	'Workspace' - innehåller information om de projekt som ingår.
.M12	Projektfil, innehåller beskrivning av projektet, dvs alla ingående källtexter, beroenden och växlar för att styra bygget av projektet.
.C	källtext ,C
.S12, .asm	källtext, MC68HC12 assembler
.O12	objektfil
.S	temporär assemblerfil, skapas av kompilatorn. Normalt tas denna fil bort efter assembleringen. Du anger att filen ska sparas genom att ge flaggan '-S' till kompilatorn.
.MAP	listfil från länkning, innehåller absoluta adresser för samtliga globala symboler i laddmodulen.
.LSC	Kommandofil för länkare
.QLD	länkad objektfil
.E12	programbibliotek
C1xxxx C2xxxx RAxxxx QLxxxx	temporärfiler som skapas av XCC. Dessa ska normalt sett tas bort av XCC. Om du hittar sådana filer beror detta på att XCC avslutat på något onormalt sätt. Du måste då avlägsna dessa filer manuellt.

Stand Alone Applikation

I detta moment visar vi hur du enkelt skapar en fristående applikation (dvs utan användning av färdiga programbibliotek). Vi delar upp applikationen i två olika källtexter:

```
appstart.s12
ml4.c
```

`appstart.s12` innehåller, som namnet antyder, den nödvändiga koden för att starta C-programmet, vars huvudfunktion (”huvudprogram”) alltid heter ”main”. En minimal ”appstart” blir följaktligen:

```
;   Här börjar exekveringen...
        JSR     _main
```

Observera “underscore” framför symbolnamnet. Detta är en konvention hos XCC och används för att ingen sammanblandning ska kunna ske mellan applikationsdefinierade symboler och reserverade namn. Då vi vill referera symboler som definierats i någon C-källtext (i detta fall ”main”) måste vi alltid tänka på denna konvention.

Vår minimala "appstart" kan vara riskabel, vad händer om inte stackpekaren har ett riktigt värde vid anropet ? I värsta fall spårar programmet ur redan då eftersom instruktionen (JSR) innebär att återhoppsadressen läggs på stacken. Vi garderar oss genom att lägga till en instruktion som initierar stackpekaren:

```
;   Här börjar exekveringen...
          LDS     #$2FFF
          JSR     _main
;
```

Vad händer nu å andra sidan om exekveringen avslutas i ”main” och processorn försöker återuppta den efter JSR-instruktionen ? Förmodligen löper programmet amok på ett mer eller mindre okontrollerat sätt. Vi kan gardera oss även mot detta.

```
;   Här börjar exekveringen...
          LDS     #$2FFF
          JSR     _main
exit:     NOP
          BRA     exit
;
```

Dvs vi avslutar med en oändlig programslinga där programmet inte gör någonting.

I en del applikationer vill man kunna avsluta med ett direkt anrop av funktionen ”exit” från ett program. Vi kan enkelt möjliggöra detta nu men måste då komma i håg kompilatorns namnkonventioner. För att kunna referera en symbol som definierats i en assemblerkälltext måste symbolen inledas med ’_’. Vår version av ”appstart” blir då:

```
;   Här börjar exekveringen...
          LDS     #$2FFF
          JSR     _main
_exit:    NOP
          BRA     _exit
;
```

Vi kan tala om för assemblatorn att symbolen ”exit” ska vara global, dvs kunna refereras från en annan källtext, med direktivet:

```
    export      _exit
```

För att avsluta ett C-program krävs nu bara funktionsanropet:

```
    exit();
```

Symbolen ”_main” är definierad i en annan källtext. Vi talar om detta för assemblatorn med direktivet:

```
    import      _main
```

Vi måste också definiera ett *segment* för koden. Eftersom detta är så kallad ”startup-kod” använder vi segmentnamnet `init`. Vid länkningen ger vi sedan segmentet en lämplig startadress.

```
        segment     init
```

På detta sätt är det lätt att bestämma en väldefinierad startpunkt för vår applikation. Segmentet får bara definieras i en fil i applikationen.

För att göra det enklare att testa vårt program inför vi start- och slut-symboler för vår startupkod. Vi definierar också koden med:

```
        function    __start,__start_end
```

Observera att dessa definitioner enbart är till för debuggern.

Vår slutliga ”appstart” blir nu:

```
        segment     init
        export      _exit
        import      _main
        function    __start,__start_end
*   Här börjar exekveringen...
__start
        LDS         #$2FFF
        JSR         _main
_exit:  NOP
        BRA         _exit
__start_end
```

Uppgift 104

Skapa en källtext `Appstart.s12` enligt ovanstående anvisningar. Du kommer att få användning av den om en stund.

Slut Uppgift 104

Vi övergår nu till själva applikationen, dvs ”main”. Vi ska skriva ett enkelt program som läser från en inport, skiftar detta värde ett steg till höger och skriver resultatet till en utport (Jämför inledande exempel i Avsnitt 1).

Denna uppgift visar exempel på hur fysiska portar, eller absoluta adresser i allmänhet, kan kommas åt direkt från ett C-program. Man behöver alltså inte (vilket är en vanlig missuppfattning) använda assemblerrutiner bara därför att man exempelvis skriver rutiner som hanterar speciella periferienheter.

Absolut adressering i C

Portar med fasta adresser i minnet kan adresseras genom att tillämpa en typkonvertering på konstanten, minns att typdeklarationer "läses bakifrån" och betrakta följande:

```
*((char *)Portadress)
```

`Portadress`, är den fysiska adressen (en konstant).

`(char *)` anger att `Portadress` är adressen till en 8-bitars port, om det är en 16-bitars port använder vi konverteringen `(short int *)`, för en 32-bitars port används `(long int *)`.

Slutligen, `*` framför yttersta parentesen anger helt enkelt att det är innehållet på denna adress som avses.

Vill vi ange innehållet på ML4’s inport får vi då

```
    *((char *) ML4IN)
```

Vi definierar nu följande *macros:*

```
#define ML4IN       0x0600
#define ML4OUT      0x0400

#define ML4READ *((char *) ML4IN)
#define ML4WRITE    *((char *) ML4OUT)
```

Följande sekvens visar hur vi sedan deklarerar en variabel, tilldelar denna värde från ML4’s inport, skiftar variabeln ett steg åt höger och slutligen skriver variabelns värde till ML4’s utport:

```
        char  c;
        c = ML4READ;
        c = c >> 1;
        ML4WRITE = c;
```

Uppgift 105

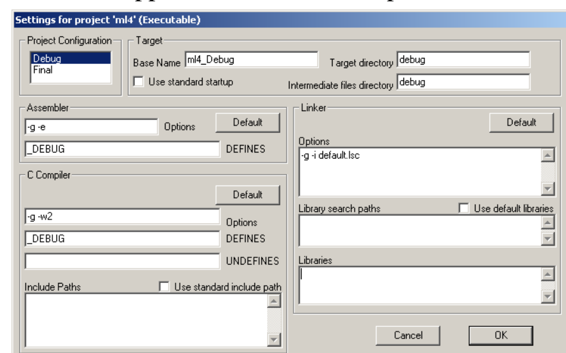
Skapa en fil `Ml4.c` med ett huvudprogram (main) som kontinuerligt läser från ML4’s inport, skiftar, och skriver resultatet till ML4’s utport. Använd makrodefinitioner för portadresserna.

Slut Uppgift 105

Det är dags att skapa projektet – ”bygga” och testa vår lilla stand-alone applikation

Skapa ett nytt projekt **ML4** ('Insert New Project into Workspace').

Följande inställningar är lämpliga för den fristående applikation vi nu ska skapa:



Notera att vi avmarkerat *alla* standardalternativ för projektet ('Use standard startup', 'Use default libraries', Use standard include path').

Uppgift 106

a. Skapa en ny simulatorkonfigurationen där 'ML4 Dip Switch Input' kopplas till adress $0600 och 'ML4 Parallel Output' kopplas till adress $0400. Spara simulatorkonfigurationen under namnet 'ml4-simple'. Koppla konfigurationen till projektet
b. Skapa den exekverbara filen, dvs "bygg" projektet med:
   Build | Build All
c. Starta debugger'n
   Debug | Source Level | Simulator

Testa programmet genom att stega (rad för rad) och kontrollera funktionen. Ställ in något värde på Dip-Switchen, observera hur variabeln 'c' ändras (i fönstret 'Local Watch').

Slut Uppgift 106

Uppgift 107

Ändra nu testprogrammet så att ett vänsterskift utförs. Testa och kontrollera som tidigare.

Slut Uppgift 107

XCC12 Kodgenerering

Vi kommer nu att ge en detaljerad beskrivning av hur *XCC12* genererar kod för *MC68HCS12*.

