Arbetsbok för MC68/MD68k

Art Nr: 120-16, ISBN 91-89280-19-9
©1997-2004 *Roger Johansson och Rolf Snedsbøl*

Kopieringsförbud
Detta verk är skyddat av upphovsrättslagen.
Kopiering är förbjuden utöver lärares rätt att kopiera för undervisningsbruk enligt BONUS-avtal. BONUS-avtal tecknas mellan upphovsrättsorganisationer och huvudman för utbildningssamordnare, exempelvis kommuner och högskolor/universitet. Förbudet gäller hela verket såväl som delar därav och inkluderar lagring i elektroniska medier, visning på bildskärm samt bandupptagning.
Den som bryter mot lagen om upphovsrätt kan åtalas av allmän åklagare och dömas till böter eller fängelse i upp till två år samt bli skyldig att erlägga ersättning till upphovsman/rättsinnehavare.

Göteborgs Mikrovaror
Tel: 031/778 93 10
e-post: `info@gbgmv.se`
Internet: `http://www.gbgmv.se`

Förord

Denna arbetsbok är en del av läromedlen kring enkortsdatorerna *MC68, MD68k* och tillhörande laborationskort. Arbetsboken är i stor utsträckning avsedd som självstudiematerial där eleven på egen hand succesivt arbetar sig igenom de olika momenten i egen takt. Utöver arbetsboken förutsätts eleven ha tillgång till programvaran *ETERM* med tillhörande simulatordelar. Observera att samtliga moment i arbetsboken är av sådan natur att de inte kräver tillgång till hårdvara.

Arbetsboken omfattar ett antal *moment* fördelade på fem avsnitt:

I) Behandlar grundläggande *programutveckling i assemblerspråk*. Syftet med momenten är att introducera grundläggande begrepp för programmering i assembler. Detta innebär såväl assemblerprogrammering av en mikroprocessor (speciellt MC68xxx) men framför allt den använda programutvecklingsmiljön. Avsnittet är direkt förberedande för en första laboration (exempelvis med *MC68* eller *MD68k* och *ML4*).

II) Under hela detta avsnitt arbetar eleven självständigt med att bygga upp ett mycket enkelt skrivargränssnitt. Alla övningar utförs med simulator. Under momentet introduceras *avbrott* men speciell vikt läggs här också vid synkronisering till realtid..

III) Även i detta avsnitt kommer eleven att självständigt bygga upp en mindre applikation, denna gång kring ett tangentbord och en display-modul. Efter att ha arbetat igenom avsnittet kan eleven lämpligtvis ges tillfälle att utföra en laboration där applikationen testas i hårdvara. För denna laboration krävs *MC68* (eller *MD68k+ML18*) och *ML5*/*ML15*/*ML23*.

IV) Under hela detta avsnitt arbetar eleven självständigt med att bygga upp ett komplett applikationsprogram för att styra en liten borrmaskin.

V) Här behandlar vi maskinnära programmering i ’C’, syftet är att ge eleven en introduktion till programutveckling i ’C’ och assembler. Målsättningen är att eleven ska kunna konstruera och testa enklare applikationer.

Göteborg i juli 2002, Roger Johansson och Rolf Snedsbøl

Förord till Andra utgåvan
Denna, den andra utgåvan har bearbetas så att den nu behandlar *ETERM 6.5*, *ETERM 7.0* och *XCC32/XCC68*. Nya uppgifter har tillkommit. Bearbetningen har i huvudsak påverkat avsnitten ett och fem medan endast mindre ändringar företagits i avsnitten två, tre och fyra. Appendix har kunnat minskas eftersom delar av det material som fanns i första utgåvans appendix numera finns i hjälpsystemen för *ETERM* och *XCC*.

Göteborg i augusti 2003, Roger Johansson och Rolf Snedsbøl

Förord till Tredje utgåvan
Denna utgåvan skiljer sig marginellt från den föregående. En del mindre rättelser har gjorts i avsnitten ett t.o.m fyra. Avsnitt fem har bearbetats för anpassning till *XCC32/XCC68* versioner efter 1.0.5.

Göteborg i april 2004, Roger Johansson och Rolf Snedsbøl

Innehåll:

Avsnitt 1
Programutveckling i assemblerspråk

Avsnitt 2
En enkel skrivarport

Avsnitt 3
Tangentbord och Display

Avsnitt 4
Programmeringsprojekt: Borrmaskinen

Avsnitt 5
Maskinnära programmering i C och assembler

Appendix
A IO-adresser för laborationskort
B Motorola S-format
C ASCII representationen
D Exceptionvektorer
E XCC objektfilsformat

Vi rekommenderar också att du skaffar dig någon instruktionslista för den använda mikroprocessorn (MC68340 om du använder MC68, MC68000 om du använder MD68k). Instruktionslistor finns oftast tillgängliga från fabrikantens hemsida. Du finner också instruktionslistor på GMV’s hemsida.

Avsnitt 1

Programutveckling i assemblerspråk

Syften:

Det viktigaste syftet med detta första avsnitt är att ge dig en bild av hur man programmerar i assemblerspråk. Du får lära dig att hantera verktyg för utveckling och test av assemblerprogram. Du kommer också att introduceras i en mikroprosessors instruktionsrepertoir.

Målsättningar:

Börja med att läsa sammanfattningen av detta avsnitt. Det ger dig konkreta detaljer om vilka målsättningar du bör ha.
Då du arbetat i genom avsnittet ska du ha förberett dig för att kunna utföra din första laboration med verklig utrustning. Till denna laboration har du ett speciellt laborations-PM, men du kan inte räkna med att kunna utföra laborationen, inom utsatt tid, om du inte kommer väl förberedd till denna.

Inledning

Detta är en arbetsbok med *övningar* som bland annat syftar till att du ska lära dig behärska enkel programutveckling i assembler (maskinnära programmering). Du kan genomföra alla övningar med hjälp av kurslitteraturen och programutvecklings-systemet *ETERM*. Tänk på att laborationer vanligtvis förutsätter att du förberett dig ordentligt genom att först ha utfört vissa övningar.

Arbetsboken har organiserats i avsnitt, indelade i moment som du arbetar dig igenom i tur och ordning. Av introduktionen till varje avsnitt framgår såväl syften som målsättningar.
Detta första moment har ägnats helt åt beskrivningar av grundläggande begrepp som vi omedelbart kommer att börja tillämpa. Vi börjar med en övergripande beskrivning av *ETERM* och programutvecklingsprocessen.

Om du inte tidigare gjort det, börja nu med att installera *ETERM* på din persondator. Är du osäker på hur du ska göra bör du konsultera din kursansvarige eller någon handledare. Kontrollera att du installerar rätt version (*ETERM 6.5*).

ETERM och Programutvecklingsprocessen

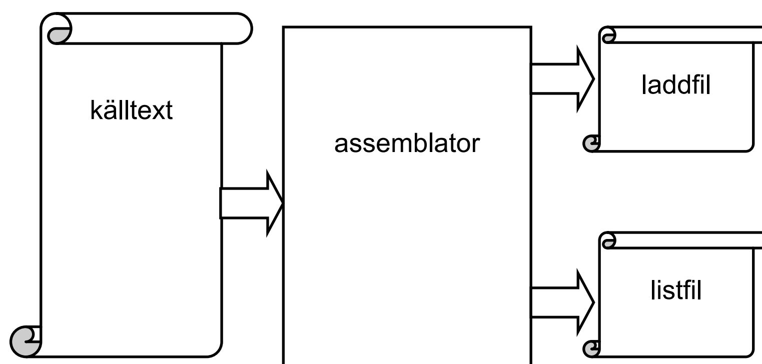
ETERM är ett utvecklingssystem för programutveckling i assembler. *ETERM* har anpassats för undervisningsändamål och fungerar under Windows. Det finns flera varianter av *ETERM* beroende på vilken måldator (laborationsdator) som används. Inledningsvis behandlas *ETERM 6.5 för MC68* och *ETERM 6.5 för MD68k*. Dessa båda varianterna är mycket lika och i de fall det förekommer skillnader kommer vi speciellt att påpeka detta.

ETERM omfattar funktioner för:

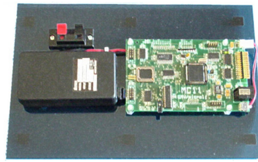
1. **Textredigering**, källtexten skrivs/redigeras med hjälp av en *Editor*, filnamnet ska sluta med *.s68* (source MC68xxx) eller *.asm*. Skillnaden är att editorn's inbyggda funktion för att färga syntax aktiveras om ändelsen är 's68'. Färgad syntax är till för att hjälpa dig upptäcka enklare stavningsfel.
2. **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.
3. **Test**, laddfilen överförs till den inbyggda *simulatorn* eller till laborationsdatorn via ETERM's *terminal* med hjälp av respektive programdels laddningsprogram.

Under detta första moment kommer vi att behandla de två första funktionerna dvs 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* ("assemblern"). Vid assembleringen skapas en *laddfil* och en *listfil*.



I laddfilen finns programmet representerat på en form som kan överföras till instruktioner och data som måldatorn kan tolka.



Laborationsdator och utvecklingssystem (PC). Förbinds via PC'ns COM-port och laborationsdatorns seriekommunikationsport.

Då programmet, i form av laddfil, överförts till måldatorn (laborationsdatorn) kan det *exekveras* (utföras) av måldatorn och man kan testa programmets funktion. 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 ska nu först visa hur man *redigerar* och *assemblerar* med *ETERM*.

MC68

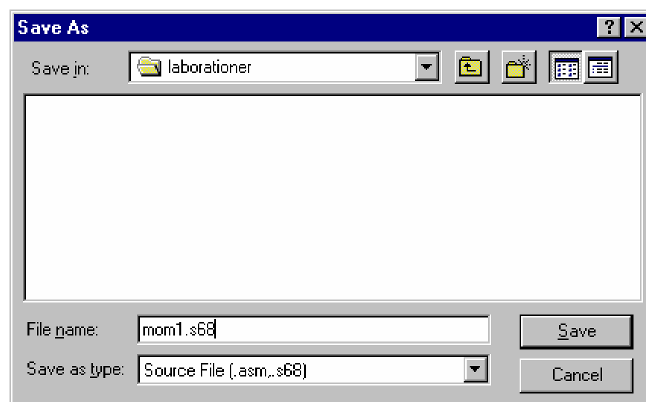
Du startar ETERM för MC68 från Program-menyn...
GMV – ETERM 6.5 -> MC68340 (MC68)

MD68k

Du startar ETERM för MD68k från Program-menyn...
GMV – ETERM 6.5 -> MC68000 (MD68k)

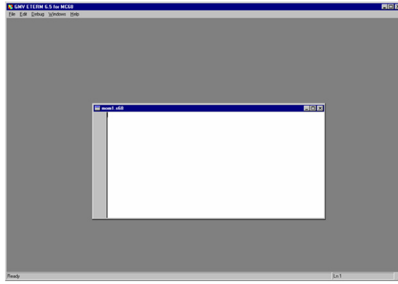
Skapa ett assemblerprogram

Du skapar en ny källtextfil genom att välja **File | New** från menyn.



Därefter skriver du in namnet på den fil du vill skapa "`MOM1.S68`" och klickar 'Save'. Anm. Om du inte anger någon ändelse lägger ETERM automatiskt till "`.asm`". Färgad syntax aktiveras *bara* om ändelsen är ".s68".

Nu skapas ett nytt fönster:



ETERM 's editor skapar ett nytt fönster där du kan redigera din källtext

Det första assemblerprogrammet...

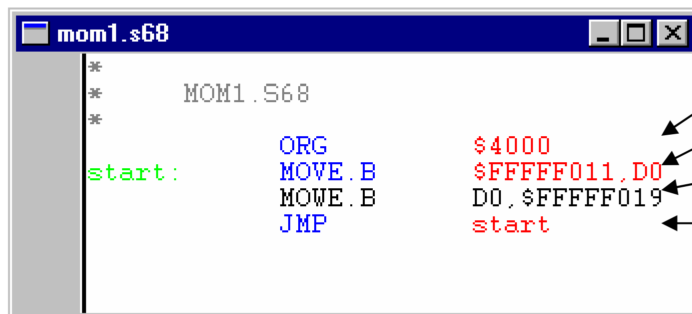
MD68k

För MD68k ska du byta ut adresserna enligt följande:

`FFFFF019⇔3E0011`
`FFFFF011⇔3E0013`

Rader som inleds med '*' tolkas som kommentarsrader, på en sådan rad kan du skriva godtycklig text

Skriv nu följande program



Ange programmets startadress

Läs från inporten till register D0 (D-noll)

Skriv innehållet i register D0 till utporten (felstavad med flit)

Upprepa … ("oändlig slinga")

Dollartecknet anger att påföljande talvärde ska tolkas på hexadecimal form.

Observera hur texteditorn färglägger din text.

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

Observera speciellt raden:

```
        MOWE.B        D0,$FFFFF019
```

som inte färgas riktigt. Detta beror på att vi (avsiktligt) stavat instruktionen fel, rätt stavning är `MOVE.B`. Låt felet vara kvar, vi ska strax rätta till det. För att spara filen använder du **File | Save**.

Om du snabbt vill göra en kopia av denna källtext gör du **File | Save As** och väljer ett nytt namn. Glöm då inte att spara 'originalet' först så att du är säker på att alla dina ändringar finns med i den ursprungliga källtexten.

UPPGIFT 1.1:
Spara filen MOM1.S68, spara den därefter dessutom under det nya namnet MOM1B.S68, ändra därefter den felstavade instruktionen i filen MOM1B.S68 och observera hur den färgas.

SLUT PÅ UPPGIFT 1.1.

Assemblering

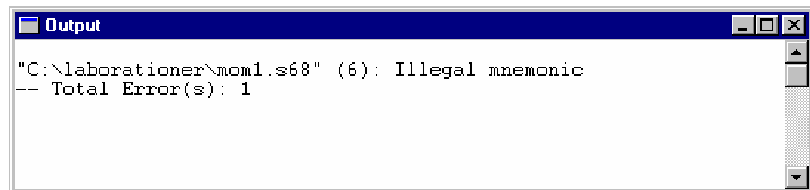
Du assemblerar genom att välja **Debug | Assemble** från menyn. Om det aktiva fönstret (det fönster med mörkblå namnlist) är en källtextfil assembleras denna direkt. Om det är någon annan typ av fönster som är aktivt öppnas en dialogruta där du får ange namnet på den fil du vill assemblera.

UPPGIFT 1.2:
Aktivera fönstret med källtexten MOM1.S68 (med felstavad instruktion) och assemblera denna.

SLUT PÅ UPPGIFT 1.2.

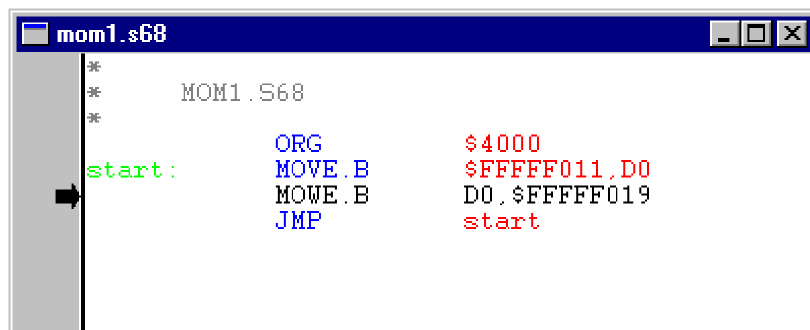
Assemblatorn kommer att klaga på den felstavade instruktionen:

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.
Dubbelklicka nu (vänster knapp) på felutskriften i 'Output'-fönstret.

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



Fönstret med den assemblerade källtexten aktiveras och den felaktiga raden pekas ut. Rätta felet och assemblera på nytt!

Observera att korrekt "färgad syntax" inte nödvändigtvis 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`).