Minnesdisposition

Programkod och data indelas i olika segment, figuren nederst på nästa sida 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 och exekveras.

prefix

Prefixet, det som vi tidigare kallat, *startupkod*, placeras först. Jämför med den tidigare övningen med "appstart". Prefixet ingår ofta i den så kallade *runtime-miljön* som installeras tillsammans med kompilatorn.

programkod

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

initierade data

Deklarationer som

```
int a = 2;
char array[] = {"String of Text"};
```

kan användas för att deklarera initierade 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. XCC använder två olika segment "`data`" och "`rodata`" (read-only data) för initierade globala variabler.

EXEMPEL

Beroende på *hur* en textsträng deklareras kommer kompilatorn att placera den i olika segment:

Satsen

```
printf("Denna text ...");
```

ger samma resultat på bildskärmen som:

```
char reftext[]={"Denna text ..."};
    printf("%s", reftext);
```

dvs en textsträng skrivs ut.

Kompilatorn betraktar dock textsträngarna på helt olika sätt. I det första fallet är det en konstant sträng, som inte kan refereras av programmet från någon annan punkt än just i printf-satsen. Eftersom den inte kan refereras kan den heller inte ändras, textsträngen är därför *read-only*, och placeras i `rodata`-segmentet.

I det andra fallet är det omedelbart klart att denna texsträng kan refereras även från andra ställen i programmet, t.ex:

```
strcpy(reftext, "Annan text...");
```

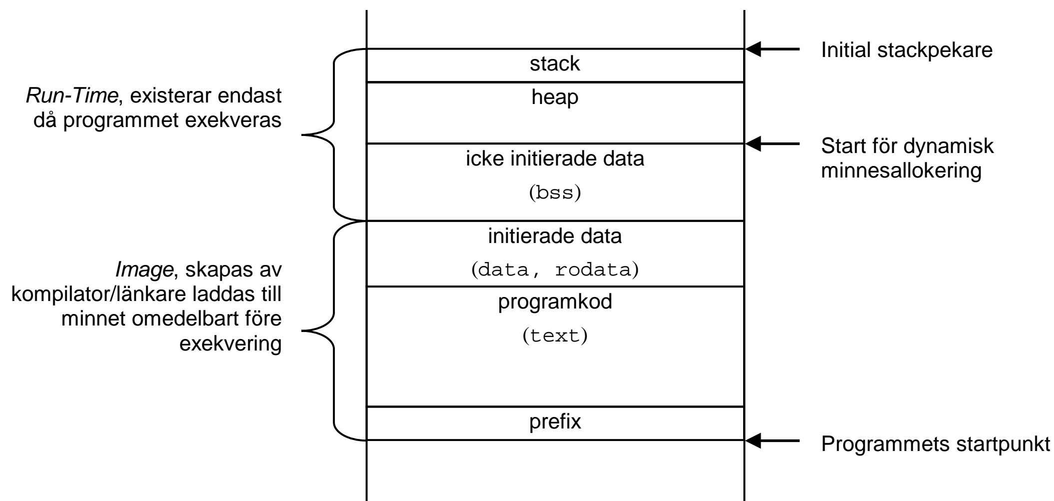
Textsträngen kan därför inte placeras i `rodata`-segmentet, i stället hamnar den i `data`-segmentet.

oinitierade data

Deklarationer som:

```
int   a;
char  array[34];
```

osv, används för att deklarera icke initierade globala variabler. 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*.



stack

Stacken används av program under exekvering. Storleken hos detta segment bestäms som regel av operativsystemet.

heap

"Heapen" benämns ofta det minnesutrymme som reserverats för dynamisk minneshantering `malloc()`, `free()` etc. Även storleken av detta segment bestäms som regel av operativsystemet.

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

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

Då programmet ska exekveras utförs följande:

1. Prefix adderas till textsegmentet.
2. Minnesbehov för segmenten 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 *XCC*, har man som regel inget operativsystem 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 (eventuellt görs detta av debuggern)
- 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.

ROM-kod

Applikationer som man vill placera i måldatorns ROM (Flash-minne) måste hanteras speciellt. XCC understödjer sådana applikationer men flaggan '-P' måste då ges vid länkningen. Dessutom måste startupkoden utökas med datakopiering från `data`-segmentet RO-minnet till en reserverad del av `bss` i RW-minne. Variabler i data-segmentet tar då alltså i själva verket dubbel så stor plats. Variablerna refereras under exekvering i RW-minnet.

namnkonventioner för segment

Observera att du *kan* definiera egna namn på segmenten, de fördefinierade namnen fungerar dock för de flesta applikationer.

Du kan läsa mer om detta i XCC's hjälpsystem.

XCC, minnesallokering

Vi ska nu se hur *XCC* genererar kod för några enkla variabeldeklarationer i **C**. Du kan själv enkelt upprepa detta genom att ange flaggan "-S" för källtexten i projektet. *XCC* kommer då att lämna de genererade assemblerfilerna som du kan granska med hjälp av texteditorn. Filnamnskonventionerna är enkla, om den kompilerade källtexten heter:

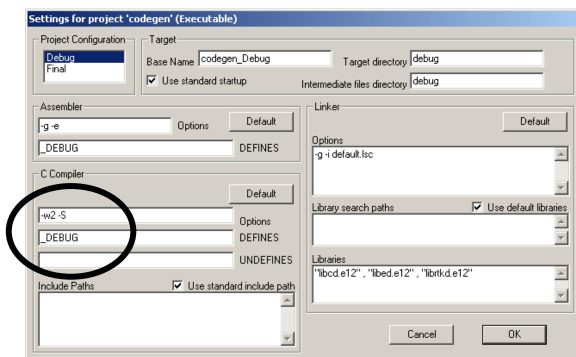
```
minfil.c
```

så kommer den genererade assemblerfilen att heta

```
minfil.S
```

Uppgift 108

Förbered följande övningar genom att skapa ett nytt projekt, namnge projektet 'codegen'. Syftet är nu *inte* att skapa exekverbara program, vi ska bara studera kodgenereringen. Följande inställningar är lämpliga för projektet:



Observera '-S' flaggan som gör att den genererade assemblerfilen finns kvar efter kompilering. Vi har dessutom tagit bort '-g' flaggan för att slippa se debug-informationen i assemblerfilen.

Slut Uppgift 108

Globala variabler, minnesallokering

Betrakta följande **C**-program bestående enbart av deklarationer av globala variabler:

```
/*
    globals.c
    Deklaration av globala variabler
*/

short      shortint;
long       longint;
int     justint;
int     intvec[10];

struct {
    int     s1;
    char        s2;
    char*       s3;
} komplex;
```

Uppgift 109

Skapa en ny källtext "globals.c" enligt ovanstående. Lägg filen till projektet. Välj

Make

för att kompilera filen.

Nu skapas filen "globals.S" i underbiblioteket "codegen/debug". Öppna denna fil.

Slut Uppgift 109

Först genereras några inledande rader med text där *XCC* skriver ut aktuell version av kompilatorn och diverse information, vi utelämnar dessa rader här.

```
;       1 | short      shortint;
    SEGMENT    bss
_shortint:     RMB     $2
    EXPORT     _shortint [r,2]

;       2 | long       longint;
_longint:      RMB     $4
    EXPORT     _longint  [r,4]

;       3 | int     justint;
_justint:      RMB     $2
    EXPORT     _justint  [r,2]

;       4 | int     intvec[10];
_intvec:       RMB     $14
    EXPORT     _intvec   [r,20]

;       5 |
;       6 | struct {
;       7 |     int     s1;
;       8 |     char        s2;
;       9 |     char*       s3;
;      10 | } komplex;
_komplex:      RMB     $5
    EXPORT     _komplex  [r,5]
```

Eftersom de deklarerade variablerna inte är initierade väljer kompilatorn:

```
SEGMENT   bss
```

som segment för påföljande assemblerdirektiv.

`shortint` är en variabel av typen `short`.

ANSI-C definitionen av datatypen `short`: *"Typen är synonym för: `short int`, `signed short` och `signed short int`. Det är ett heltal med tecken som kan representeras med 16 bitar."*

Efter symbolnamnet `_shortint` har *XCC* placerat direktivet

```
_shortint:  RMB      $2
```

vilket alltså reserverar 2 bytes för variabeln.

Observera också den inledande "understrykningen". Alla globala namn, såväl funktioner som variabler förses med detta av kompilatorn. På så sätt undviks namnkonflikter mellan exempelvis reserverade namn i assemblerspråket och variabler/funktioner som definierats i ett C-program

Eftersom variablerna är deklarerade med global synlighet sker alla referenser till dem med *namn* så att *samtliga* referenser (även från andra källtextfiler) kan lösas upp vid den slutliga länkningen. Det är ju normalt först då som alla globala variabler är kända. Direktivet:

```
EXPORT   _shortint
```

innebär att variabelnamnet får ett *globalt scope* (synlighet). På motsvarande sätt ser vi att variabeln `_longint` av typen `long` tilldelas 4 bytes i segmentet och att variabeln `_justint` av typen `int` tilldelas 2 bytes. För *XCC12* gäller alltså att typen `short` och typen `int` är likvärdiga. Observera dock att detta inte gäller generellt för **C**.

Variabeln `intvec` är en vektor bestående av 10 komponenter, var och en av typen `int`. Följaktligen tilldelas variabeln `_intvec` 10*2 = 20 bytes minnesutrymme.

Variabeln `komplex` är en sammansatt typ `struct`, dvs en datatyp som är komponerad av flera grundläggande typer. Här beräknar kompilatorn det sammanlagda minnesbehovet för variabeln och genererar därefter ett direktiv som åstadkommer detta.

`_komplex` består av

- en variabel `s1` av typen `int`,
- en variabel `s2` av typen `char`
- en variabel `s3` av typen: *pekare till* `char`.

Vi ser att minnesbehovet bestämts till 5 bytes och kan enkelt kontrollera detta:

- Datatypen `int` kräver 2 bytes,
- datatypen `char` kräver 1 byte,
- pekartyper (ovidkommande vad den pekar på) kräver i allmänhet 2 bytes.

Uppgift 110

Översätt följande variabeldeklarationer, givna i programspråket C, till assemblerdirektiv.

```
int             a;
short   int     b;
long    int     c;
char            d;
```

Kontrollera din lösning genom att skapa en källtextfil i C och kompilera denna till en assemblerfil.

Slut Uppgift 110

Uppgift 111

Översätt följande variabeldeklarationer, givna i programspråket C, till assemblerdirektiv.

```
char        cvec[128];
int     ivec[128];
short       sivec[128];
```

Kontrollera din lösning genom att skapa en källtextfil i C och kompilera denna till en assemblerfil.

Slut Uppgift 111

Uppgift 112

Översätt följande variabeldeklarationer, givna i programspråket C, till assemblerdirektiv.

```
struct mystructtype{
    int    previd;
    char      *name;
    int    id;
    short     number;
    int    nextid;
};
struct mystructtype mystruct;
char * char_pointer;
```

Slut Uppgift 112

Synlighet (Scope)

En global variabel är ”synlig” dvs kan refereras från alla delar av programmet oavsett i vilken källtextfil variabeln deklarerats.

För att kompilatorn, vid kompileringstillfället ska veta att en refererad symbol är deklarerad i en annan källtext måste man ange detta med en extern-deklaration.

EXEMPEL

```
extern    int    foo;
main()
{
    ...
    foo = 1;
    ...
}
```

Extern-deklarationen i C ger normalt *inte* upphov till speciella direktiv i assemblerfilen, deklarationen har däremot betydelse för kompilatorns typkontroll.

Om man vill ange att en variabel ska ha samma fysiska egenskaper som en global variabel, dvs en fix adress i datorns minne, men samtidigt vara osynlig utanför den källtext, eller det sammanhang, den deklarerats i använder man lagringsklassen `static`. Variabeln är då åtkomlig från alla funktioner i källtexten men dess namn kommer inte att skickas vidare till länkaren. Detta innebär exempelvis att samma variabelnamn kan förekomma i olika källtexter (`static`-deklarerade) utan att interferera med

varandra. Static-deklarerade variablers namn ersätts under kompileringen med internt genererade symbolnamn som är unika inom källtextfilen.

Uppgift 113

Kompilera följande deklarationer till assembler och studera assemblerfilen. Vilken skillnad upptäcker du?

```
int        a;
static  int b;
```

Slut Uppgift 113

Unsigned och Signed

I C förekommer de reserverade orden `unsigned` respektive `signed` i samband med datatyper. De föregår alltid typdeklarationen och om de utelämnas så förutsätts alltid att `signed` avses. Låt oss se exempel på vad detta innebär för kodgenereringen av ett C-program.

Uppgift 114

Betrakta följande C-program

```
int j,k;
main()
{
if( k < 100 )
    j = 1;
else
    j = 2;
}
```

Kompilera programmet till assemblerkod och identifiera kodutläggningen för if-satsen. Vilken villkorlig branch-instruktion används?

Ändra nu datatypen för `j` och `k` till `unsigned int` och kompilera på nytt till assembler, vilken villkorlig branch-instruktion används denna gång.

Slut Uppgift 114

Typkonverteringar

Typkonverteringar är en viktig del i kompilatorns arbete att översätta C-kod till assemblerkod. I programspråket definieras så kallade *implicita typkoverteringar*, dvs regler för hur kompilatorn ska bete sig vid operationer på variabler av olika typer. Man kan sätta implicit typkonvertering ur spel genom att ange vilken typ

Uppgift 115

Kompilera följande program till assembler och studera assemblerfilen.

```
long    int     la;
short   int     sa,sb;
void main()
{
    /* implicit typkonvertering */
    sa = la;
    /* explicit typkonvertering */
    sa = (short int) la;
}
```

Upptäcker du någon skillnad mellan tilldelningssatserna?

Ändra nu kompilatorns varningsnivå genom att ändra flaggan "-w2" (Project | Settings, *C-Compiler, Options*) till ”-w6”, – kompilera på nytt. Vilket varningsmeddelanden får du? Hur tolkar du meddelandet.

Slut Uppgift 115

Uppgift 116

Kompilera följande program till assembler.

```
unsigned int  ia;
unsigned short int sb,sc;
void   main()
{
   ia = (sb<<2)-sc;
}
```

Studera assemblerfilen, fyll i följande tabell med instruktionsföljden:

ia = (sb<<2)-sc;

Slut Uppgift 116

Lokala variabler, minnesallokering

Utrymme för lokala variabler allokeras annorlunda än för globala variabler. Lokala variabler ska bara finnas under exekvering av den funktion i vilken de deklarerats och det gör det onödigt att placera dem i ett bss-segment eftersom de bara ska vara åtkomliga under en begränsad tid i exekveringen av programmet. Därför allokeras utrymme för lokala variabler på stacken. Då rutinen exekverats färdigt återställs stacken och minnesutrymmet för dessa variabler kan återanvändas under den fortsatta exekveringen.

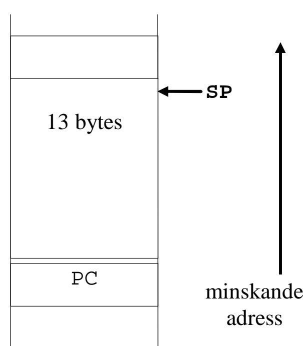
Samma deklarationer som användes tidigare placeras nu i en funktion "main" enligt följande.

```
main() {
    short  shortint;
    long      longint;
    int    justint;

    struct {
        int    s1;
        char      s2;
        char      *s3;
    } typen;
    justint = 0;
}
```

Programmet kompileras och kod genereras.

```
    SEGMENT   text
    EXPORT    _main  [r,2]
_main:
    LEAS   -13,SP
;      2 |     short  shortint;
;      3 |     long      longint;
;      4 |     int    justint;
;      5 |
;      6 |     struct {
;      7 |         int    s1;
;      8 |         char      s2;
;      9 |         char      *s3;
;     10 |     } typen;
;     11 | justint = 0;
    CLRA
    CLRB
    STD    5,SP
;     12 | }
    LEAS   13,SP
    RTS
```

Stackpekarens värde minskas (`LEAS -13,SP`) den totala storleken för de lokala variablerna, dvs. skapar utrymme på stacken. Stackens utseende efter inledningen, eller "prologen" som den också kallas blir nu:



Det totala minnesbehovet för de deklarerade variablerna är alltså 13 bytes vilket vi enkelt kan kontrollera. Variablerna refereras därefter genom att ange en offset relativt till stackpekaren **SP**.

Vid utträde ur funktionen återställs stacken i "epilogen" som helt enkelt är prologens omvända funktion, dvs här adderas värdet 13 till stackpekaren (`LEAS 13,SP`) Överst på stacken ligger nu returadressen för anropet till rutinen.