Med assemblerdirektivet ORG (origin) anger vi en startadress i laborationsdatorns minne.
Med direktivet:

```
        ORG    $4000
```

anger vi exempelvis att påföljande instruktion ska placeras på adress $4000 i minnet.

instruktion
källoperand,
destinationsoperand...

Instruktionen MOVE.B $FFFFF011,D0 (*move byte*) laddar ett värde i processorns D0-register. Operanden ($FFFFF011) anger att värdet hämtas från adress $FFFFF011 i måldatorns minne.

MD68k

För MD68k ska du byta ut adresserna enligt följande:

```
FFFFF019⇔3E0011
FFFFF011⇔3E0013
```

Instruktionen MOVE.B D0,$FFFFF019 skriver värdet från processorns register D0 till den adress som anges av operanden (dvs. $FFFFF019).

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 bli 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. Om radens första tecken är blankt uppfattas detta som att raden inte definierar någon symbol.

Ett annat ofta använt assemblerdirektiv är EQU (equate). Det kan användas för att definiera konstanter och fasta adresser. Adressen $FFFFF011 som är ansluten till en in-port i *MC68'*s minne kan exempelvis definieras:

```
ML4_IN     EQU   $FFFFF011
```

På motsvarande sätt kan utporten på adress $FFFFF019 definieras:

```
ML4_OUT    EQU   $FFFFF019
```

MD68k

För MD68k ska du byta ut adresserna enligt följande:

```
FFFFF019⇔3E0011
FFFFF011⇔3E0013
```

UPPGIFT 1.3:
Ändra i källtexten MOM1.S68 så att portarna refereras som symbolerna ML4_IN och ML4_OUT.

SLUT PÅ UPPGIFT 1.3.

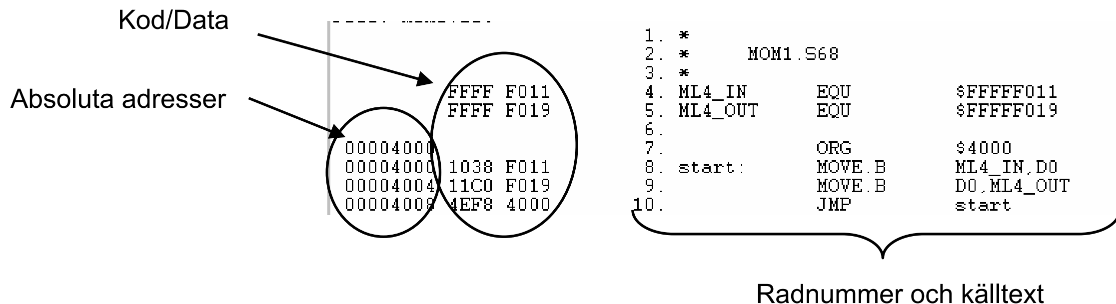
Laddfil och Listfil

Laddfilen används för att överföra programmet (som maskininstruktioner) till måldatorn. I *MC68* (*MD68k*) finns en funktion som kan ta emot och tolka laddfilsformatet. Assemblatorn skapar laddfilen med filnamnstillägget "S19". Eftersom det inte i nuläget är så intressant att känna till laddfilsformatet i detalj nöjer vi oss med att här visa hur laddfilen för vårt programexempel ser ut.

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

```
mom1.s19
S004000000FB
S21000400001038F01111C0F0194EF8400006
S80400400CAF
```

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. Listfilen för vårt programexempel ser ut på följande sätt:



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 1.4:

Assemblera om programmet med följande startadresser:

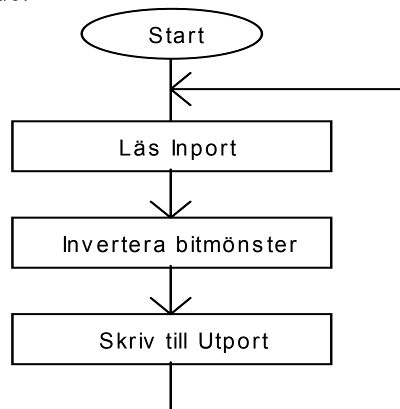
ORG $4400
ORG $5000

För varje assemblering, studera listfilen.

Vilka skillnader upptäcker du i ”kod/data”-fältet?

SLUT PÅ UPPGIFT 1.4.

UPPGIFT 1.5:
Skapa en ny källtext INVERT.S68 och skriv ett assemblerprogram som utför följande:



Ledning:
Instruktionen

```
NOT.B D0
```

inverterar bitmönstret i register D0.

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

SLUT PÅ UPPGIFT 1.5.

Simulatorns grundläggande funktioner

Under detta moment kommer vi att introducera simulator-användning. Så här inledningsvis 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 succesivt att presenteras och illustreras efter hand i kommande moment.

Programdelen *Simulator* i *ETERM* kan användas för att simulera instruktionsutförandet i laborationsdatorn. Med simulatorns hjälp kan du få en första inblick i hur assembler-instruktioner fungerar. Du kan utföra ett assemblerprogram instruktionsvis och i lugn och ro studera effekterna. Simulatorn kan visserligen mycket mer än så, men i detta avsnitt nöjer vi oss med att använda simulatorn för att testa några enkla assemblerprogram. Glöm inte att spara dina källtextfiler, programmen fungerar också i laborationsdatorn (*MC68* eller *MD68k*).

Inledande övningar

Vi ska nu fortsätta arbeta med programmet i ”INVERT.S68” från föregående moment. Vi ska testa programmet i den *MC68-simulator* (*MD68k-simulator*) som finns i *ETERM*. Eftersom programmet utför in- ut- matning från/till laborationskortet *ML4* måste vi dock göra vissa förberedelser innan vi startar simulatorn.

IO-Simulator
Input/Output Simulator simulerar kringenheter till laborationsdatorn

Till *Simulatorn* finns en fristående del som kallas IO-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 *MC68*’s (*MD68k’s*) adressområde måste vi göra vissa inställningar.

Välj från menyn

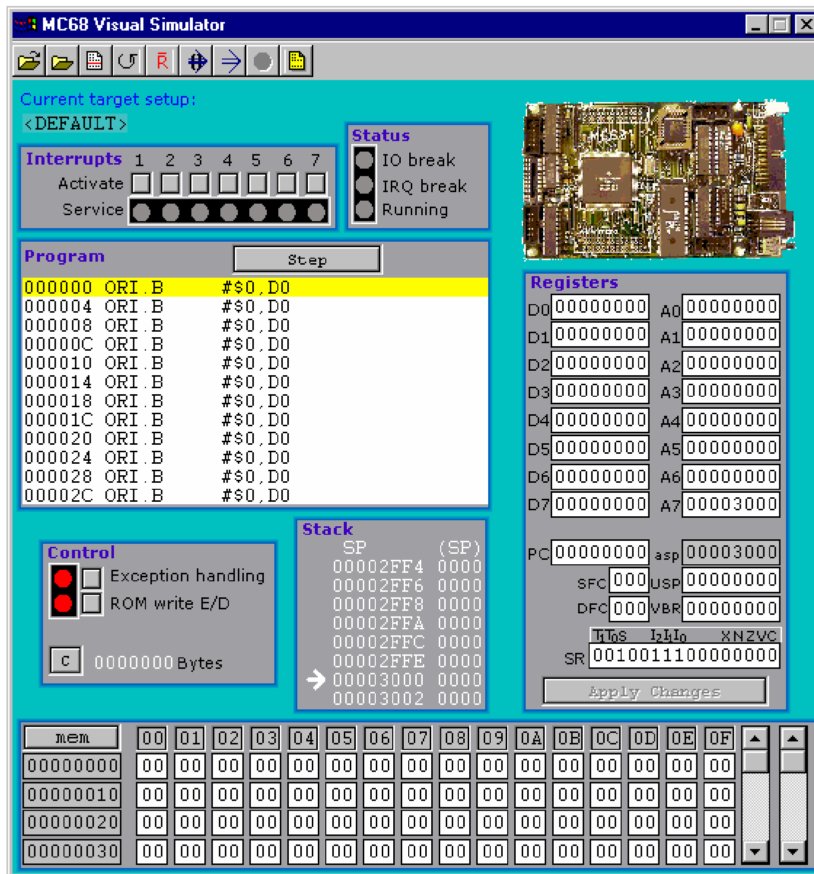
File | Open | invert.s68

källtexten öppnas i ett nytt fönster..
Välj nu från menyn

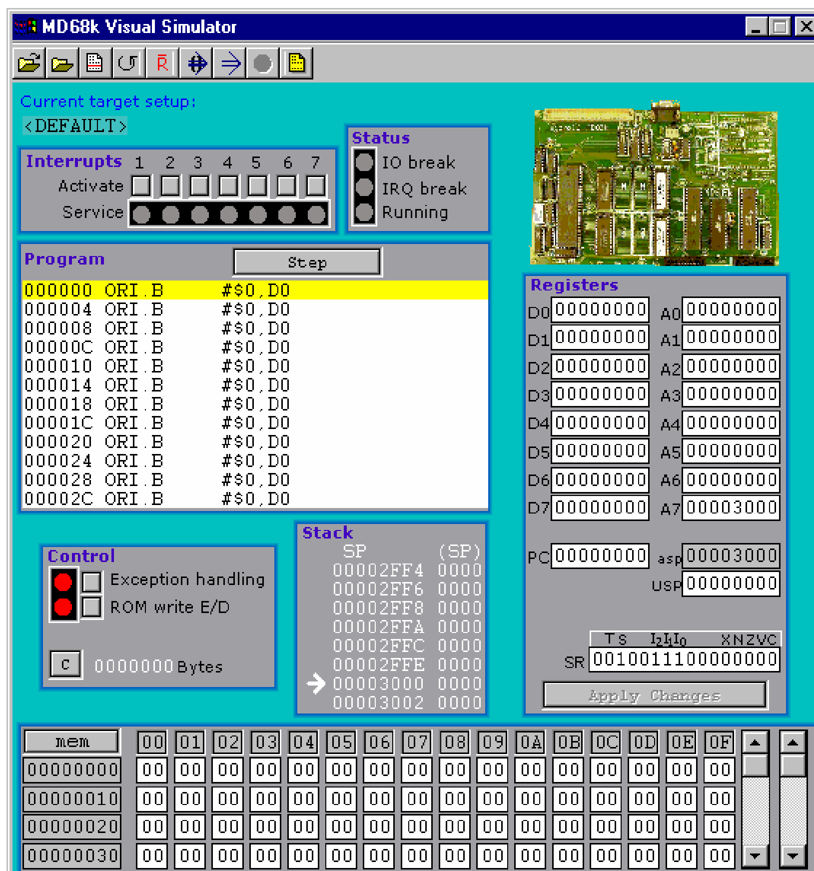
Debug | Simulator

simulatorn startas med laddfilen för den källtext som för tillfället är aktiv...

Simulatorn startas nu, filen ”INVERT.S68” assembleras och den resulterande laddfilen (INVERT.S19) har laddats till simulatorn.



ETERM's MC68 simulator...



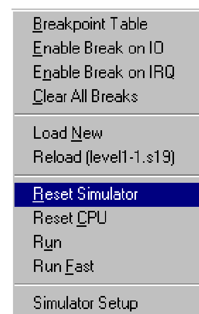
ETERM's MD68k simulator...

Observera att det är mycket små skillnader mellan de olika simulatorerna. Förklaringen är enkel, det är inte speciellt stora skillnader mellan de aktuella datorerna. Du kan ändå inte flytta program mellan dessa utan vidare. Vi kommer fortsättningsvis att illustrera med bilder och exempel hämtade från *MC68*-varianten. För dig som använder *MD68k* kommer vi dessutom att påpeka skillnader som är väsentliga.

Simulatorns POPUP meny visas om du högerklickar utanför 'Program'-fönstret

Placera nu markören någonstans i simulatorfönstret UTOM "Program"-fönstret (vi återkommer till detta senare). Högerklicka pekdonet.

En POPUP meny visas - flera av menyvalen har du dessutom tillgång till via verktygsfältet:



Välj: "Reset Simulator" eller klicka på motsvarande ikon i verktygsfältet.

Programfönstret uppdateras och visar nu det *disassemblerade* programmet i simulatorns minne.

Observera att det disassemblerade programmet ser något annorlunda ut än vår källtext. Det är helt i sin ordning, vi kommer att förklara detta i ett senare moment då vi undersöker olika *adresseringssätt*.

MC68

```
Program                 Step
004000 MOVE.B    ($F011).W,D0
004004 NOT.B     D0
004006 MOVE.B    D0,($F019).W
00400A JMP       ($4000).W
00400E ORI.B     #$0,D0
004012 ORI.B     #$0,D0
004016 ORI.B     #$0,D0
00401A ORI.B     #$0,D0
00401E ORI.B     #$0,D0
004022 ORI.B     #$0,D0
004026 ORI.B     #$0,D0
00402A ORI.B     #$0,D0
```

MD68k

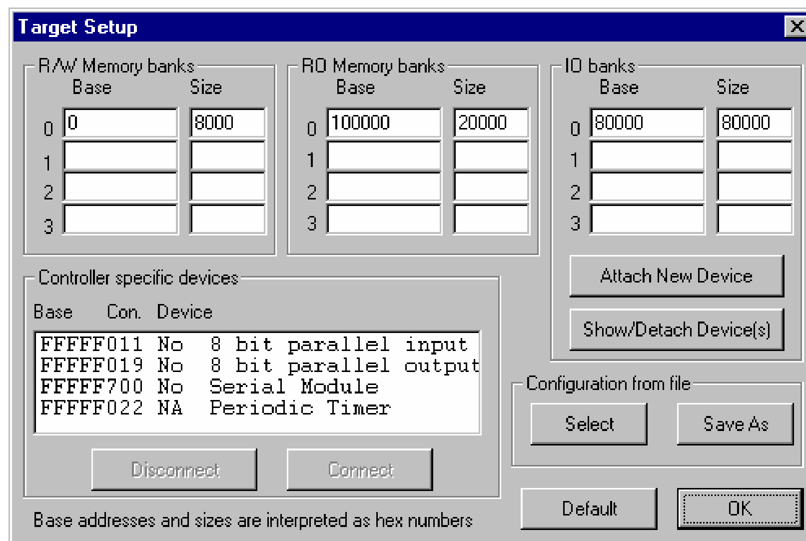
```
Program                 Step
004000 MOVE.B    ($3E0013).L,D0
004006 NOT.B     D0
004008 MOVE.B    D0,($3E0011).L
00400E JMP       ($4000).W
004012 ORI.B     #$0,D0
004016 ORI.B     #$0,D0
00401A ORI.B     #$0,D0
00401E ORI.B     #$0,D0
004022 ORI.B     #$0,D0
004026 ORI.B     #$0,D0
00402A ORI.B     #$0,D0
00402E ORI.B     #$0,D0
```

Simulatorn initierar programräknaren så att den innehåller adressen till den första instruktionen i laddfilen. I fönstret 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.

Simulatorns meny har många alternativ, längst ned finner du 'Simulator Setup' och du ska nu välja denna...

Högerklicka pekdonet och välj nu
" Simulator Setup"
Du kan också använda verktygsfältet...

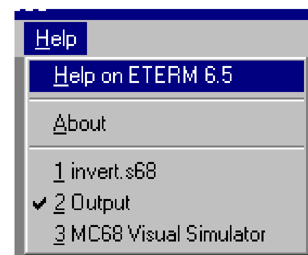
Om du använder MC68



Bilden visar 'Default'-inställningar för en standard MC68.

Om du i stället använder MD68k kan du hoppa över dessa sidor.