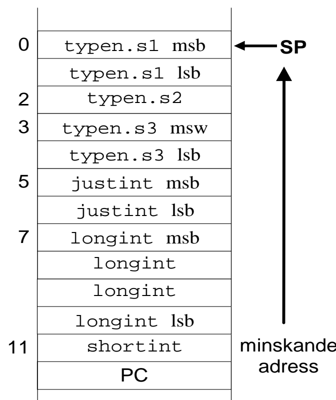
Kompilatorn måste självfallet hålla ordning på var någonstans i aktiveringsposten respektive lokal variabel är placerad. Lägg till tilldelningar i funktionen och kompilera på nytt.

```
main() {
    short  shortint;
    long      longint;
    int    justint;
    struct {
        int    s1;
        char      s2;
        char      *s3;
    } typen;
    shortint = 1;
    longint = 2;
    justint = 3;
    typen.s1 = 4;
    typen.s2 = 5;
    typen.s3 = (char *) 6;
}
```

Följande assemblerkod genereras av kompilatorn:

```
    ( Prolog )
;     11 |     shortint = 1;
    LDD    #1
    STD    11,SP
;     12 |     longint = 2;
    LDY    #0x0000
    LDD    #0x0002
    STY    7,SP
    STD    9,SP
;     13 |     justint = 3;
    LDD    #3
    STD    5,SP
;     14 |     typen.s1 = 4;
    LDD    #4
    STD    0,SP
;     15 |     typen.s2 = 5;
    LDAB   #5
    STAB   2,SP
;     16 |     typen.s3 = (char *) 6;
    LDD    #6
    STD    3,SP
;     17 | }
    ( Epilog)
```

Följande figur visar hur aktiveringsposten skapats på stacken och var `SP` pekar. Av figuren framgår också de lokala variablernas position. Observera ordningen i vilken variablerna tilldelas offset, den sist deklarareade variabeln har lägst offset, den först deklararade variabeln har störst offset.



Uppgift 117

Antag att en funktion deklarerats på följande sätt:

```
void    main( void )
{
    int     a;
    short   int     b;
    char    c;

    a = 1;
    b = 2;
    c = 3;
}
```

Beskriv hur tilldelningarna översätts till assemblerkod av *XCC*. Kontrollera ditt svar genom att kompilera till assemblerkod.

a = 1;
b = 2;
c = 3;

Slut Uppgift 117

Anropskonventioner (Parameteröverföring)

Detta moment handlar om hur *XCC* översätter funktionsanrop och hur rutiner skrivs i assemblerspråk för att fungera tillsammans med rutiner skrivna i **C**.

Tidigare har vi visat hur variabler, såväl globala som lokala hanteras. När det gäller överföring av parametrar kan detta liknas vid lokala variabler, dvs deras "livslängd" begränsas av den tid (under exekvering) som den anropade funktionen använder sig av dom. Parametrar överförs via stacken och det gäller för den anropade funktionen (subrutinen) att korrekt referera sina parametrar.

Allmänt gäller för *XCC* att listan av parametrar i ett funktionsansrop behandlas "bakifrån". Betrakta följande exempel på funktionsanrop:

EXEMPEL

Funktionsanrop

```
int    a,b;
void   main( void )
{
    callfunc( a,b );
}
```

För funktionsanropet genererar *XCC12* följande kod:

```
;      4 |     callfunc( a,b );
    LDD    _b
    PSHD
    LDD    _a
    PSHD
    JSR    _callfunc
    LEAS   4,SP
```

Vi ser hur kompilatorn genererar kod för att:

- placera värdet av variabeln "b" på stacken
- placera värdet av variabeln "a" på stacken
- utför anropet av funktionen "callfunc"
- adderar 4 bytes till stackpekaren, dvs återställer denna

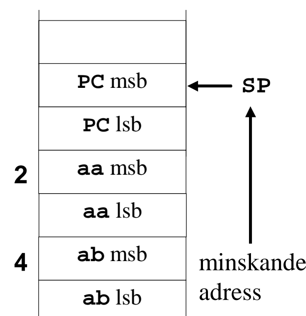
Vid `JSR` läggs återhoppsadressen (2 bytes) på stacken och i subrutinen `_callfunc` kan vi enkelt bestämma den offset från stackpekaren som gäller för de överförda parametrarna. Vi visar detta genom att låta kompilatorn generera kod för "dummy"-funktionen `callfunc`.

EXEMPEL

```
callfunc( int aa , int ab )
{
    aa = 1;
    ab = 2;
}
```

XCC12 genererar följande kod:

```
    SEGMENT   text
    EXPORT    _callfunc [r,2]
_callfunc:
;      2 | {
;      3 |     aa = 1;
    LDD    #1
    STD    2,SP
;      4 |     ab = 2;
    LDD    #2
    STD    4,SP
;      5 | }
    RTS
```



Returvärden från funktioner

För funktioner som returnerar ett värde gäller vanligtvis att detta värde finns i register **D** efter funktionsanropet. Undantaget är följande fall:

- Funktionen returnerar `char`, returvärdet finns då i register **B**.
- Funktion returnerar `long`, returvärdet finns i registerparet **Y/D** med msw i **Y** och lsw i **D**.
- Funktion returnerar `float`, `double` eller `long double`, alla typerna behandlas som float i *XCC12*, returvärdet finns i registerparet **Y/D** med msw i **Y** och lsw i **D**.
- Funktion returnerar `struct`, utrymme för returvärde allokeras av anropande funktion.

Uppgift 118

Översätt följande funktionsanrop till assemblerkod.

```
do_nothing();
```

där följande deklaration har gjorts:

```
void do_nothing(void);
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

Slut Uppgift 118

Uppgift 119

Översätt följande funktionsanrop till assemblerkod. Register **D** används för returvärde**.**

```
result = do_something();
```

där följande deklarationer har gjorts:

```
int     do_something(void);
int     result;
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

Slut Uppgift 119

Uppgift 120

Översätt följande funktionsanrop till assemblerkod. Ange också hur parametrar refereras i den anropade funktionen. Register **D** används för returvärde.

```
result = max(low,high);
```

där följande deklaration har gjorts:

```
int     result,low,high;
int     max( int , int );
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

Slut Uppgift 120

Uppgift 121

Översätt följande funktionsanrop till assemblerkod. Register **D** används för returvärde. I denna uppgift skickas en vektor som parameter. Observera dock att inte hela vektorn ska placeras på stacken utan endast en *pekare* till vektorns första element (C-konvention).

```
size = scalar( pvec , PVECSIZE );
```

där följande deklaration har gjorts:

```
#define PVECSIZE   32
int     pvec[PVECSIZE];
int     size;
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

Slut Uppgift 121

32-bitars aritmetik

Med *XCC12* följer ett färdigt programbibliotek för 32-bitars operationer. Biblioteket heter `libcc` (*C-Compiler library*). Källtexter till kompilatorbiblioteken `libcc` hittar du i `{INSTALL}\src\libcc\common`

Då vi vill utföra aritmetik på datatypen `long` (eller `float`) måste vi använda två 16-bitars register för returvärden.

Låt oss illustrera användningen av funktioner i `libcc` med följande exempel:

EXEMPEL

Betrakta följande tilldelning:

```
long  la,lb,lc;
main()
{
   la = lb + lc;
}
```

För tilldelningen genererar XCC12 kod enligt följande:

```
;     5 |    la = lb + lc;
   LDY    _lc
   LDD    2+_lc
   PSHD
   PSHY
   LDY    _lb
   LDD    2+_lb
   PSHD
   PSHY
   JSR    add32
   LEAS   8,SP
   STY    _la
   STD    2+_la
```

Observera hur registerparet `Y`/`D` används för att placera parametrarna på stacken.

Notera även att funktionen **`add32`** definierats utan '_'-konventionen. Detta görs för att undvika risken för namnkonflikt med användardefinierade funktioner.

Efter **`add32`** finns resultatet av additionen i registerparet **Y**/**D**.

Motsvarande hantering gäller för flyttal. Du kan läsa mer om `libcc` i *XCC's* hjälpsystem (*Libraries | Compiler Libraries*).

Tillämpning: ML4

Vi ska nu ge ett litet, men komplett, exempel på hur vi kombinerar källtexter skrivna i assembler respektive C. För exemplet använder vi in- och utmatning med *ML4*.

Dina uppgifter blir därefter att redigera dessa källtexter, kompilera och testa funktionen.

Vi börjar med att beskriva C-källtexten som utformats så att alla rutiner (i assemblerkälltexten) används, helt enkelt ett testprogram.

```
/*
   ML4TEST.C
*/
void   main( void )
{
   unsigned char c;
   while( 1 )
   {
      c = ML4_DipSwitch();
      c = c << 1;
      ML4_Diodes( c );
   }
}
```

Testprogrammet får väl anses vara tämligen självdokumenterande.

Vi ska nu implementera rutinerna `ML4_DipSwitch` och `ML4_Diodes` i assemblerkod.

```
;
; ml4dr.s12
; rutiner för in-/utmatning med ML4
;
ML4_INPORT      EQU     $0600
ML4_OUTPORTEQU      $0400

; Programkod placeras i 'text'-segmentet
    SEGMENT     text

;**********************************
; C-interface:
; unsigned char ML4_DipSwitch( void );
;
    DEFINE      _ML4_DipSwitch
    FUNCTION    _ML4_DipSwitch,End_DipS

_ML4_DipSwitch:

; Rutinen ska returnera värdet i D
; I detta fall används endast minst
; signifikanta delen (ack B)
    LDAB        ML4_INPORT
    RTS
End_DipS

;*************************************
;   C-interface:
;   void ML4_Diodes(unsigned char);
;
    DEFINE      _ML4_Diodes
    FUNCTION    _ML4_Diodes,End_Diode
_ML4_Diodes:
; Värdet som ska skrivas ut kommer som
; en parameter på stacken ...
; Anm. Register X,Y och D betraktas som
; arbetsregister av kompilatorn.
; Dessa behöver därför INTE sparas
; även om de används...
    LDAB        2,SP          ;parametern
    STAB        ML4_OUTPORT   ; skriv...
    RTS
End_Diode
```

Uppgift 122

- Skapa ett nytt projekt `ML4-SIMPLE`, använd föreslagna standardinställningar.
- Redigera två nya filer, `ML4TEST.C` och `ML4DR.s12` enligt de givna exemplen och lägg filerna till projektet.
- Testa programmet och kontrollera funktionen.

Slut Uppgift 122

Inbäddad assemblerkod

”Inbäddad assemblerkod” betyder i princip att man skriver sitt assemblerprogram i en C-källtext. Det är viktigt att påpeka att detta inte är en ANSI standard och således inte kan förväntas fungera likadant i olika programutecklings-miljöer och under olika kompilatorer. Samtidigt bör det sägas att kompilatorer som tillåter inbäddad assemblerkod aldrig kontrollerar koden som anges. Detta betyder att man alltså kan skapa ett program som kompileras korrekt men som trots detta genererar felmeddelanden vid assembleringen.

För att kunna skapa program, där inbäddad assemblerkod förekommer, måste man alltså vara väl förtrogen med den använda utvecklings-miljöns assemblator, man måste dessutom kunna de konventioner kompilatorn tillämpar.

I *XCC* kan assemblerkod “bäddas in” i C-källtexten genom att använda följande konstruktion

`__asm("assemblertext");`

assemblertext kopieras direkt till assemblerfilen.

Följande exempel visar hur funktionerna `ML4_DipSwitch` och `ML4_Diodes` kan implementeras med hjälp av inbäddad assemblerkod, snarare än som i föregående uppgift, i en separat assemblerkälltext.

```
/*
  ML4EMBA.C
  Illustrerar inbäddad assemblerkod
*/

unsigned char ML4_DipSwitch( void )
{
    __asm(" LDAB $0600");
}

void ML4_Diodes( unsigned char c )
{
    __asm(" LDAB 2,SP");
    __asm(" STAB $0400");
}
```

Du kan också använda en alternativ form av `__asm`, där du låter *XCC12* översätta till assemblerinstruktionernas operander. Detta ger ofta mer lättläst och överskådlig kod.

```
/*
   ML4EMBA.C
   Illustrerar inbäddad ssemblerkod
*/
#define   ML4IN     0x600
#define   ML4OUT    0x400

unsigned char ML4_DipSwitch( void )
{
    __asm(" LDAB %a", ML4IN );
}

void ML4_Diodes( unsigned char c )
{
    __asm(" LDAB %a", c);
    __asm(" STAB %a", ML4OUT);
}
```

Den slutgiltiga assemblerkoden kan du studera om du kompilerar med '-S'-flaggan, funktionerna översätts till följande kod:

```
_ML4_DipSwitch:
   LDAB   $0600
   RTS

_ML4_Diodes:
   LDAB   2,SP
   STAB   0x400
   RTS
```

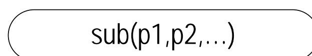
Rätt använt, ger inbäddad assemblerkod möjlighet att implementera funktioner på ett mycket effektivt sätt vare sig det gäller kodstorlek eller prestanda. Vi måste dock komma i håg att metoden är tveksam då det gäller skalbarhet (har vi skrivit de optimala instruktionerna för den använda processorn?) såväl som portabilitet (fungerar den inbäddade koden under en *annan* kompilator).

Appendix

Appendix A: MC12 IO-adresser för laborationskort

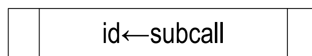
Laborationskort Noter	Register/Port Symboliska namn	Adress (hexadecimal form)
ML4		
"Borrmaskin" är också ansluten till dessa adresser. OBS: Skillnad mellan IO-simulator och fysisk hårdvara.	Out	0400
	In	0600
ML5		
De här angivna adresserna för ML5 gäller PAL-revision 2.	Out	0C00
	In	0C01
	Out	0C02
	Out	0C03
ML13		
	Ctrl/Status	0B00
	IRQ Ctrl/Status	0B01
ML15		
	Kbd Data	09C0
	Kbd Status	09C1
	Led Mode	09C2
	Led Ctrl/Data	09C3
ML19		
	Status	0DC0
	Kvittera ev 1	0DC2
	Kvittera ev 2	0DC3

Appendix B: Symboler i flödesdiagram



Inträde i och utträde ur subrutiner.

Inträde, typiskt med subrutinens namn och symbolisk representation av eventuella parametrar då sådana finns.
Utträde, ("RETUR") typiskt med angivande av returnvärde (rv) om sådant finns.

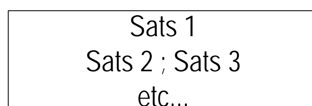


Subrutinanrop.

Med parametrar, symbolisk representation av eventuella aktuella parametrar som skickas med subrutinanropet.
Med returvärde, tilldelningsoperator placeras framför den anropade subrutinen.

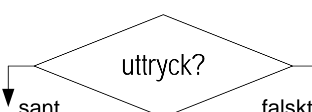
Engångsinitieringar.

Placeras typiskt i omedelbar anslutning till en inträdessymbol



Exekveringsblock.

En eller flera satser som ordnats och exekveras sekvensiellt.



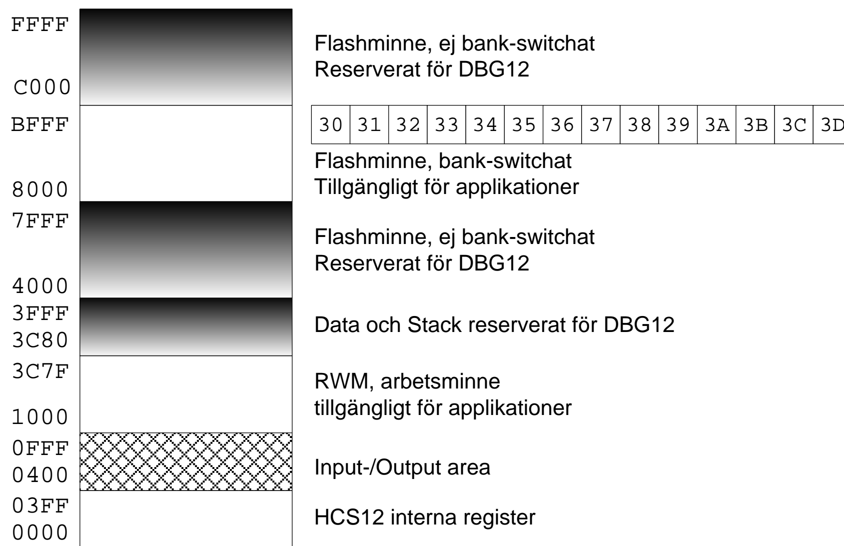
Villkorsblock.

Ett uttryck testas, utfallet kan vara sant eller falskt och exekveringsväg väljs därefter.

Appendix C: MC12/dbg12 minnesdisposition

Detta appendix beskriver den fullständiga, bank-switchade, minnesmodellen för *MC12/DBG12*.

Adressrummet disponeras på följande sätt:



DBG12 använder internt RWM 3D00-3FFF. Applikationsprogram kan använda intervallet 1000 t.o.m. 3CFF.

DBG12 använder "naturlig" översättning av adresser i det bankade minnet. Detta innebär att de 6 mest signifikanta bitarna i adressen används direkt för att initiera PAGE-registret. Undantag är bank 30, som adresseras om du bara använder 16-bitars adresser (8000-BFFF).

MC12 ger möjlighet att använda maximalt 224 kByte expanderat minne.

Appendix D: Motorola S-format

S-formatet är ett sätt att överföra program och data mellan olika datorer, ofta via en enkel serielänk. S-formatet innehåller endast ASCII tecken vilket innebär att det enkelt kan inspekteras och även redigeras. All representation i S-formatet är på hexadecimal form, dvs tecknen 0-9 representerar sina decimala motsvarigheter, 'A' motsvarar 10, 'B' motsvarar 11 osv till 'F' som motsvarar 15. Dessutom används ASCII 'S' för att markera postens första tecken.