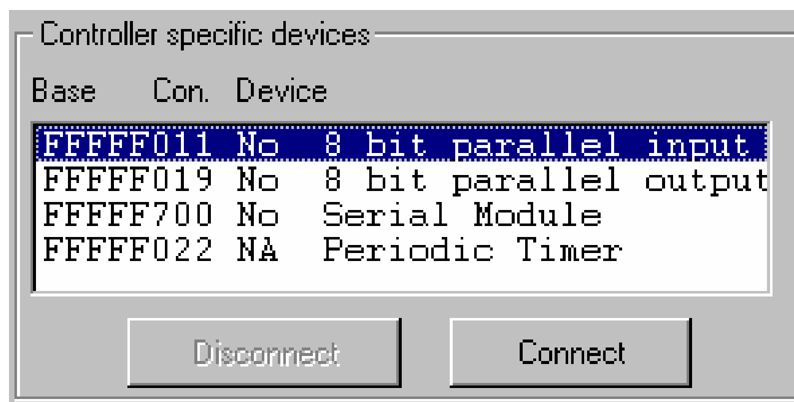
Inställningar för MD68k beskrivs härnäst.

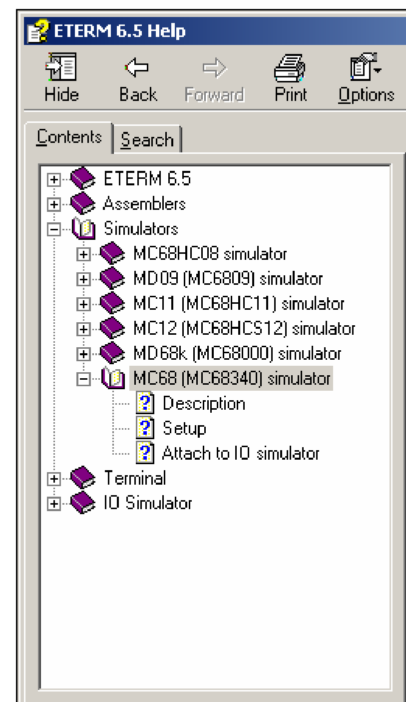


Det finns en rad inställningar som kan göras för simulatorn men vi börjar med de som har att göra med anslutningar till så kallade ”kring-enheter” (*peripheral devices*):

- *Controller specific devices*, här listas de anslutningsmöjligheter som erbjuds direkt av den använda mikrodatorns centralenhet. I *MC68's* fall innebär det exempelvis en 8-bitars parallell inport (adress FFFFF011) och en 8 bitars parallell utport (adress FFFFF019).
- *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 *MC68* i standardutförande.

Vi ska nu ”koppla” portar mellan *MC68* och simulerade kringenheter. I listan 'Controller specific devices' väljer du först ”8-bit parallel input”





inställningar under ETERM's hjälpsystem...

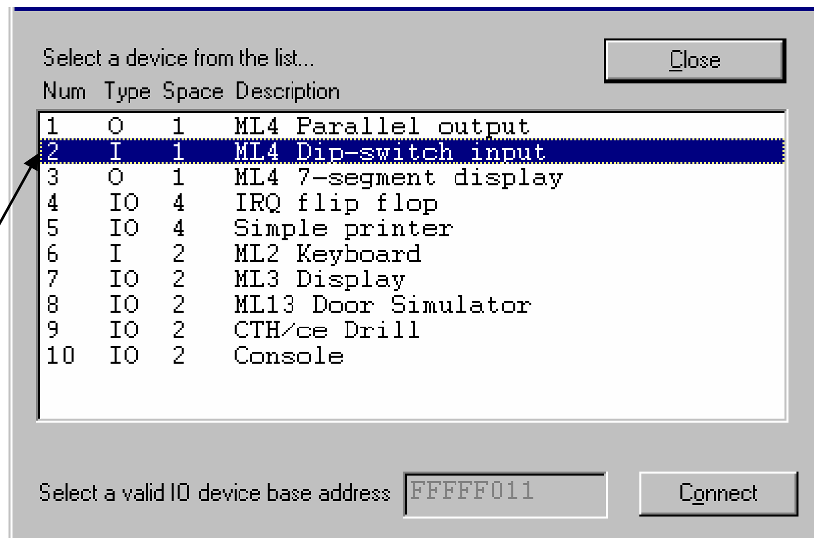
Klicka därefter på 'Connect'....

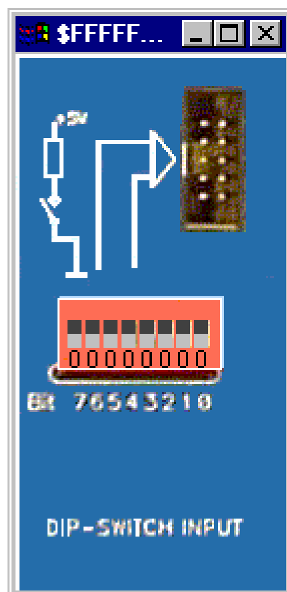
Vi måste ansluta portarna till simularade in- och ut- enheter

Observera att listan med enheter kan se annorlunda ut beroende på den aktuella versionen av IO-simulatorn

Du får nu en ny dialog-box som visar de möjliga anslutningarna. Du kan inte ändra anslutningsadressen (i detta fall FFFFF011, den har grå bakgrund) eftersom du har valt att ansluta en fysisk enhet vars adress redan är definierad. Nu väljer du den *enhet* du vill ansluta.

Märk först enheten
ML4 Dip-switch input
klicka därefter på 'Connect'



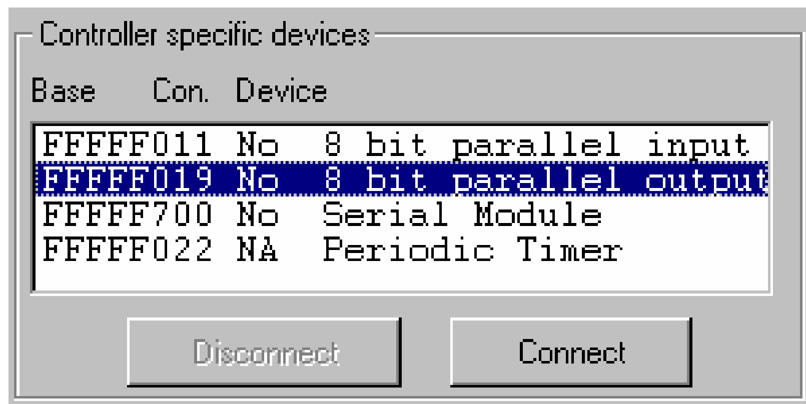


IO-simulatorns bild av ML4-sektionen 'Dip-switch Input"

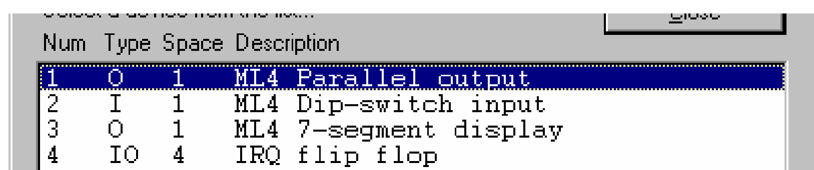
Ett nytt fönster skapas nu, detta är en enhet i IO-simulatorn som används för att simulera kringenheter under ETERM.

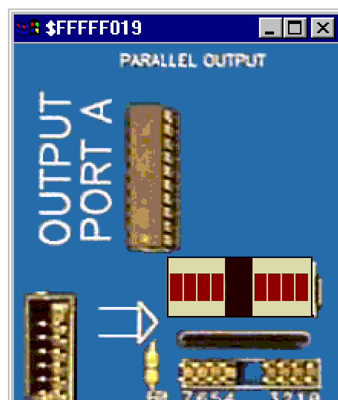
Men vi kan ansluta ytterligare enheter och för vårt inledande programexempel behöver vi nu också en 'Ut-enhet'...

Välj denna gång '8-bit parallell output' och klicka på 'Connect'



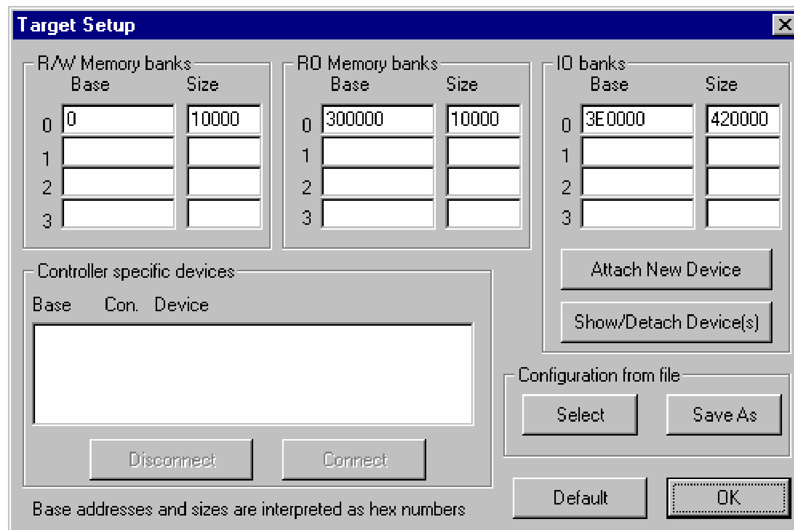
Välj därefter "ML4 Parallel output" som enhet.... klicka på 'Connect'.





IO-simulatorns bild av ML4-sektionen 'Parallel output'

Om du använder MD68k



Bilden visar 'Default'-inställningar för en standard MD68k.

Det finns en rad inställningar som kan göras för simulatorn men vi börjar med de som har att göra med anslutningar till så kallade "kring-enheter" (*peripheral devices*):

- *Controller specific devices*, eftersom *MD68k* är bestyckad med en mikroprocessor (snarare än en microcontroller) är denna box tom, dvs det finns inga enheter här som direkt kan anslutas till någon kring-enhet.
- *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 MD68k i standardutförande.

Vi måste ansluta portarna till simulerade in- och ut- enheter

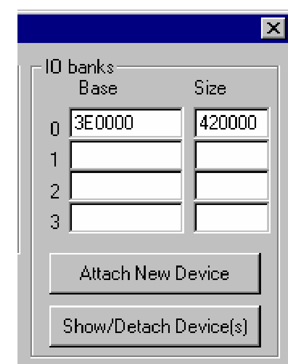
Observera att listan med enheter kan se annorlunda ut beroende på den aktuella versionen av IO-simulatorn

Vi ska nu "koppla" portar mellan *MD68k* och simulerade kringenheter. Observera då att vi endast kan ansluta kring-enheter till adresser som är avsedda för detta ändamål, dvs adresser i den så kallade "IO-arean". För *MD68k* gäller också att kring-enheter måste anslutas till udda adresser. Vi återkommer till detta i avsnitt 3.

I detta inledande exempel ansluter vi adress 3E0011 till en ut-enhet och adress 3E0013 till en in-enhet. Dessa adresser motsvaras av två parallellportar hos *MD68k*.

I sektionen 'IO-banks' klickar du nu på 'Attach New Device'...
Du får nu en ny dialog-box som visar de möjliga anslutningarna. Du måste också ange den adress du vill ansluta till kring-enheten.

Vi börjar med att ansluta ML4's 'Dip-switch' till adress 3E0013.



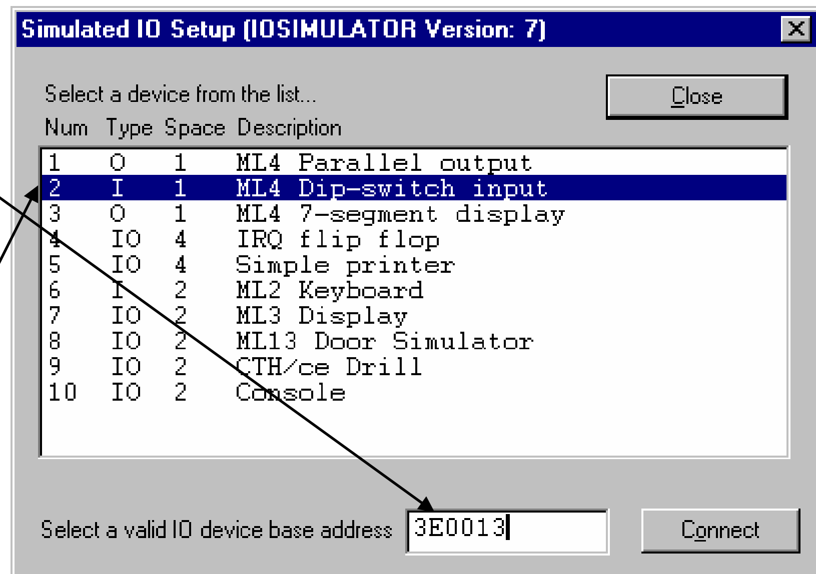
Inställningar av minnesområden för IO (Input/Output)

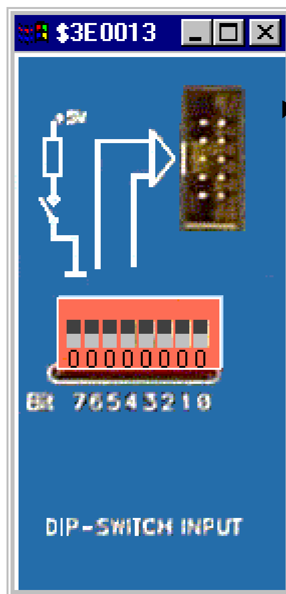
Observera att listan med enheter kan se annorlunda ut beroende på den aktuella versionen av IO-simulatorn

Märk först enheten
ML4 Dip-switch input

Skriv därefter adressen

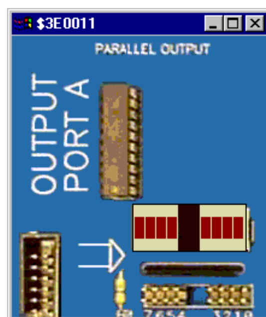
klicka därefter på 'Connect'





IO-simulator:
ML4 Dip switch input

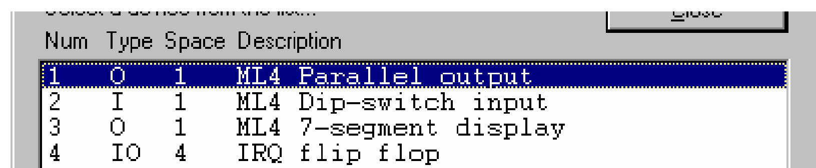
Ett nytt fönster skapas nu (se marginalen), detta är en enhet i IO-simulatorn som används för att simulera kringenheter under ETERM.



IO-simulator:
ML4 Parallel output

Men vi kan ansluta ytterligare enheter och för vårt inledande programexempel behöver vi nu också en 'Ut-enhet'...

Välj denna gång "ML4 Parallel output" som enhet....Skriv in adressen 3E0011 och klicka på 'Connect'.



Nu öppnas ett fönster för den simulerade utenheten (se marginalen).

Då du nu öppnat två nya fönster för simulering av kring-enheter stänger du dialogboxarna genom att klicka på "Close"

TIPS:

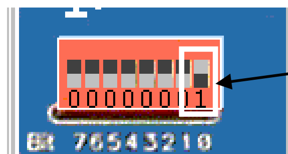
Du kan spara inställningarna för den konfiguration du gjort: Välj (i ”Configuration from File”) ”Save As” – och ange ett lämpligt namn. Nästa gång du vill konfigurera väljer du ”Select” och anger den fil som innehåller den konfiguration du avser.

Fortsätt här då du anslutit enheterna och stängt dialogrutan ”Simulator Setup”.

ML4 Dip switch input

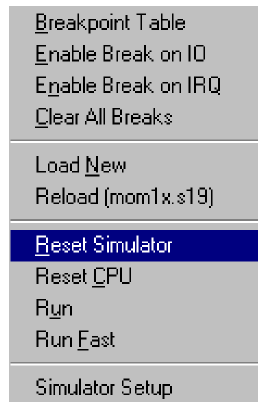
Fältet för bit 0 representeras av den grå/svarta ytan. Genom att klicka i detta fältet ändrar du ’dip-switchens’ utsignal mellan 0 och 1.

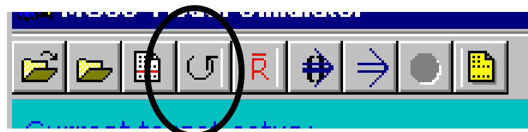
Ställ in värdet 1 på inenheten genom att klicka på fältet för bit 0.



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

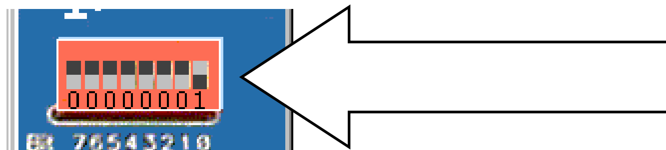
Efter att ha anslutit enheter för såväl in- som utmatning kan vi börja testa vårt program. Högerklicka pekdonet och välj ”Reset Simulator” *eller* klicka på motsvarande ikon i verktygsfältet...





”Reset Simulator” återställer simulatorn till ett begynnelsetillstånd...

Kontrollera att värdet 1 (bit 0 = 1, övriga bitar = 0) är inställt på *ML4 Dip Switch Input*.



Anm.
Om du använder MD68k kommer andra adresser att visas här...

Klicka på 'Step' för att utföra den märkta instruktionen

```
Program                    Step
004000 MOVE.B    ($F011).W,D0
004004 NOT.B     D0
004006 MOVE.B    D0,($F019).W
00400A JMP       ($4000).W
00400E ORI.B     #$0,D0
004012 ORI.B     #$0,D0
004016 ORI.B     #$0,D0
00401A ORI.B     #$0,D0
00401E ORI.B     #$0,D0
004022 ORI.B     #$0,D0
004026 ORI.B     #$0,D0
00402A ORI.B     #$0,D0
```

Du låter simulator'n utföra instruktionen som märkts med *gul* bakgrund genom att klicka på "Step".

Observera, i Register-fönstret, hur värdet i processorns register D0 ändras...

Det värde du ställde in på dip-switchen hamnar nu i register D0...

Registers

D0	00000001	A0	00000000
D1	00000000	A1	00000000
D2	00000000	A2	00000000
D3	00000000	A3	00000000
D4	00000000	A4	00000000
D5	00000000	A5	00000000
D6	00000000	A6	00000000
D7	00000000	A7	00003000

Utför ytterligare en instruktion genom att klicka på 'Step'

Simulatorn har nu utfört

```
NOT.B      D0
```

Observera instruktionens inverkan på innehållet i register D0:

Registers

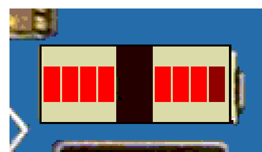
D0	000000FE	A0	00000000
D1	00000000	A1	00000000

Fortsätt nu och utför nästa instruktion:

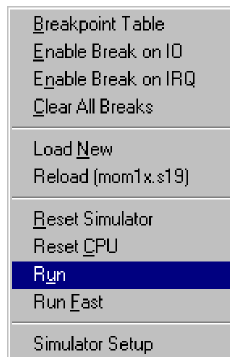
```
MOVE.B     D0,($F019).W
```

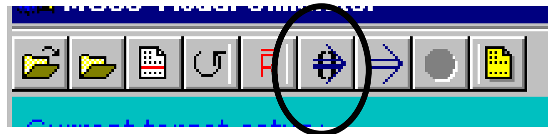
och du bör iaktta följande (hos UT-ENHETEN)

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



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





Starta simulatorn’s ”Run”-kommando från menyn eller genom att klicka på dess ikon i verktygsfältet...

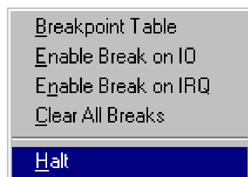
Instruktionerna utförs då i långsam takt (c:a 10 instruktioner per sekund) och efter varje instruktion uppdateras alla simulatorns fönster. Detta är ett sätt att följa programflödet utan att för den sakens skull behöva ”klicka” sig igenom programmet.

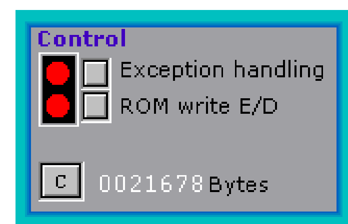
Prova långsam simulering av instruktionsexekvering.

”Statusfönstret”

Programexekveringen startar, observera att dioden ’Running’ nu tänds i ”statusfönstret”. Observera också hur de olika fönstrens innehåll uppdateras mellan varje instruktion

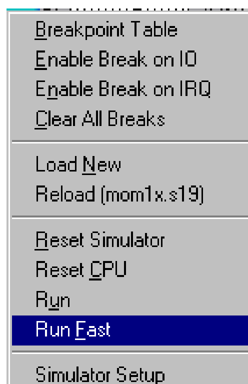
Då du valt ”Run” (eller ”Run Fast”) kommer verktygsfältet att få ett annat utseende. Det enda aktiva alternativet är nu ”Halt”-ikonen. På motsvarande sätt, om du högerklickar, kommer POPUP-menyn att ha ett lite annorlunda utseende. Välj ”Halt” för att avbryta den simulerade instruktionsexekveringen.

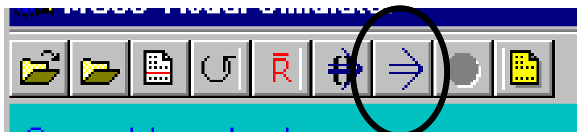




I ”kontrollfönstret” kan du se hur många bytes som används under exekveringen. Räknarna nollställs genom att du klickar på ’C’-knappen.

Om du i stället för ’Run’ väljer ’Run Fast’ simuleras instruktionsexekveringen med maximal hastighet. Fönstren uppdateras då inte på samma sätt och det enda du ser är hur räknarna i statusfönstret uppdateras och att dioden ’Running’ lyser.





Prova nu att utföra programmet med ”Run Fast”. Under simulatorns programexekvering ändrar du inställningarna på Dip-Switchen och observerar ändringarna hos ut-enheten.

UPPGIFT 1.6:
Ställ in följande olika värden på ”Dip-Switchen” – utför programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

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

SLUT PÅ UPPGIFT 1.6.

Du har nu undersökt instruktionen NOT, som ingår i en grupp av instuktioner som ofta samlas under rubriken ”logiska instruktioner”. Missuppfatta inte samlingsnamnet. Det finns ingenting logiskt i instruktionerna i sig, snarare används de för att utföra logiska operationer i dess strikt matematiska bemärkelse. Ytterligare exempel på instruktioner i denna grupp har vi i form av

`ANDI` (*AND Immediate*)
`ORI` (*OR Immediate*)
`EORI` (*Exclusive OR Immediate*).

De sistnämnda utgör *binära* operationer (de har *två* operander) medan NOT, som vi redan sett, är en *unär* operation, dvs använder endast *en* operand.
Vi ska nu i tur och ordning undersöka dessa instruktioner.

Binära logiska operationer

Instruktionen AND utför bitvis logiskt OCH mellan operanderna. Den speciella instruktionen ANDI förutsätter att en av operanderna är en konstant och därför kan adresseringssättet ”*immediate*” (omedelbar) användas, därav instruktionsformens namn.
Låt ”`Inport`”symbolisera det värde vi ställer in på Dip-switchen. Låt ”`Utport`” symbolisera det värde vi skriver till ut-enheten. Låt `X` symbolisera någon konstant. Vi kan då utföra den logiska operationen:

```
Utport = X AND Inport
```

med följande instruktionssekves:

```
MOVE.B     Inport,D0
ANDI.B     #X,D0
MOVE.B     D0,Utport
```

Observera hur operationen använder register D0 (destinations-operanden) även som källoperand. Ändra nu i programmet (byt ut NOT) enligt ovan och använd X=binärt 00001111 (%00001111 kan också skrivas $0F).

UPPGIFT 1.7:
Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt)	Avläst värde (binärt) Utport = $0F AND Inport
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.7.

Ändra nu programmet igen för att utföra operationen:

```
Utport = X OR Inport
```

Använd instruktionen

```
ORI.B
```

UPPGIFT 1.8:
Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt)	Avläst värde (binärt) Utport = $0F OR Inport
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.8.

Slutligen, ändra nu programmet för att utföra operationen

```
Utport = X XOR Inport
```

Använd instruktionen

```
EORI.B
```

UPPGIFT 1.9:
Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt)	Avläst värde (binärt) Utport = $0F XOR Inport
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.9.

De instruktioner du nu undersökt utför grundläggande operationer och du hittar dem därför hos alla mikroprocessorer. Verkar det enkelt så beror det på att det också är det. Ägna ändå nu en stund åt att lösa följande uppgifter...

UPPGIFT 1.10:
Skriv en programsekvens som utför:

```
f(X) = (X XOR 5) AND 16
```

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt) = X	Avläst värde (binärt) Utport = f(X)
1111 0000	
1010 1010	
1100 0011	

Har du tänkt på evalueringsordning? Räknereglerna föreskriver att parenteser alltid evalueras först...

SLUT PÅ UPPGIFT 1.10.

UPPGIFT 1.11:

Skriv en programsekvens som utför:

```
f(X) = X XOR (5 AND 17)
```

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt) = X	Avläst värde (binärt) Utport = f(X)
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.11.

UPPGIFT 1.12:

Skriv en programsekvens som utför:

```
f(X) = NOT (X OR 16)
```

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt) = X	Avläst värde (binärt) Utport = f(X)
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.12.

UPPGIFT 1.13:

Skriv en programsekvens som utför:

```
f(X) = NOT ( (X XOR 5) AND 16)
```

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Inport (binärt) = X	Avläst värde (binärt) Utport = f(X)
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.13.

Det kan nu vara på sin plats med en avslutande anmärkning om användningen av instruktionen NOT. Observera att denna utför *bitvis* komplementering av operanden. I matematisk mening är NOT en negation av värdet SANT eller FALSKT, men instruktionen som sådan *implementerar* (”*förverkligar*”) inte detta meningsfyllt eftersom resultatet av en NOT-instruktionen vanligtvis inte kan tolkas som vare sig SANT eller FALSKT. Vi ska längre fram se hur den matematiska funktionen NOT implementeras i vanliga programspråk.

Aritmetiska operationer

Vi använder aritmetiska operationer för att utföra de fyra olika räknesätten. Operationerna är oftast binära, dvs:

```
resultat = operand1 OP operand2
```

Men vi har också ett specialfall, nämligen

```
resultat = - operand
```

dvs unärt minus.
Vi har lärt oss att tolka detta som:

```
resultat = 0 - operand
```

Operationen är så vanlig att den oftast motsvaras av en speciell instruktion, nämligen

`NEG operand` (*negate*)

OP står för något av de fyra räknesätten:
- + addition
- - subtraktion
- * multiplikation
- / division

UPPGIFT 1.14:
Låt `A` vara vår operand, skriv en programsekvens som utför:

```
A = Inport
Utport = -A
```

Använd register D0 för operanden `A`, instruktionen blir då:

```
NEG.B     D0
```

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Operand = A	Avläst värde (binärt) Utport = -A
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.14.

UPPGIFT 1.15:
Låt `A` vara vår operand, skriv en programsekvens som utför:

```
A = Inport
Utport = -(-A)
```

Använd register D0 för operanden `A`.

Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell:

Operand = A	Avläst värde (binärt) Utport = -(-A)
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.15.

Addition och subtraktion är väl kända operationer. Antag att vi känner den första (källoperanden) och att denna är en konstant. Vi kan då använda:

`ADDI` (*add immediate*)
`SUBI` (*subtract immediate*)

Låt den kända operanden (konstanten) betecknas `B` och antag en variabel som vi läser från `Inport`. Låt vidare operationen betecknas `mnem` (*mnemonic*) och att vi vill skriva vårt resultat till

`Utport.` Vi kan då använda följande instruktionssekvens för att utföra operationen

```
MOVE.B    Inport,D0
mnem      #B,D0
MOVE.B    D0,Utport
```

UPPGIFT 1.16:
Låt `B` vara vår operand, skriv en programsekvens som utför:

```
Utport = (Inport) + B
```

Använd register D0 operationen.

Anm. Parentesen runt ”Inport” används för att understryka att vi menar värdet som finns på adress ”Inport” snarare än adressen för ”Inport”.

Låt `B` vara 5. Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell (översätt också från binär form till decmal form och fyll i respektive kolumner)

(Inport)		Avläst värde (binärt) Utport = (Inport) + B	
binärt	decimalt	binärt	decimalt
1111 0000			
1010 1010			
1100 0011			

SLUT PÅ UPPGIFT 1.16.

UPPGIFT 1.17:
Låt `B` vara vår operand, skriv en programsekvens som utför:

```
Utport = (Inport) - B
```

Använd register D0 operationen.

Låt `B` vara 5. Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell (översätt också från binär form till decmal form och fyll i respektive kolumner)

(Inport)		Avläst värde (binärt) Utport = (Inport) + B	
binärt	decimalt	binärt	decimalt
1111 0000			
1010 1010			
1100 0011			

SLUT PÅ UPPGIFT 1.17.

Vi sparar behandlingen av instruktioner för multiplikations- (*) och divisions- (/) operatorerna till senare moment.

Skiftinstruktioner

Skiftoperationer används för att flytta grupper av bitar ett eller flera steg. MC68xxx-familjen stödjer skiftoperationer med fyra olika instruktioner:

logiskt skift	(LS)
aritmetiskt skift	(AS)
rotation med Carry	(RO)
rotation med Extra Carry	(ROX)

Låt oss undersöka den första, dvs *logiskt skift*.

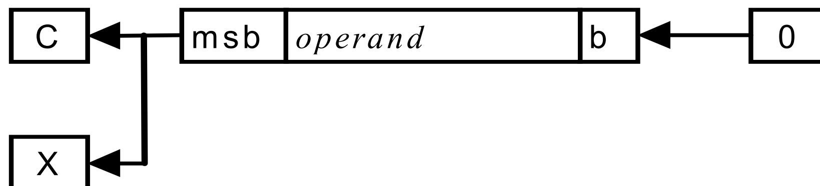
Såväl vänsterskift som högerskift kan utföras. Instruktionen finns i tre olika former, här nöjer vi oss dock med att introducera:

instruktion **`#<count>,Dn`**

"statiskt skift", innehållet i ett dataregister skiftas det antal steg som anges av källoperanden. För de följande exemplen antar vi att vi hela tiden använder register D0.

`LSL` ***logical shift left***

Operandens (dvs innehållet i register D0) mest signifikanta bit kopieras till såväl C som X flaggan. En nolla skiftas in i den minst signifikanta bitens position.



UPPGIFT 1.18:

Undersök instruktionen LSL. Skriv en programsekvens som utför:

```
Utport = (Inport) Shift Left B
```

där B anger antalet skift som ska utföras. Använd register D0 för operationen.

Ledning, instruktionen

```
LSL.B     #B,D0
```

utför logiskt skift 'B' bitar av operanden (innehållet i D0). Resultat efter skift placeras i D0 av processorn.

Låt `B` vara 1. Ställ in följande olika värden på "Dip-Switchen" – utför det nya programmet med "Run" - läs av ut-enheten och fyll i följande tabell.

Operand = (Inport)	Avläst värde (binärt) efter 1 skift
1111 0000	
1010 1010	
1100 0011	

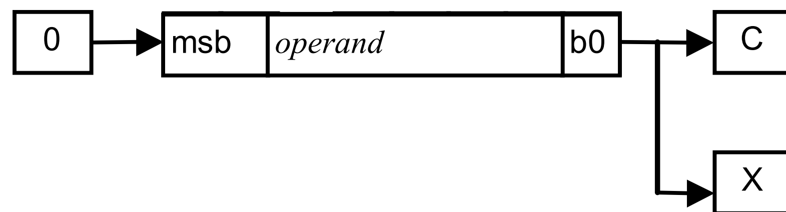
SLUT PÅ UPPGIFT 1.18.