Sx-post

En Sx-post består av en sekvens ASCII tecken avslutade med <NL>, dvs en text-sträng om en rad. Raden innehåller maximalt 5 olika fält enligt följande struktur:

TYP LÄNGD ADRESS KOD/DATA K-SUM

Fältnamn	*Storlek*	*Beskrivning*
TYP:	2	S-post typ, dvs "S0", "S1", "S2" osv.
LÄNGD:	2	Antalet ASCII-par i posten, *TYP* och *LÄNGD* fält är ej inräknade
ADRESS:	4,6 eller 8	Startadress för kod/data från *KOD/DATA* fältet För S1-post är adressen 16 bitar, dvs 4 ASCII tecken. För S2-post är adressen 24 bitar, dvs 6 ASCII tecken. För S3-post är adressen 32 bitar, dvs 8 ASCII tecken.
KOD/DATA:	0-2n	Innehåller från 0 upp till 2n bytes exekverbar kod eller data som skall laddas i måldatorns minne.
K-SUM:	2	Innehåller en kontrollsumma som används av det mottagande systemet för att verifiera att inget fel uppstått under överföringen.K-sum beräknas som: Ett-komplementet av summan från Längd-,Adress och kod/data-fält. Vid summeringen används modulo 8 addition

Sx-post typ

En Sx-post kan vara en av följande typer:

S0 Indikerar "startblock", posten innehåller ingen kod/data utan används för att mottagande sida ska förbereda laddning.

S1 Typen innehåller kod/data som kan laddas av system med 16-bitars adressrum. Typiskt gäller detta Motorolas 8-bitars mikroprocessorer/mikrocontrollers.

S2 Typen innehåller kod/data som kan laddas av system med 24-bitars adressrum. Typiskt används posten för det ”bankade” minnet i HCS12.

S3 Typen innehåller kod/data som kan laddas av system med 32-bitars adressrum. Används inte tillsammans med HCS12

.

S7 Indikerar "slutblock" för överföring av S3-poster.

S8 Indikerar "slutblock" för överföring av S2-poster.

S9 Indikerar "slutblock" för överföring av S1-poster.

Sx-fil, exempel

Följande exempel visar hur en programsekvens översatts till S2-format. I exemplet antas att första instruktionen i sekvensen startar på adress $7000.

```
*   Programsekvens ..
    ...
    ...
    move.l   #$feedc0de,($68000).l
    move.l   #$aabbccdd,($68020).l
    move.l   #$11223344,($68030).l
    ...
    ...
```

Den resulterande ".S2" -laddfilen kommer att innehålla följande poster:

```
S004000000FB
 ....
S22200700023FCFEEDC0DE0006800023FCAABBCCDD0006802023FC1122334400068030ED
 ...
S80400701E6D
```

Dvs, inleds med en S0-post enligt:

Typ Längd Adress K-sum
S0 04 000000 FB
där:

Typ-fältet anger "startblock"
Längd-fältet anger postens längd i antal bytes. I detta fall 4 bytes.
Adress-fältet anger adress 0, vilket är betydelselöst eftersom posten ej innehåller kod/data.
K-sum-fältet innehåller en kontrollsumma som beräknats på Längd, Adress och kod/data fält.

Observera att Längd-fältet anger det antal bytes som posten innehåller. Eftersom en byte kräver två ASCII-tecken blir antalet ASCII tecken (i Adress- och Längd- fält) dubbelt så många, dvs 8.

Efter S0-posten följer en S2-post som är indelad enligt:

```
S2 22 007000 23FCFEEDC0DE0006800023FCAABBCCDD0006802023FC1122334400068030 ED
```

Dvs *Typ*-, *Längd*-, *Adress*-, *Kod/data*, och *K-sum fält*. Vi koncentrerar oss nu på kod/data-fältet. De övriga fälten i posten har samma betydelse som för S0-posten.

Kod/data-fältet ska placeras på adress $7000 enligt Adress-fältet. Om vi jämför med programsekvensen ovan ser vi att kod genererats enligt:

```
23FCFEEDC0DE00068000    move.l   #$feedc0de,($68000).l

23FCAABBCCDD00068020    move.l   #$aabbccdd,($68020).l

23FC1122334400068030    move.l   #$11223344,($68030).l
```

En Sx-laddfil avslutas alltid med en S7, S8 eller S9 post. I detta fall, där filen var av S2-typ skall den alltså avslutas med en S8-post:

```
S8 04 00701E      6D
```

dvs posten består, precis som S0-posten av *Typ*-, *Längd*-, *Adress*- och *K-sum* fält. Posten markerar "slutblock" för den mottagande datorn.

Appendix E: ASCII representation

__A__merican __S__tandard __C__ode for __I__nterchange of __I__nformation.

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
0	NUL	20		40	@	60	`
1	SOH	21	!	41	A	61	a
2	STX	22	"	42	B	62	b
3	ETX	23	#	43	C	63	c
4	EOT	24	$	44	D	64	d
5	ENQ	25	%	45	E	65	e
6	ACK	26	&	46	F	66	f
7	BEL	27	'	47	G	67	g
8	BS	28	(	48	H	68	h
9	HT	29	)	49	I	69	i
A	LF	2A	*	4A	J	6A	j
B	VT	2B	+	4B	K	6B	k
C	FF	2C	,	4C	L	6C	l
D	CR	2D	-	4D	M	6D	m
E	SO	2E	.	4E	N	6E	n
F	S1	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[Ä	7B	{ ä
1C	FS	3C	<	5C	\ Ö	7C	| ö
1D	GS	3D	=	5D	] Å	7D	} å
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Förklaring av ASCII 01-1F

ACK	Acknowledge	GS	Group Separator
BEL	Bell	HT	Horizontal Tabulation
BS	Backspace	LF	Line Feed
CAN	Cancel	NAK	Negative Acknowlege
CR	Carriage Return	NUL	Null
DC	Device Control	RS	Record Separator
DEL	Delete	SI	Shift-In
DLE	Data Link Escape	SO	Shift-Out
EM	End of Medium	SOH	Start of Heading
ENQ	Enquiry	SP	Space
EOT	End of Transmission	STX	Start of Text
ESC	Escape	SUB	Substitute
ETB	End of Transmission Block	SYN	Synchronous Idle
ETX	End of Text	US	Unit Separator
FF	Form Feed	VT	Vertical Tabulation
FS	File Separator		

Appendix F: Exceptionvektorer

ROM adress	MC12 adress	Funktion
FFFE	3FFE	RESET, Startvektor
FFFC	3FFC	Clock Monitor Fail, (ej impl i simulator)
FFFA	3FFA	COP Watchdog Timeout, (ej impl i simulator)
FFF8	3FF8	Illegal Op Code (ej impl i simulator)
FFF6	3FF6	SWI
FFF4	3FF4	XIRQ
FFF2	3FF2	IRQ
FF00-FFF0	3F00-3FF0	Enhetsspecifika vektorer, skiljer sig något beroende på de olika varianterna (se nedan)

ROM adress	MC12 adress	MC9S12DG128B/256B
FFF0	3FF0	Real Time Interrupt
FFEE	3FEE	Enhanced Capture Timer channel
FFEC	3FEC	Enhanced Capture Timer channel 1
FFEA	3FEA	Enhanced Capture Timer channel 2
FFE8	3FE8	Enhanced Capture Timer channel 3
FFE6	3FE6	Enhanced Capture Timer channel 4
FFE4	3FE4	Enhanced Capture Timer channel 5
FFE2	3FE2	Enhanced Capture Timer channel 6
FFE0	3FE0	Enhanced Capture Timer channel 7
FFDE	3FDE	Enhanced Capture Timer overflow
FFDC	3FDC	Pulse accumulator A overflow
FFDA	3FDA	Pulse accumulator input edge
FFD8	3FD8	SPI0
FFD6	3FD6	SCI0
FFD4	3FD4	SCI1
FFD2	3FD2	ATD0
FFD0	3FD0	ATD1
FFCE	3FCE	Port J
FFCC	3FCC	Port H
FFCA	3FCA	Modulus Down Counter underflow
FFC8	3FC8	Pulse Accumulator B Overflow
FFC6	3FC6	PLL lock
FFC4	3FC4	CRG Self Clock Mode
FFC2	3FC2	Används ej (BDLC)
FFC0	3FC0	IIC Bus
FFBE	3FBE	SPI1
FFBC	3FBC	Reserverad
FFBA	3FBA	EEPROM I-Bit
FFB8	3FB8	FLASH I-Bit
FFB6	3FB6	CAN0 wake-up
FFB4	3FB4	CAN0 errors
FFB2	3FB2	CAN0 receive
FFB0	3FB0	CAN0 transmit
FFAE	3FAE	Används ej (CAN1 wake-up)
FFAC	3FAC	Används ej (CAN1 errors)
FFAA	3FAA	Används ej (CAN1 receive)
FFA8	3FA8	Används ej (CAN1 transmit)
FFA6	3FA6	Används ej (ByteFlight Rx FIFO not empty)
FFA4	3FA4	Används ej (ByteFlight receive)
FFA2	3FA2	Används ej (ByteFlight general)
FFA0	3FA0	Används ej (ByteFlight Synchronisation)
FF9E	3F9E	Reserverad
FF9C	3F9C	Reserverad
FF9A	3F9A	Reserverad
FF98	3F98	Reserverad
FF96	3F96	CAN4 wake-up
FF94	3F94	CAN4 errors
FF92	3F92	CAN4 receive
FF90	3F90	CAN4 transmit
FF8E	3F8E	Port P Interrupt
FF8C	3F8C	PWM Emergency Shutdown
FF8A	3F8A	Reserverad
FF88	3F88	Reserverad
FF86	3F86	Reserverad
FF84	3F84	Reserverad
FF82	3F82	Reserverad
FF80	3F80	Reserverad