UPPGIFT 1.19:
Utför samma operation som i föregående uppgift men låt denna gång `B` vara 3. Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell.

Operand = (Inport)	Avläst värde (binärt) efter 3 skift
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.19.

`LSR` ***logical shift right***
Operandens minst signifikanta bit kopieras till C respektive X. En nolla skiftas in i den mest signifikanta positionen



UPPGIFT 1.20:
Undersök instruktionen LSR. Skriv en programsekvens som utför:

```
Utport = (Inport) Shift Right B
```

där B anger antalet skift som ska utföras. Använd register D0 för operationen.

Ledning, instruktionen

```
LSR.B      #B,D0
```

utför logiskt skift ‘B’ bitar av operanden (innehållet i D0). Resultat efter skift placeras i D0 av processorn.

Låt `B` vara 2. Ställ in följande olika värden på ”Dip-Switchen” – utför det nya programmet med ”Run” - läs av ut-enheten och fyll i följande tabell.

Operand = (Inport)	Avläst värde (binärt) efter 2 skift
1111 0000	
1010 1010	
1100 0011	

SLUT PÅ UPPGIFT 1.20.

Simulatorns registerfönster

Simulatorns registerfönster visar registeruppsättningen hos den använda mikroprocessorn (MC68340 för MC68 och MC68000 för MD68k), du upptäcker säkert de smärre skillnaderna. Det finns något fler register hos MC68340 och dessutom används en extra bit i statusregistret. Dessa skillnader har dock ingen som helst betydelse så här inledningsvis, vi återkommer till detta senare.

Registeruppsättning

D0...D7, generella dataregister
A0-A6, generella adressregister
A7 stackregister
PC programräknare
SR statusregister

dessutom följande register, vars användning vi kommer att behandla senare i kursen:

USP, (*User Stack Pointer*) alternativt stackregister

asp är inget register utan bara ett extra fönster som anger värdet hos ”aktiv stackpekare”, dvs A7 eller USP.

för MC68 gäller även
VBR, (*Vector Base Register*)
SFC, (*Source Function Codes*)
DFC, (*Destination Function Codes*)

MC68 ***MD68k***

Du kan ändra registrens innehåll manuellt genom att klicka på fönstret du avser och skriva in ett nytt värde. För registren SR, SFC och DFC tolkas värdet som på binär form, för övriga register tolkas värdet som på hexadecimal form. Då du ändrat ett registerinnehåll aktiveras knappen ’Apply Changes’, *du måste då klicka på denna* för att ändringarna ska registreras i simulatorn.

UPPGIFT 1.21:
Skriv en programsekvens som utför:

`Utport = operand Shift Right B`

operand förutsätts finnas i register D0.

Låt `B` vara 2. Utför programsekvensen instruktionsvis. Inför varje skiftinstruktion ställer du in nya värden i D0 (enligt tabell) komplettera följande tabell:

Operand = D0	Avläst värde (binärt) efter skift
0011 0000	
0010 1011	
1000 0001	

SLUT PÅ UPPGIFT 1.21.

Simulatorns hantering av minnet

Den simulerade datorn’s minnesinnehåll kan visas i block om 64 bytes. Du ändrar visningsintervall med hjälp av scrollningslisterna till höger.

mem	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Med den yttre scrollningslisten ändrar du de 16 *mest* signifikanta bitarna hos visningsadressen. Med den inre scrollningslisten ändrar du de 16 *minst* signifikanta bitarna hos visningsadressen.

Du kan också ange en startadress för blocket som ska visas. Klicka på knappen ”mem” längst upp till vänster.

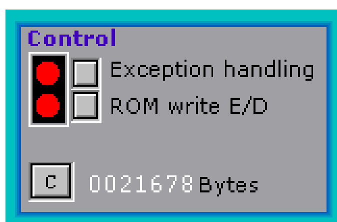
set	00	01
00100000	00	00
00000010	00	00

Det första fönstret blir nu skrivbart. Skriv in den adress du vill visningen ska börja med och klicka sedan på samma knapp (som nu heter ”set”).

För de minnesadresser där det är tillåtet, kan du ändra minnesinnehåll genom att skriva in ett nytt (hexadecimalt) värde. Sådana minnesinnehåll har en *vit* bakgrund.

Innehållet på adresser inom vissa minnesintervall kan du *inte* ändra, det kan exempelvis bero på att dessa adresser är reserverade för en kring-enhet eller att simulatorn ej konfigurerats med minne på dessa adresser. De minnesinnehåll för kring-enheter du inte kan ändra har en *grå* bakgrund. Dessa kan i bland visa godtyckligt innehåll. Du ska alltså betrakta dessa värden med stor skepsis. Följande figur ger exempel på ett sådant minnesintervall.

mem	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
FFD0FFC0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0
FFD0FFD0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0
FFD0FFE0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0
FFD0FFF0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0

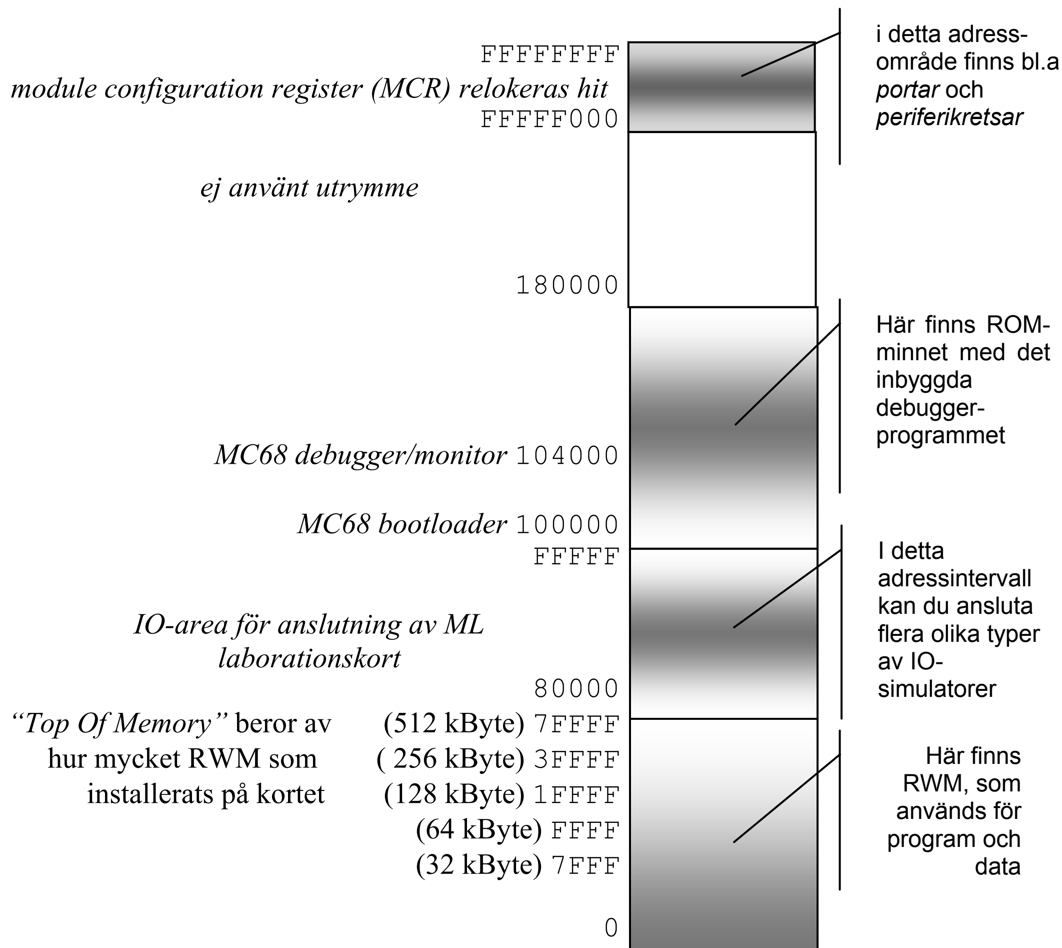


I ”kontrollfönstret” väljer du om RO-minne ska vara skrivbart i simulatorn.

En annan typ är de minnesareor som är reserverade för ROM (enbart läs-minne). 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) är detta minne skrivbart vilket innebär att minnet behandlas som skrivbart. Om du vill att det därefter ska representera ett fysiskt RO-minne, klickar du en gång på knappen.

Minnesdisposition – MC68

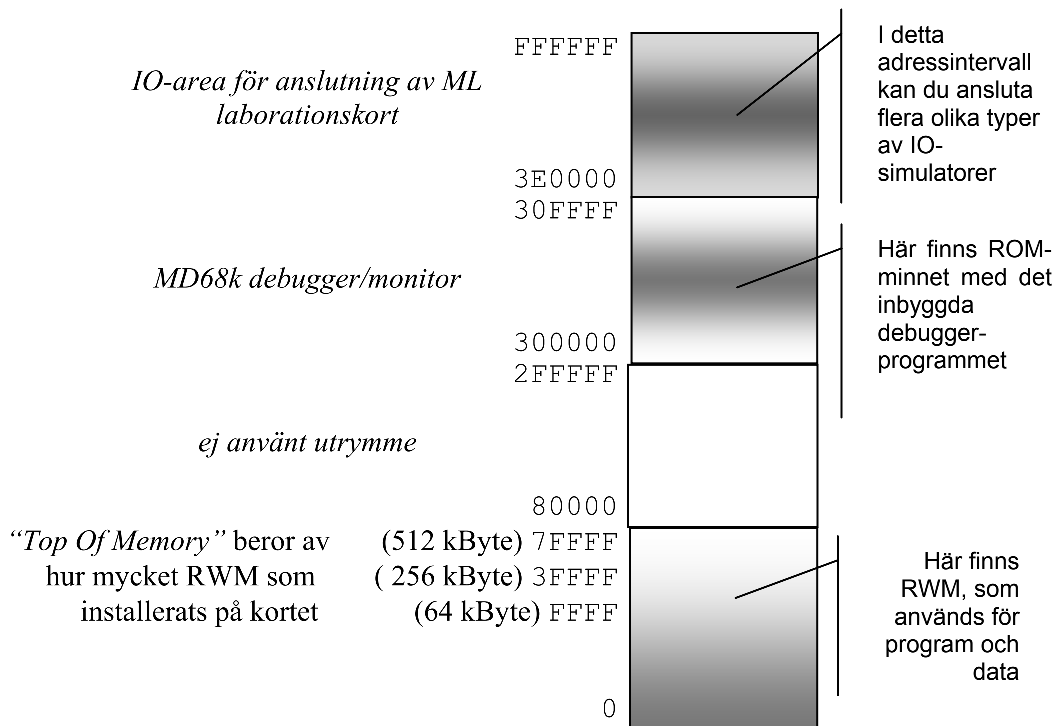
MC68's minnesdisposition anger de adressområden som kan användas och *hur* de kan användas. Dessutom anger minnesdispositionen adresserna till systemets yttre enheter.



Då du startar *MC68* kontrolleras datorn av det inbyggda *debugger*-programmet (*db68*). Detta är alltså placerat i PROM-minne, dels för att behålla innehållet då spänningen till *MC68* slås av, men också för att andra program inte av misstag ska kunna ändra debugger-programmet.

Under programutvecklingen placeras det program som ska testas i RWM-minnet. Detta sker genom en enkel *laddningsprocedur* där programmet överförs från *värddatorn* till *måldatorns* minne. Efter laddningen kan programmet testas med hjälp av debuggern.
Mikrodatorn kommunicerar med *omvärlden* via in- och ut-portar och dessa har placerats på *bestämda* adresser grupperade i den så kallade *IO-arean*.

Minnesdisposition – MD68k



Statisk minnesinitiering

Med ”statisk minnesinitiering” menar man helt enkelt att man initierat ett minnesinnehåll före det att programmet startas. Det finns ett enkelt sätt att utföra denna initiering med hjälp av assemblerdirektiv.

```
ORG   $5000
DC.B  $43
```

(*define constant byte*) instruerar assemblatorn att placera hexadecimala värdet 43 på adress 5000 i måldatorns primärminne.

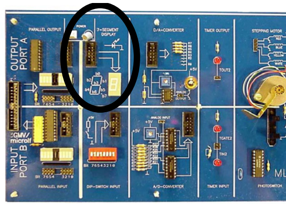
Flera argument (värden) kan ges med direktivet. Dessa måste då skiljas åt med kommatecken. Observera att inga blanktecken får förekomma. Följande exempel illustrerar en tabell, med start på adress $5000 som innehåller de decimala värdena 1-9.

```
ORG   $5000
DC.B  1,2,3,4,5,6,7,8,9
```

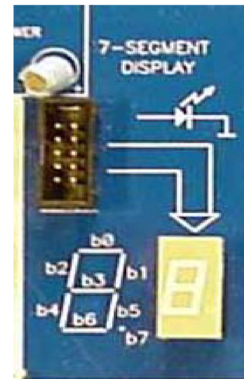
UPPGIFT 1.22:
Skapa en tabell, med start på adress $5000, som innehåller alla udda tal mellan 0 och 20. Skriv nödvändiga direktiv och assemblera din källtext. Ladda till simulatorn och undersök minnesinnehållet. Kontrollera att minnet initierats på rätt sätt.

SLUT PÅ UPPGIFT 1.22.

Sju-sifferindikator



Vi skall under detta moment använda en 7-*segmentsdisplay* eller kort och gott *sifferindikator,* som kan användas till exempelvis tid- eller temperaturvisning. Här använder vi en enkel mindre sifferindikator som också finns på laborationskortet *ML4*.



Introduktion

Sju-sifferindikatorn är en av de vanligaste presentationsenheterna i digitala system. Bakgrunden är enkel, man vill 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 enda uppgift är att tända en decimalpunkt, vi kommer inte att använda decimalpunkten i detta moment, men det kan ändå vara bra att veta.

Sju-sifferindikatorn på *ML4* 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.

Din uppgift, under detta moment är:
Skriv ett assemblerprogram som utför översättning och utmatning av de binära siffrorna 0 t.o.m 9 till motsvarande representationer på sju-sifferindikatorn.

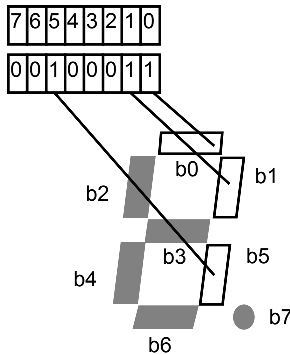
Här får du lära dig:

⇒ hur en 7-sifferindikator (eller 7-segmentsdisplay) fungerar och hur siffror kan visas på en sådan indikator
⇒ villkorligt programflöde
⇒ hur en mikrodators arbetstakt kan anpassas till mänsklig uppfattningsförmåga

Funktionsbeskrivning

Uppgiften består i att konstruera ett assemblerprogram som klarar utskrift av siffrorna 0-9 (på sju-sifferindikatorn) utgående från de decimala siffrorna 0-9.

EXEMPEL:
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 hexadecimalt '23' vilket medför att detta bitmönster formar en “sjua” på sifferindikatorn

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

Decimalsiffra	Binärkod
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
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. Här använder vi ML4 och du finner ett exempel på översättningen av den decimala siffran '7' till dess motsvarande sju-sifferkod, i marginalen. Observera speciellt att det alltid åtgår 8 bitar för en sju-sifferrepresentation, medan siffrorna 0-9 (binärt) kan representeras med 4 bitar.

Fyll nu i följande tabell med sju-sifferkoder för de binära talen 0-9.

Decimalsiffra		Sjusegmentskod		
	Binärkod	Binärkod	Hexkod	Förklaring
0	0000			segmentkod_noll
1	0001			segmentkod_ett
2	0010			segmentkod_två
3	0011			segmentkod_tre
4	0100			segmentkod_fyra
5	0101			segmentkod_fem
6	0110			segmentkod_sex
7	0111	0010 0011	23	segmentkod_sju
8	1000			segmentkod_åtta
9	1001			segmentkod_nio

En enkel programsekvens som i tur och ordning skriver ut segmentkoderna för 0,1,2,3 osv kan exempelvis konstrueras på följande sätt:

```
Skriv (segmentkod_noll) -> UTPORT
Skriv (segmentkod_ett) -> UTPORT
Skriv (segmentkod_två) -> UTPORT
Skriv (segmentkod_tre) -> UTPORT
osv.
```

UPPGIFT 1.23:
Skriv en programsekvens som kontinuerligt matar ut sju-sifferkoderna för siffror 0-9, i tur och ordning till indikatorn.

Testa programsekvensen i ETERM's simulator genom att först koppla IOSIMULATORNs sektion *ML4 7-segment display* till utportens adress.

Utför instruktionsvis exekvering och kontrollera att samtliga siffror skrivs ut korrekt. Om siffrorna 0..9 visas i tur och ordning har du också skrivit en korrekt programsekvens. Om inte – måste du lokalisera och avhjälpa felet.

Då din programsekvens fungerar riktigt, enligt ovanstående, prova med simulatorns 'Run'.... Vad händer ?

Är displayen läsbar?

Om inte, har du någon rimlig förklaring till varför det som förut fungerade inte längre synes bete sig rätt?

Om du inte kommer på något svar på denna fråga kan du ändå fortsätta, du får ytterligare en chans senare...

SLUT PÅ UPPGIFT 1.23.

En programsekvens som utför rätt sak behöver nödvändigtvis inte ge det resultat man tänkt sig. Den nya komplikation vi illustrerat leder oss till att diskutera mikroprocessorns arbetstakt. Uppenbarligen är det så att det inte räcker med att prestera en programsekvens som utför rätt saker. Den måste också synkroniseras (anpassas i tid) på ett sådant sätt att resultatet är meningsfyllt, dvs kan uppfattas av en operatör som observerar resultatet.

Det finns flera sätt att utföra denna anpassning, vi ska här visa ett av de enklaste sätten. Metoden bygger på att mikroprocessorn fås att utföra någon obetydlig uppgift vars enda syfte är att fördröja programexekveringen. Metoden kallas ofta ”busy-waiting” – dvs ”upptagen med att vänta” som trots den skenbara meningslösheten är en enkel och mycket effektiv metod.

Styrning av programflöde

Med *programflödesändring* menar man att det gängse arbetssättet:

läs nästa adress och tolka som instruktion

avbryts, och programflödet, dvs instruktion från nästa adress, inte längre hämtas sekvensiellt, utan anges *i* en speciell instruktion. Vi har redan använt en sådan instruktion:

```
JMP         <adress>
```

Instruktionen har endast en operand (en adress) som anger var *nästa* instruktion ska hämtas. Det finns andra instruktioner med samma betydelse, exempelvis

```
BRA         <adress>
BRA.S       <adress>
BRA.W       <adress>
BRA.L       <adress>
```

Instruktionerna har alla samma effekt men *kodas* annorlunda. För närvarande har detta mindre betydelse och du bör betrakta dom som likvärdiga. Vi kommer, för enkelhetens skull, att använda varianterna JMP och BRA. (*Jump* resp. *BRanch Always*) även fortsättningsvis.

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.

Villkorliga instruktioner

Villkorliga instruktioner används, som namnet antyder, för att utföra en eller flera instruktioner då någon förutsättning är uppfylld. Ofta kallas dessa instruktioner för ”hopp”- instruktioner eller *branch*-instruktioner. I instruktionslistan anges ett antal *Bcc*-instruktioner som utför ett programhopp om hoppvillkoret är uppfyllt, (*Bcc: Branch conditionally*). Villkorliga instruktioner används alltså alltid tillsammans med någon (omedelbart föregående) instruktion som åstadkommit flaggsättning. Det är därför viktigt att du alltid kontrollerar hur flaggorna sätts av instruktionen som föregår branch-instruktionen.

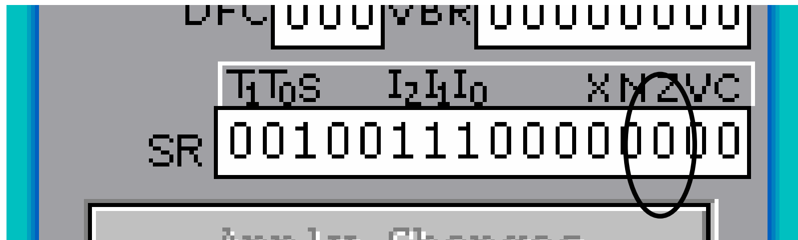
En villkorlig (branch-) instruktion:

- Testar villkoret mot innehållet i flaggregistret (**SR**)
- Om resultatet av testen är SANT, utförs instruktionen (hoppet)
- Om resultatet är FALSKT fortsätter exekveringen med nästa instruktion (hoppet utförs inte).

14 olika villkor (hoppinstruktioner) kan anges, men vi ska här koncentrera oss på villkoren:

BEQ (Branch on EQual)
BNE (Branch on Not Equal

I nedre delen av simulatorns Registersektion hittar du registret SR (Status Register). Bland de minst signifikanta bitarna kan du identifiera ”Z-flaggan”. Biten sätts till 1 av processorn om resultatet av den senast utförda operationen blev 0. Biten nollställs om resultatet blev skilt från noll. De flesta instruktioner påverkar Z-flaggan



UPPGIFT 1.24:
Skriv en programsekvens som utför följande instruktionssekvens. Undersök för varje instruktion hur den påverkar Z-flaggan.

Instruktion	Z
`MOVE.B #1,D0`	
`MOVE.B #0,D0`	
`ADDI.B #1,D0`	
`SUBI.B #1,D0`	

SLUT PÅ UPPGIFT 1.24.

UPPGIFT 1.25:
Skriv en programsekvens som utför följande instruktionssekvens. Stega genom programmet För varje gång programmet ska utföra SUBI-instruktionen läser du av D0 och Z-flaggan. Fyll i nedanstående tabell.

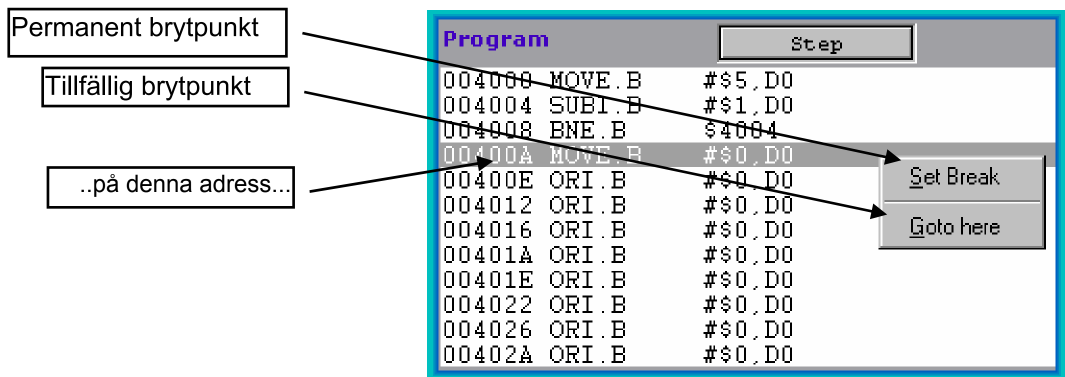
```
          ORG       $4000
          MOVE.B    #5,D0
again:    SUBI.B    #1,D0
          BNE       again
          MOVE.B    #0,D0
          ...
```

Iteration	D0	Z-flagga
1		
2		
3		
4		
5		
6		

SLUT PÅ UPPGIFT 1.25.

Brytpunkter i det simulerade programmet

Om du *högerklickar* exempelvis då markören är över raden som inleds med ’00400A MOVE.B...’ i programfönstrets *vita* bakgrund får du denna POPUP-meny:

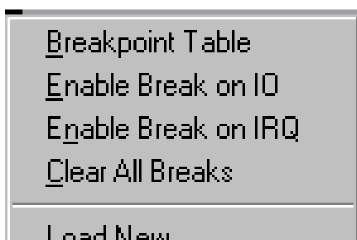


En 'brytpunkt' är ett ställe där programexekveringen avbryts. Brytpunkten kan vara permanent (läggs då in i en så kallad brytpunktstabell, beskrivs nedan). Varje gång programmet ska utföra instruktionen på denna adress kommer simulatorn att avbryta exekveringen. Brutpunkten kan också vara tillfällig ('Goto here'). Programmet exekveras tills denna adress uppträder nästa gång. Brytpunkten tas därefter bort. Brytpunkter är bara meningsfyllda då du använder 'Run' eller 'Run Fast'.

UPPGIFT 1.26:

Starta om simulatorn. (Reset Simulator).
Placera en tillfällig brytpunkt på adress 400A. (Goto Here).
Observera simulatorns beteende.

SLUT PÅ UPPGIFT 1.26.

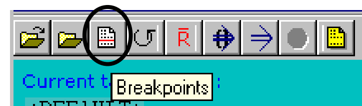


POPUP-menyns del för brytpunktshantering

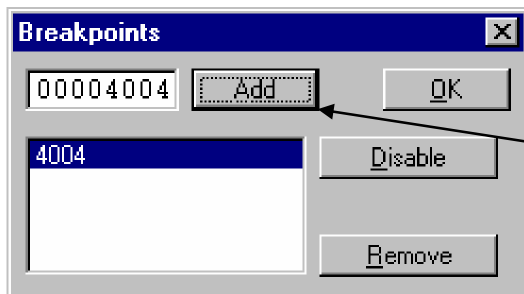
Det finns andra möjligheter att sätta ut brytpunkter:

- *Enable Break on IO*, brytpunkt sätts automatiskt vid skrivning/läsning till/från någon adress som reserverats för 'Input/Output', IO. Brytpunkterna markeras inte i programfönstret men en diod i statusfönstret indikerar att denna funktion aktiverats.
- *Enable Break on IRQ*, brytpunkt sätts automatiskt vid avbrott, vi återkommer till detta i ett senare moment.
- *Breakpoint Table*, brytpunkstabell. Här kan du lägga in upp till 20 samtidiga brytpunkter i ditt program. För att bestämma vilka adresser som kan vara lämpliga för brytpunkter bör du studera listfilen av ditt program.

Följande exempel visar den enkla hanteringen av brytpunktstabellen. Välj ’Breakpoint Table’ från POPUP-menyn, du kan också använda motsvarande ikon i verktygsfältet.

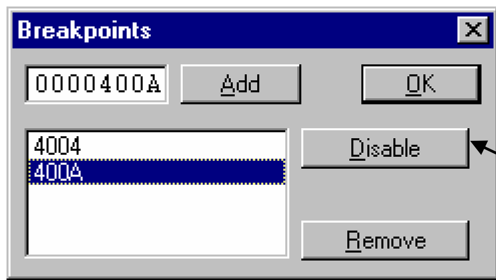


Följande dialogruta visas:



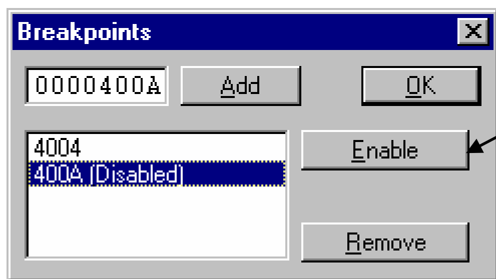
Skriv adressen och klicka på ’Add’ för att lägga in en ny brytpunkt i tabellen

Lägg nu in ytterligare en brytpunkt, på adressen 400A...



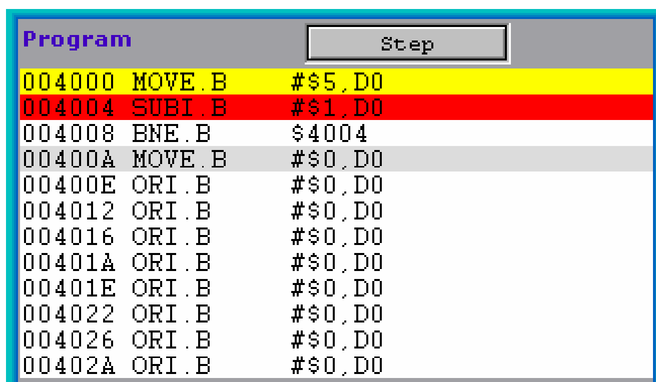
Välj en existerande brytpunkt i tabellen genom att klicka på dess adress..

En brytpunkt kan inaktiveras med ’Disable’, brytpunkten finns dock kvar i tabellen och kan aktiveras på nytt med ’Enable’



Klicka nu på ’OK’ för att stänga dialogrutan

Observera nu hur programfönstret påverkats av brytpunkterna.



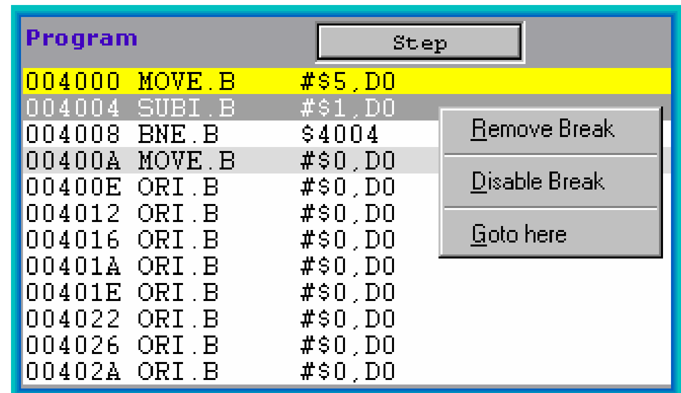
- En *aktiv* brytpunkt illustreras av röd bakgrund
- Den gula bakgrunden anger vilken instruktion som kommer att utföras härnäst
- En *passiv* brytpunkt illustreras av grå bakgrund

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

Brytpunkter som på detta sätt är synliga i programfönstret är enkla att hantera...

Högerklicka på raden med den aktiva brytpunkten

En POPUP-meny 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 POPUP-meny

---

UPPGIFT 1.27:
Utgå nu från tidigare lösningar. Skapa ett program enligt följande:

```
skriv '0' till indikator
fördröj
skriv '1' till indikator
fördröj
```

Ledning, med 'fördröj' menas här en vänteslinga av den typ som illustrerats i tidigare uppgift. Använd 10 som första värde för fördröjning.

Prova programmet med 'Run' respektive 'Run Fast'. Observera simulatorns beteende och kommentera....

SLUT PÅ UPPGIFT 1.27.

Fler adresseringssätt

I slutet av föregående moment fick du se hur du kan skapa en ”tabell” med konstanter i datorns minne. Vi ska nu se hur en sådan tabell, tillsammans med ett nytt adresseringssätt, kan användas för att förenkla olika programkonstruktioner.

UPPGIFT 1.28:
Använd simulatorn för att undersöka följande adresseringssätt. Fyll i tabellen.

Instruktion	Påverkan (adress eller register)	Resultat
`MOVE.L #$12345678,D0`	D0	$12345678
`MOVE.W #$1234,D0`		
`MOVE.B #$12,D0`		
`MOVEA.L #$5000,A0`		
`MOVE.L #$87654321,($5000).L`		
`MOVE.L (A0),D1`		
`MOVE.B (A0)+,D2`		
`MOVE.B -(A0),D3`		
`MOVE.B (2,A0),D4`		
`MOVE.B #$55,(1,A0,D4)`		

SLUT PÅ UPPGIFT 1.28.