Appendix G: XCC objektfilsformat

XCC använder en speciell assemblator (*RA*) som skiljer sig något från den variant (*QA*) som används under *ETERM*. Det finns flera typer av RA-assemblatorer (för olika mikroprocessorer/ mikrocontrollers) och det som sägs i detta appendix är gemensamt för de olika typerna.

Vid assemblering av en källtextfil skapar RA-assemblatorerna en objektfil. Dessa innehåller en intern representation på ASCII-format, vilket innebär att objektfilerna enkelt kan inspekteras och till och med redigeras med en vanlig texteditor. Objektfilerna är ej avsedda för överföring via serielänkar mellan olika system och innehåller därför inga kontrollsummor. Däremot har alla poster i objektfilen ett exakt format där speciella ASCII-tecken har en precis betydelse och inte utan vidare kan ersättas.

En speciell typ av poster skapas av '.stab'-direktivet. Dessa är enbart avsedda för information till källtextdebuggern, de behandlas inte här. I objektfiler inleds sådana rader med 'SY'. Du kan ignorera dessa.

Varje rad i en objektfil utgör en *post*. Varje post består av ett antal *fält*. Det första, eller eventuellt de två första tecknen i raden anger *postens typ*. Separation mellan fälten i en post markeras med "horisontal tab",`<TAB>` eller något annat specialtecken. Varje post avslutas med en radslutsmarkering, `<NL>`. Flera olika posttyper kan förekomma:

- *Generella posttyper*, dessa förekommer alltid, och i varje modul.
- *Initierade segment*, dessa poster förekommer praktiskt taget alltid, men inte nödvändigtvis. De representerar den maskinkod och de initierade data som genererats av RA-assemblatorn.
- *Symbol poster*, varje post representerar *en* symbol som deklarerats som global, med ett DEFINE-direktiv.
- *Relokeringskommandon*, dessa poster representerar referenser till symboler i assemblerkälltexten. Symbolerna kan vara lokala, dvs definierade och refererade i samma modul, men *inte* refererade från någon annan modul, eller globala dvs refererade från, eventuellt, flera moduler. För referenser till lokala, respektive globala symboler genereras olika relokeringskommandon. Ett relokeringskommando bär information om *vilken offset* (eller vilken *symbol*) som refererats samt *varifrån* referensen görs.

I detta appendix beskrivs de olika posttyperna gruppvis, därefter ges exempel på hur olika poster skapas av RA-assemblatorn och slutligen beskrivs länkningsförfarandet.

Generella poster

De generella posterna (3 olika) finns i varje objektmodul.

Varje objektmodul inleds med:

`m:`*modulnamn*:*objekttyp*:*objektfil*`<NL>` start av objektmodul med namnet *modulnamn*

Objektmodulen ärver namnet från den objektfil den ingår i. Exempelvis vid assemblering av filen `test.s12`, skapas en objektfil `test.o12`, modulnamnet blir då `test`. Objekttypen blir `O12`.

Dessutom förekommer poster för de olika segmenten som ingår i modulen:

`seg`:*segmentsnamn*:*storlek*`<NL>`

segmentsnamn är ett unikt namn som definierar kod/data som skall grupperas tillsammans vid den slutliga länkningen. Varje segment kan ha en användardefinierad startadress. XCC använder standardsegmenten INIT, TEXT, DATA, RODATA, BSS, och ABS (se nedan) men andra namn är tillåtna.

stl anger antalet bytes i modulens text-segment.Storleken (antal bytes) anges här på decimal form.

En modul avslutas alltid med:

`e<NL>` slut av objektmodul

Speciella segment
Generellt gäller att ett godtyckligt namn accepteras som segment förutsatt att namnet innehåller maximalt 16 tecken som kan vara a-z, A-Z och 0-9.
XCC kompilator och länkare använder dock några fördefinierade segmentsnamn för att underlätta handhavandet, se följande tabell.

Namn	*Kommentar*
`INIT`	Av konvention används detta segmentsnamn för den kod som alltid inleder programmet. Observera att det är olämpligt att använda segmentet i mer än en källtext eftersom detta kan resultera i att segmentet kombineras i fel ordning och programmets startadress blir fel. Dessutom används ofta symbolnamnet `__start` för att debuggern ska kunna identifiera programmets startpunkt. Observera att segmentet inte är nödvändigt eftersom det räcker med *symbolen* för programmets startpunkt. Det visar sig dock praktiskt att använda ett specifikt segmentsnamn för startupkod och att låta en specifik symbol representera segmentets första exekverbara instruktion. Du kan läsa mer om segment i hjälpsystemet under ”Linker/Librarian, Script Files”
`TEXT`	Standardsegment för programkod. Kodutläggning till TEXT segmentet kan ändras med kompilator-pragma, exempelvis: `#pragma TEXT mytextsegment`
`DATA`	Standardsegment för initierade data. Kodutläggning till DATA segmentet kan ändras med kompilator-pragma, exempelvis: `#pragma DATA mydatasegment`
`RODATA`	Standardsegment för konstant data. Kodutläggning till `RODATA` segmentet kan ändras med kompilator-pragma, exempelvis: `#pragma RODATA myrodatasegment`
`BSS`	Standardsegment för programkod. Kodutläggning till BSS segmentet kan ändras med kompilator-pragma, exempelvis: `#pragma BSS mybsssegment`
`ABS`	Konstantsegment, symboler som definierats i detta segment påverkas inte av relokering, dvs de behåller sitt värde genom adressbindningen. Används exempelvis för att definiera portadresser i måldatorns primärminne.

Allmänt gäller att segments-pragmas ska användas med försiktighet. Följande tumregel bör tillämpas:

Placera segments-pragmas före kod- och datadeklarationer i källtextfilen.

Segment i utökat (”paged”) minne
XCC tillåter att man använder HCS12’s utökade minne för program. Eftersom adressinformationen för kod som finns i det utökade minnet är 24 bitar (snarare än 16) krävs speciell hantering av dessa funktioner. Du kan läsa mer om detta i XCC’s hjälpsystem.

Initierade segment
För att representera segmentens innehåll används C-poster:

`C:`*segmentsnamn*`<TAB>`*kod*`<NL>`

anger ett konsekutivt block. Posten innehåller ett block av segmentet (hexadecimal form). C-posterna läses konsekutivt och den inbördes ordningen i objektmodulen kan inte ändras.

Den kod som innefattas av modulens `C`-poster har genererats som om respektive segment startar på adress 0. All information som krävs för att "flytta" dessa startadresser (relokera) finns i form av relokeringskommandon (beskrivs nedan). I `C`-poster kan man därför i bland upptäcka långa strängar av nollor. Detta är ett resultat utav att RA-assemblatorerna inte kodar *några* absoluta adresser i `C`-poster. Det är länkarens uppgift att ersätta dessa adresser först då den *slutliga* basadressen för respektive segment är känd.

Symbolposter
För varje symbol som deklarerats global (med DEFINE-direktivet) genereras en symbolpost. Symbolposten registreras av länkaren i en speciell symboltabell, och alla referenser till denna symbol kan därefter lösas upp. Observera att *globala* symboler refereras med namn. En viktig skillnad mellan *namn*-refererade symboler och *offset*-refererade symboler framhävs exempelvis av lagringsklassen "static" i programspråket C. "static" innebär att symbolen är global i den aktuella källtextfilen, men inte ska exporteras till andra källtextfiler. Följaktligen är en sådan symbol endast "synlig" under

kompilering/assemblering av den aktuella källtextfilen. I en *annan* källtextfil kan man därför deklarera *samma* symbolnamn, utan att detta förvirrar länkaren. Motsvarigheten i assemblerkälltexter till detta fenomen i C-program, är att använda, eller inte använda DEFINE-direktivet. Genom att definiera en symbol global (DEFINE symbol) i assemblerkälltexten, har man uteslutit nyckelordet `static`, dvs, detta symbolnamn kan inte definieras som globalt i en annan källtext. Självfallet kan dock samma namn användas lokalt för någon annan källtext (det är ju det samma som en "static" deklaration).

För referenser till lokala respektive globala symboler genererar RA-assemblatorerna olika typer av *relokeringskommandon* (se nedan).

En *symbolpost* har följande utseende:

`G:`*segment*:*modul*:*adress*:*namn*`<NL>`

G*-posten* representerar en symbol som deklarerats global i modulen med DEFINE direktivet. Symbolen kan då refereras från andra moduler.

- ***segment*** är det segment som symbolen deklarerats i.
- ***modul*** är modulnamnet för den källtextfil symbolen deklarerats i
- ***adress*** är offseten i symbolens segment
- ***namn*** är symbolens namn.

Relokeringskommandon

För varje *symbolreferens* genereras ett *relokeringskommando*. Relokeringskommandot är av olika typ beroende på om referensen avser en *lokal* eller en *global* symbol. Ett lokalt relokeringskommando `RL`, genereras om den refererade symbolen *inte* är EXTERN-deklarerad och heller *inte* är definierad global med DEFINE. För globala referenser genereras relokeringskommandot `RG`.

I ett lokalt relokeringskommando har RA-assemblatorn kastat bort informationen om symbolens namn. Det enda som nu behövs är uppgift om symbolens segment, och symbolens offset i detta segment. Detta är uppenbarligen känt vid assembleringen av källtexten, eftersom den refererade symbolen finns i samma källtext. Jämför detta med betydelsen av C-nyckelordet `static`. Ett relokeringskommando kan ha följande utseende:

`RL:`*segment*:*modul*:*offset*:*typ*:*längd*:*refoffset*:*refsegment*`<NL>`
anger referens till lokal symbol
eller:
`RG:`*segment*:*modul*:*offset*:*typ*:*längd*:*refsym*`<NL>`
anger referens till global symbol

- ***segment*** anger det segment som referensen finns.
- ***modul*** anger den modul där referensen finns.
- ***offset*** anger var, i segmentet referensen är placerad.
- ***typ*** anger om den substituerade adressen ska beräknas med offset till segmentets början (`O`), vilket är det vanligaste. `P`, i detta fält anger att den substituerade adressen ska bestämmas relativt **PC**.
- ***längd*** anger om 1,2 eller 4 bytes (`B`,`W` eller `L`) ska substitueras, dvs referensens storlek

För referenser till globala symboler (RG)

- ***refsym*** anger den refererade symbolen.

För referenser till lokala symboler:

- ***refoffset*** anger offseten till den refererade symbolen.
- ***refsegment*** anger det segment den refererade symbol definierats i.

Appendix H: XCC skript filer

I *XCC* kan man använda "scripts" för att ge direktiv om hur kod/data etc. ska placeras i måldatorns minne. *Group*-direktivet används för att definiera kontinuerliga sektioner (grupper) bestående av olika segment.

Exempel ”Group”-direktivet

Följande direktiv kan användas för att samla i hop alla de standardsegment som används av kompilatorn till en enda grupp och samtidigt namnge gruppen ”test_group”:

```
group( rw , test_group)
{
    init,
    text,
    rodata,
    data,
    bss
}
```

I deklarationens parentes anges först gruppens ”attribut”, i detta fall `rw` (*read/write*). Attributen används för att länkaren ska kunna kontrollera konsistens mellan olika deklarationer, mer om detta kommer senare. I parentesen anges också gruppens namn, i detta fall `test_group`, detta namn används enbart vid länkningsproceduren och gör att vi senare kan referera till gruppen som *en* enhet.
Mellan ”hakarna” specificeras sedan de segment som ingår i gruppen. Observera att dessa namn är unika i hela applikationen, exempelvis anger här ”text” samtliga applikationens text-segment. Den inbördes ordningen i denna uppräkning behålls också i gruppen. Notera speciellt att vi placerar ”init”-segmentet först för att vi sedan enkelt ska kunna identifiera applikationens startadress.

Efter att ha placerat samtliga segment i någon grupp använder vi *Layout*-direktivet för att associera fysiskt minne till grupper. En sådan minnestilldelning är alltså specifik för den använda måldatorn.

Exempel Layout för applikation i RWM

För exempelvis laborationsdatorn MC12 kan följande direktiv användas för ett fall där vi endast vill utnyttja måldatorns tillgängliga RWM-minne för vårt färdiga program.

```
layout
{
    0x1000,0x3C80 <= test_group
}
```

Tillgängligt fysiskt RWM är $1000-$3FFF, men intervallet $3C80-$3CFF används av MC12’s inbyggda monitor/debugger DBG12.

Attribut används i länkar-scripten för att ange vilken typ av minne som avses. Följande attribut används:

`r` : minnet kan läsas

`w` : minnet kan skrivas

`e` : minnet kräver ”bank-switching”

Memory-direktivet kan, men behöver inte finnas. Direktivet används för att ange tillgängligt fysiskt minne och minnesattribut för den använda måldatorn. Syntaxen är:

```
memory ( attribut , basadress, storlek i bytes )
```

Exempel *Memory*-direktiv för MC12

Följande direktiv specificerar minnet hos MC12 (bestyckad med MC68912DG256) då ingen extra ”sida” (page) ska användas. Hela FLASH-minnet blir då ett sammanhängande linjärt adressrum.

```
memory ( rw , 0x1000, 0x3000 )       // RWM
memory ( r  , 0x4000, 0xC000 )       // Flash
```

Observera att andra beskrivningar kan komma i fråga får samma typ av microcontroller. Detta hänger samman med att exempelvis EEPROM-minnet kan relokeras. I ovanstående exempel finns inget sådant minne synligt eftersom det döljs av MC12’s IO-area.

Entry-direktivet används endast för att tillhandahålla debug-information. Syntaxen är:

```
entry ( Symbol för programmets startpunkt )
```

Script-filen kan innehålla kommentarer (C++ stil), dvs:

```
// resten av raden är en kommentar.
```

”Flaggor” som accepteras av länkaren kan också ges i script-filen, dessa måste dock föregå alla andra typer av deklarationer.

Några standardskript i ”config”

`default.lsc`

Avsett för mindre applikationer som ryms i MC12's RWM. Används typiskt med standard ”_startup.s12” och förutsätter att IO-rutiner i DBG12 används.

```
-M        // listfil  <konfigurationsnamn>.map skapas.

entry( __start ) //  absolut startpunkt, behövs endast för debugger

// konstantsegment, måste alltid finnas
group  ( c , const_group )
{ abs }

// Kompilatorgenererade segment och 'init'-segmentet till en grupp
group( r , test_group)
{ init, text, rodata, data, bss }

// Separat grupp för avbrottsvektorer i RWM (konvention med DBG12)
group( r, interrupt_vectors )
{ vectors }

// Slutligen placerar vi ut grupperna i RWM
layout
{
    0x1000,0x3C80 <= test_group,
    0x3F80,0x3FFF <= interrupt_vectors
}
```

`flash-dbg12.lsc`

Med detta skript placeras applikationen i den lediga (linjära) portionen FLASH minne med adresser 8000-BFFF. Applikationen samexisterar alltså med DBG12 och kan laddas med hjälp av ”fload” kommandot (Se användarbeskrivning DBG12, minst version 1.14).

```
-M     // listfil  <konfigurationsnamn>.map skapas.
-P     // applikationen ska placeras i Read-Only minne.

entry( __start )  //   absolut startpunkt, behövs endast för debugger

group  ( c , const_group )
{ abs }             // konstantsegment, måste alltid finnas

//  RO-segment, till FLASH minne.
group( r , code_group)
{ init, text, rodata, data }

//  Globala variabler till RWM
group( rw, volatile )
{ bss  }

// Allt tillgängligt minne 0-64k som INTE används av DBG12...
layout
{
    0x1000,0x3C80    <= volatile,
    0x8000,0xBFFF    <= code_group
}
```

Då nya skript-filer skapas är det lämpligt att installera sådana i ”config” biblioteket eftersom detta är en ”standard sökväg” för XCC då länkarskript (.LSC-filer) söks.