Adresseringssätten :

`(xxxx).W` *absolute word*
`(xxxx).L` *absolute long*

är lika men det finns en mycket viktig skillnad. I *absolute word* kodas adressen med 16 bitar, dessa teckenutvidgas till 32 bitar innan operandens adress slutglitigt bestäms. Detta innebär att adresseringssättet endast kan användas i adressintervallen:

`00000000 – 00007FFF`

respektive

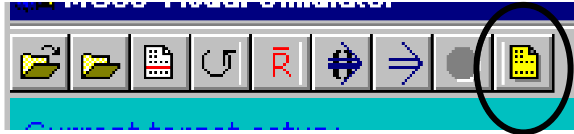
`FFFF8000- FFFFFFFF`

Du kan enkelt se skillnaden genom att prova följande enkla program:

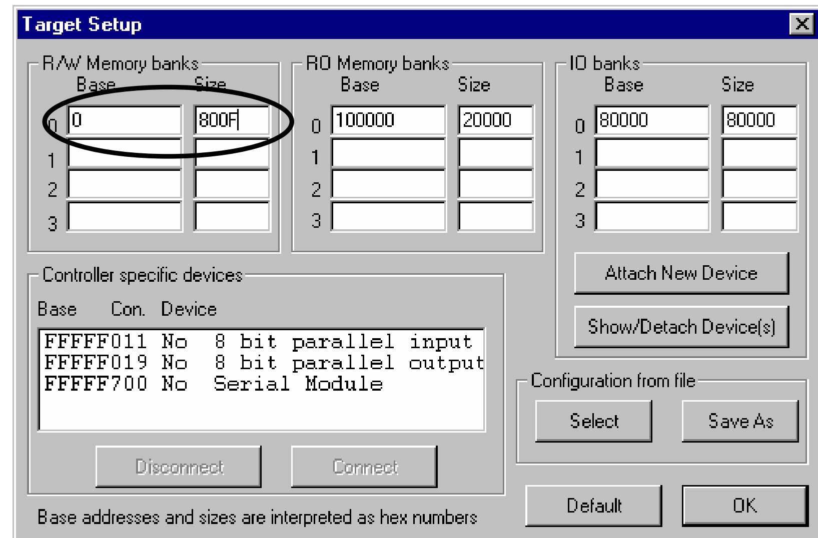
```
ORG         $4000
MOVE.B      ($8000).L,D0
MOVE.B      ($8000).W,D1

ORG   $8000
DC.B  $33
```

Innan du fortsätter måste vi dock påpeka att standardinställningar för *MC68* inte klarar detta program. Det beror på att den normalt inte har konfigurerats med minne över adress 8000. Med simulatorn är det dock inga större problem eftersom vi kan konfigurera denna för en praktiskt taget godtycklig minneskonfiguration. Välj ’Simulator Setup’ genom att först högerklicka, eller välj direkt från verktygsfältet.

Högerklicka pekdonet och välj nu
” Simulator Setup”
Du kan också använda verktygsfältet...



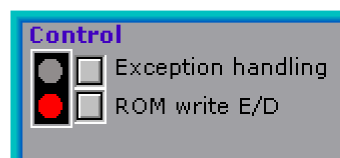
Om du använder MC68:



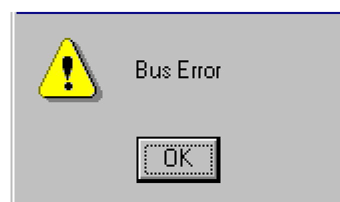
Notera hur vi utvidgat den första minnesbanken (till 800F).

Om du använder *MD68k* behöver du inte ändra minneskonfiguration.

Sätt nu ’Exception Handling’ ”disable”, enligt följande figur:



Prova nu det givna programmet. Observera hur instruktionen som refererar adress 8000 via adresseringssättet *long*, fungerar som avsett medan instruktionen som refererar adress 8000 via word genererar ’Bus Error’...



Detta beror helt enkelt på att i andra fallet blir den effektiva adressen FFFF8000 och här finns inget konfigurerat minne.
Man kan ange ett alternativt skrivsätt för detta adresseringssätt, dvs genom att endast ange adressen.

```
      MOVE.B      $8000,D0
```

Assemblatorn översätter därefter till lämpligt adresseringssätt (*word* eller *long*).

För att du ska få en uppfattning om hur denna översättning bestäms ska du nu assemblera följande sekvens och därefter studera listfilen.

```
      ORG         $4000
ADR_1:
      MOVE.B      ADR_1,D0
      MOVE.B      ADR_2,D1
      MOVE.B      (ADR_2).W,D2
      MOVE.B      (ADR_2).L,D3
ADR_2:
```

Jämför hur de två första MOVE-instruktionerna kodas av assemblatorn. Kan du tänka ut någon rimlig förklaring till detta?

Ledning:
Assemblatorn behandlar källtexten rad för rad i två pass. Under det första passet bestäms alla adresser. I de fall adresseringssätten är okända och dessutom symbolernas värden ännu okända måste assemblatorn ”gissa”.

Adresseringssätt med adressregister

Låt oss nu speciellt studera adresseringssättet:

`(An)+` ***address indirect, post auto increment***

Du kan använda valfritt adressregister (A0-A7). I dessa fall ska du dock undvika register A7 som har speciell funktion. Vi återkommer strax till detta.

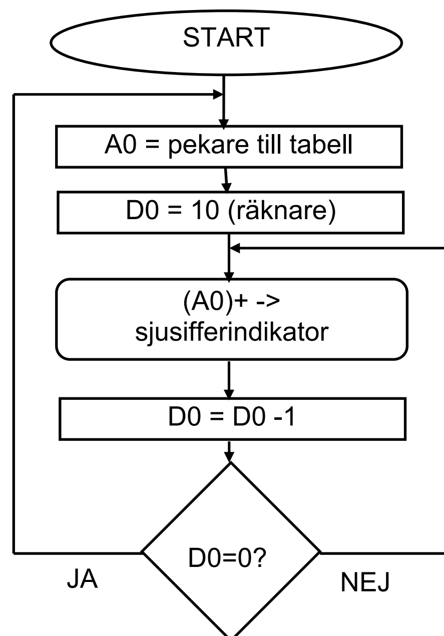
Operanden ges av innehållet på den adress som finns i adressregistret. Omedelbart efter att operanden bestämts ökas innehållet i adressregistret med antingen 1,2 eller 4 (*Byte*, *Word* eller *Long*).

UPPGIFT 1.29:
Använd simulatorn för att undersöka följande instruktionssekvens. Fyll i tabellen.

Instruktion	Register D0	Register A0
`MOVE.L #$87654321,($5000).L`	-	-
`MOVEA.L #$5000,A0`	-	
`MOVE.L (A0)+,D0`		
`MOVE.W (A0)+,D0`		
`MOVE.B (A0)+,D0`		

SLUT PÅ UPPGIFT 1.29.

Vi återvänder nu till programmet som skrev segmentskoder för 0,1,2,3 ...9 till vår sjusifferindikator. Följande flödesschema visar en lösning där vi använder register D0 som räknare och adressregister A0 som en pekare, till det mönster vi vill skriva ut.



UPPGIFT 1.30:
I samma källtext ska du nu kombinera en tabell (data) med programtext (kod).

- Skapa en tabell, med start på adress $5000 som innehåller segmentskoder för siffrorna 0..9 i tur och ordning.
- Skapa programtexten, enligt ovanstående flödesschema, med start på adress $4000.
- Använd simulatorn och övertyga dig om att programmet fungerar som det ska genom instruktionsvis exekvering (Step).

Komplettera programmet med en lämplig fördröjningsslinga. Använd register D1 som räknare i fördröjningen. Försök dimensionera fördröjningen så att varje siffra visas c:a 1 sekund då du använder ’Run’.

SLUT PÅ UPPGIFT 1.30.

Låt oss nu studera ytterligare ett användbart adresseringssätt:

`(d16,An)` ***address indirect with displacement***

Operanden bestäms här genom att en konstant (d16) adderas till innehållet i An, operanden hämtas sedan från denna adress, i symbolisk form skulle detta kunna skrivas:

```
(An) + d16 -> temp;
(temp) -> operand
```

Konstanten, som här kallas "displacement" (ung. *förskjutning*) är ett 16 bitars tvåkomplementstal, dvs förskjutningar i intervallet

$$-32768 \leq d16 \leq 32767$$

kan användas.
Observera att i de fall d16 är noll, ersätter assemblatorn med adresseringssättet:

`(An)` ***address register indirect***

vilket kodas något kortare.

UPPGIFT 1.31:
Precis som i föregående övning ska du nu kombinera en tabell (data) med programtext (kod).

Skapa en tabell, med start på adress $5000 som innehåller segmentskoder för siffrorna 0..9 i tur och ordning.

Skapa programtexten, enligt nedan, med start på adress $4000.

Använd simulatorn för att undersöka följande instruktionssekvens. Fyll följande tabell.

Instruktion	Register D0	Register A0
`MOVEA.L #$5000,A0`	-	
`MOVE.B (A0),D0`		
`MOVE.B (0,A0),D0`		
`MOVE.B (1,A0),D0`		
`MOVE.B (2,A0),D0`		
`MOVE.B (3,A0),D0`		
`MOVE.B (4,A0),D0`		
`MOVE.B (5,A0),D0`		
`MOVE.B (6,A0),D0`		
`MOVE.B (7,A0),D0`		
`MOVE.B (8,A0),D0`		
`MOVE.B (9,A0),D0`		

SLUT PÅ UPPGIFT 1.31.

Låt oss slutligen studera ytterligare ett adresseringssätt som använder adressregister men som dessutom ger oss möjlighet att addera/subtrahera en *variabel* förskjutning:

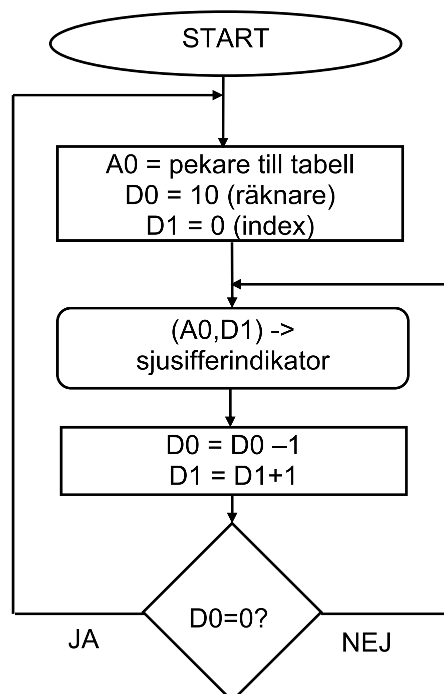
`(d8,An,Dn)`

address indirect, displacement and register index

Operandens adress bildas genom att innehållet i `An` adderas till innehållet i `Dn` och konstanten `d8`.

I nästa uppgift ska du använda detta adresseringssätt för att skriva mönster till sjusifferindikatorn.

Följande flödesschema visar en lösning där vi använder register D0 som räknare, adressregister A0 som en pekare med register D1 som indexregister, till det mönster vi vill skriva ut.



UPPGIFT 1.32:

I samma källtext ska du nu kombinera en tabell (data) med programtext (kod).

- Skapa en tabell, med start på adress \$5000 som innehåller segmentskoder för siffrorna 0..9 i tur och ordning.
- Skapa programtexten, enligt ovanstående flödesschema, med start på adress \$4000.
- Använd simulatorn och övertyga dig om att programmet fungerar som det ska genom instruktionsvis exekvering.

Komplettera programmet med en lämplig fördröjningsslinga. Använd register D2 som räknare i fördröjningen. Försök dimensionera fördröjningen så att varje siffra visas c:a 1 sekund då du använder 'Run'.

SLUT PÅ UPPGIFT 1.32.

Subrutiner

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

En subrutin karakteriseras av att den har ett inträde (entry) och en avslutning (exit). Inträdet anges oftast genom att man placerar en symbol i symbolfältet, avslutningen, dvs "åter från subrutin" specificeras av en speciell maskininstruktion. Följande subrutin uppfyller detta men utför i övrigt ingenting:

```
sub:
      RTS
```

Observera instruktionen som avslutar subrutinen

RTS ***return from subroutine***

Vi ska strax förklara denna, innan vi gör det ska vi emellertid visa de instruktioner som används för att utföra ett *subrutinanrop*.

```
      JSR   sub
```
eller
```
      BSR   sub
```

skillnaden mellan dessa hänförs huvudsakligen till kodningen. Vi koncentrerar oss nu på:

JSR ***jump to subroutine***

operanden för JSR är en subrutins inträde. Denna kan anges med flera olika adresseringssätt men det vanligaste är en symbolisk adress, vilket alltså kan jämställas med adresseringssätten *absolute word* eller *absolute long*.

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 A7, måste alltså ha initierats med en lämplig adress innan JSR används.

För såväl *MC68* som *MD68k* gäller att stacken initierats till $3000 av den inbyggda programvaran. Så är även fallet i simulatorn varför 'Reset Simulator' bland annat kommer att placera denna adress i A7.

Då JSR "sparar" en adress kommer denna att placeras på stacken, instruktionen använder (implicit) processorns register A7. På motsvarande sätt kommer RTS att återställa en adress, genom att hämta denna från stacken. Formellt kan detta beskrivas som:

JSR:

- Minska stackpekaren med 4
- Placera återhoppsadress på stacken
- Placera adressen till subrutinen i PC

RTS:

- Placera adressen som för närvarande ligger överst på stacken i PC (programräknaren)
- Öka stackpekaren med 4.

Simulatorns stackhantering

Simulatorns 'stack' ruta används för att visa den aktiva stackpekarens värde (`A7` eller `USP`) beroende på S-biten i statusregistret (`SR`) Stackrutan hjälper dig att få en överblick av den aktiva stackpekaren och innehållet på stacken. Den vita pilen indikerar såväl aktiva stackpekarens innehåll `SP` som innehållet på den adress som pekas ut `(SP)`. Ytterligare några närliggande adresser visas. Varje gång aktiva stackpekaren påverkas (ändras) av någon instruktionsexekvering ändras också pilens läge och färg.

UPPGIFT 1.33:

Studera noggrannt stackhanteringen vid utförandet av följande instruktionssekvens.

```
          ORG    $4000
start:           JSR   sub1
                 JSR   sub2
                 BRA   start

sub1:            MOVE.B     #1,D0
                 RTS

sub2:            MOVE.B     #2,D0
                 RTS
```

SLUT PÅ UPPGIFT 1.33.

UPPGIFT 1.34:

I ett tidigare moment avrådde vi i från att använda register A7 (urskiljningslöst) även om adresseringssättet tillät detta. Varför tror du vi gav detta råd:

SLUT PÅ UPPGIFT 1.34.

UPPGIFT 1.35:
Du har tidigare skapat ett program som matar ut segmentkoder till sju-sifferindikatorn. I detta program har du också placerat en ”fördröjnings-slinga”. Organisera nu detta program så att fördröjningen åstadkoms av en subrutin. Ditt huvudprogram ska anropa denna rutin för att skapa fördröjning mellan utmatningen av tecknen.

SLUT PÅ UPPGIFT 1.35.

I nästa övning ska du kombinera inmatning med ditt program. Använd ’Simulator Setup’ för att koppla ’ML4 Dip-switch input’ till en inport.

UPPGIFT 1.36:
Modifiera ditt program, från förgående uppgift, så att detta läser indata från inporten, översätter till segmentskod och matar ut resultatet till sjusifferindikatorn.

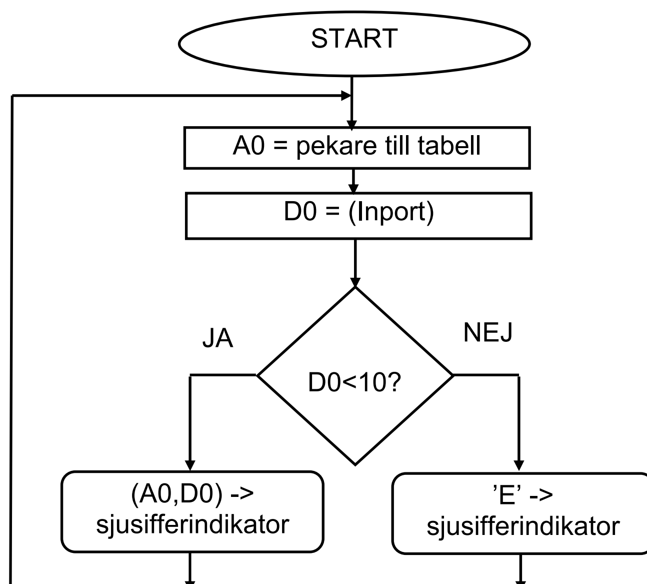
Prova programmet genom att använda ’Run’ – ställ in siffrorna 0-9 (på binär form) på sifferindikatorn. Verifiera funktionen.

Vad händer om du ställer in andra värden?

SLUT PÅ UPPGIFT 1.36.

Villkorligt programflöde

Vi ska nu ytterligare modifiera programexemplet från föregående avsnitt. I programmet ska vi lägga till en instruktion som jämför om indata är högre än nio och i så fall utför en programsekvens som skriver ut tecknet 'E', enligt följande flödesplan.



Det modifierade programmet kan också uttryckas med följande pseudokod:

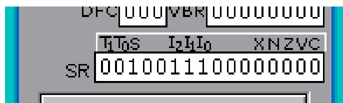
```
do{
      byte=read_inport();        Läs inporten
      if(byte < 10){             Om värdet är mindre än 10
        byte=convert(byte);      så omvandla till segmentkod
        display(byte);           och visa data
      }else                      annars
        display('E');            visa ett 'E'
}forever
```

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 *relation* används. Låt oss nu se hur vi kan koda denna i assemblerspråk på ett enkelt sätt. För just dessa ändamål finns, i instruktionsuppsättningen, ett antal villkorliga hoppinstruktioner, (Bcc=*branch on condition*). Gemensamt för dessa instruktioner är att flaggorna i statusregistret testas för att avgöra om ett speciellt villkor är sant eller inte (jfr BEQ, BNE). Hoppet utförs om villkoret är sant annars fortsätter exekveringen med instruktionen omedelbart efter branch-instruktionen.

mnemonic	villkor	sanningsuttryck
BHI	High	$\overline{C} \bullet \overline{Z}$
BLS	Low or Same	$C + Z$
BCC	Carry Clear	$\overline{C}$
BCS	Carry Set	C
BNE	Not Equal	$\overline{Z}$
BEQ	Equal	Z
BVC	Overflow Clear	$\overline{V}$
BVS	Overflov Set	V
BPL	Plus	$\overline{N}$
BMI	Minus	N
BGE	Greater or Equal	$N \bullet V + \overline{N} \bullet \overline{V}$
BLT	Less Than	$N \bullet \overline{V} + \overline{N} \bullet V$
BGT	Greater Than	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$
BLE	Less or Equal	$Z + N \bullet \overline{V} + \overline{N} \bullet V$

För att använda oss av dessa villkorliga instruktioner måste vi också övertyga oss om att flaggorna satts korrekt av en *omedelbart föregående* instruktion. Åtskilliga instruktioner påverkar flaggorna och det är därför viktigt att instruktionsföljden väljs med omsorg. Det finns ett fåtal instruktioner vars enda uppgift är att sätta flaggorna. Exempel på sådana instruktioner är:

TST ***testa operand***



Flaggorna Z och N sätts beroende på operanden medan flaggorna C och V alltid nollställs.

Observera att detta är en *unär* operation, dvs har endast *en* operand. De enda relationer som kan bestämmas är alltså om operanden är noll (skild från noll) eller om dess teckenbit är 1 (teckenbit är 0). En testoperation kan aldrig generera CARRY eller OVERFLOW och dessa statusflaggor nollställs alltså alltid. Följdaktligen blir vissa Bcc-instruktioner meningslösa (de som testar C och/eller V flaggorna) i samband med TST-instruktionen.

Ett annat exempel på instruktioner som endast påverkar flaggsättning är:

`CMP` ***(compare) jämför operand med...***

Medan TST är en unär operation är CMP en binär operation, dvs kräver två operander. Utförandet av CMP är det samma som SUB (subtract) bortsett från att vid CMP påverkas endast flaggorna. Instruktionen sätter även V och C flaggorna baserat på utfallet av jämförelsen.

UPPGIFT 1.37:
Utred, genom praktiska försök med simulatorn, flaggornas och register D0's innehåll i följande fall.

D0 *före*	Instruktion	D0 *efter*				X	N	Z	V	C
$1F 1F 1F 1F	`CMPI.B #$F1,D0`									
$F1 F1 F1 F1	`CMPI.B #$F1,D0`									
$FF FF FF FF	`CMPI.B #$F1,D0`									
$1F 1F 1F 1F	`SUBI.B #$F1,D0`									
$F1 F1 F1 F1	`SUBI.B #$F1,D0`									
$FF FF FF FF	`SUBI.B #$F1,D0`									
$1F 1F 1F 1F	`ADDI.B #$F1,D0`									
$F1 F1 F1 F1	`ADDI.B #$F1,D0`									
$FF FF FF FF	`ADDI.B #$F1,D0`									

Besvara nu följande frågor.

Hur sätts C-flaggan i processorns statusregister av följande jämförelseinstruktion?

```
CMPI.B    #10,D0
```

a) om D0 innehåller 9?_____________

b) om D0 innehåller 10?____________

c) om D0 innehåller 11?____________

För vilka värden (i D0) utförs programflödesändringen av följande instruktionssekvens:

```
CMPI.B    #10,D0
BCS       ...
```

SLUT PÅ UPPGIFT 1.37.

UPPGIFT 1.38:
Du ska nu konstruera ett program som kontinuerligt:

- Läser ett värde från Inport
- Kontrollerar 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 siffran 'E' skrivas till sjusifferindikatorn.

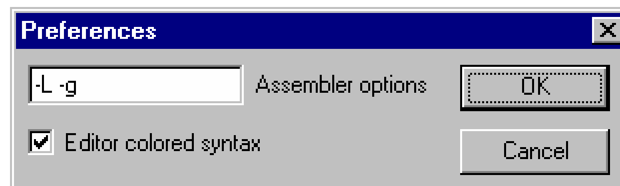
SLUT PÅ UPPGIFT 1.38.

Introduktion till ETERM 7

ETERM 7 är en utvidgad variant av ETERM 6.5. Du känner igen de flesta funktioner men du har dessutom en rad användbara tillägg. Vi ska här ge en introduktion genom att uppmärksamma skillnader från ETERM 6.5.

Inställningar

I File-menyn hittar du nu ett nytt alternativ: "Preferences":



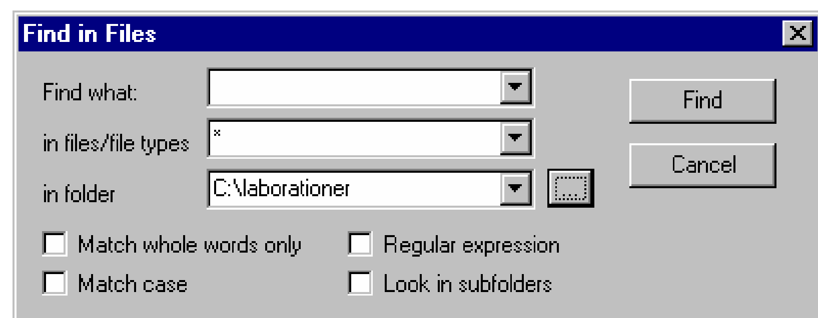
Assembler options:
Du kan styra assembleringen i viss utsträckning genom att ange "flaggor" ("växlar") . Flaggorna och deras användning finns beskrivna i hjälpsystemet under "Assemblers/Options".

Editor colored syntax:
Du kan stänga av syntaxfärgningen om du vill.

Sökfunktioner

Under Edit-menyn hittar du nya funktioner för att navigera, söka och ändra i en källtext. Alternativen "Go To Line", "Find" och "Replace" aktiveras då du arbetar med en källtextfil.

Med "Find in Files" kan du söka efter textmönster (symboler, namn etc) i flera filer. Om du väljer "Find in Files" öppnas följande dialogfönster:



Find What:
Här skriver du den text du vill söka efter

in file/files type:
Här anger du om det är någon speciell fil du avser. Du kan också ange sökmönster som exempelvis "*.S68" för att söka igenom källtextfiler.

in folder:
Här anger du det bibliotek som du vill söka igenom. Den lilla knappen till höger öppnar en dialogbox som låter dig välja ett nytt bibliotek.

Dina senaste val sparas och du kan bläddra fram dem med pilen till höger i respektive val.

Match whole words only:
Om du söker efter en precis textsträng ska du kryssa i denna ruta, annars kommer sökningen att ge träffar för alla textsträngar som innehåller din söktext.

Regular expression:
Reguljäruttryck, du kan använda meta-tecken så som '?' och '*' i din söktext, men du måste då kryssa i denna ruta.

Match case:
Kryssa i denna ruta om du vill att gemena och versaler ska behandlas som olika i sökningen. Om du inte kryssar denna ruta kommer exempelvis sökning på 'move' att ge träff även på 'MOVE'.

Look in subfolders:
Sökningen kan utföras även i alla underbibliotek. En sådan sökning kan i bland bli tidskrävande och om du har startat sökning och vill avbryta den kan du välja 'Stop Search' från menyn.

Välj 'Find' för att starta en sökning.

Källtextdebugger

ETERM's inbyggda simulator kan också användas tillsammans med *källtextedebuggern (SLD = Source Level Debugger)*. Debuggern är till för att förenkla testning av program.

Du startar debuggern från menyn Debug:

Härifrån startar du också simulatorn (*Machine Level*) eller öppnar ett terminalfönster (*Open Terminal*) om du inte vill arbeta med debuggern.

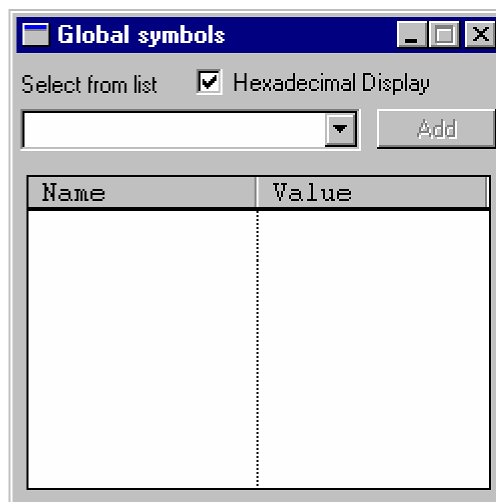
Du kan också välja att koppla en konfigurationsfil till simulatorn (*Select Simulator Configuration*). Denna läses då automatiskt in varje gång du startar debuggern.

Då debuggern startas, startas också simulatorn och två nya fönster:

- **Debug** - här ser du din källtext. I vänstra kanten finns ett grått fält där en röd pil markerar den rad som står i tur att exekveras.
- **Global symbols** - i detta fönster kan du inspektera och ändra värden på variabler i ditt program

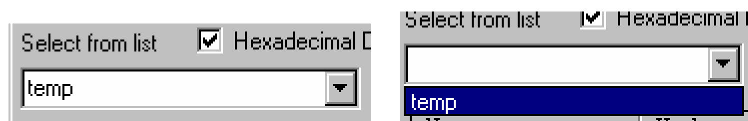
Debug: 'sld.s68'

```
*
*       SLD.S68
*
        ORG         $4000
startuplabel:
        MOVE.B      $FFFFF011,D0
        MOVE.B      D0,temp
        NOT.B       D0
        MOVE.B      D0,$FFFFF019
        JMP         startuplabel

* Variabel
temp:       DS.B        1
```

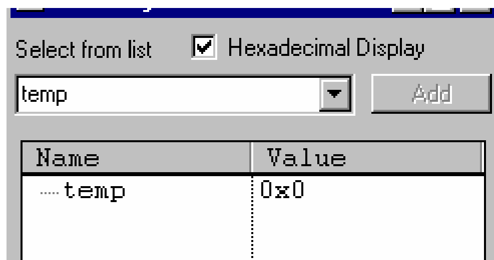


Vi börjar med fönstret 'Global symbols', i vårt lilla exempel har vi deklarerat en variabel 'temp' och vi vill nu inspektera denna:

Skriv variabelns namn eller välj från den lista som visas om du klickar på 'nedåtpilen':

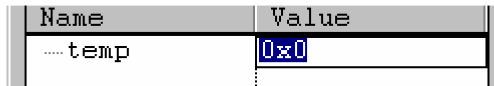


Klicka därefter på 'Add'.



Variabeln läggs nu till fönstret. Du kan ändra mellan visning på hexadecimal form respektive decimal form.

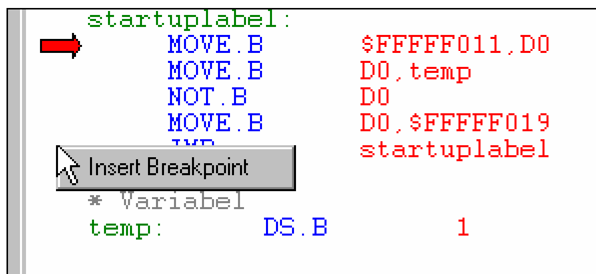
För att ändra variabelns värde, klicka på värdet i 'Value'-fältet:



Skriv in det nya värdet, exvis '0x20' och tryck <Enter>.

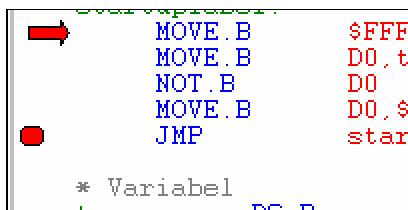
Debugfönstret

Fönstret består av två fält och en verktygslist. Det grå fältet till vänster används för att visa nästa exekverinspunkt och för att hantera brytpunkter.
Högerklicka i det grå fältet på en rad som innehåller en exekverbar instruktion, exempelvis:

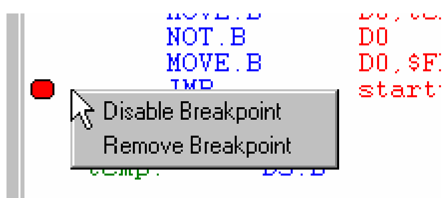


Sätt en brytpunkt på denna rad genom att klicka på pop-up alternativet 'Insert Breakpoint'.

Brytpunkten illustreras nu i det grå fältet med en röd rektangel.



Om du högerklickar igen på samma rad får du nu i stället alternativen 'Disable Breakpoint' och 'Remove Breakpoint'.

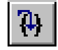


Välj 'Remove Breakpoint' om du vill ta bort brytpunkten.
Välj 'Disable Breakpoint' om du tillfälligt vill avaktivera brytpunkten men inte ta bort den.

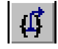
Debugfönstrets verktygslist

Längst upp i Debug-fönstret finns en rad knappar för att underlätta användningen.

- **Next** - Exekvera förbi - detta gäller *raden*. Om instruktionen är exempelvis "jump to subroutine" kommer hela subrutinen att utföras. Nästa exekveringspunkt blir raden efter subrutinanropet.



- **Step Into** - Instruktionsvis exekvering, används för att stega in i subrutiner.



- **Step Out** - För att snabbt utföra programmet till funktionens sista rad. Detta innebär 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.

- **Breakpoints** - Öppna dialogruta för brytpunktstabellen. Används för att ge en översikt av samtliga brytpunkter.

- **Restart** - Starta om programmet från början. Förbereder omstart av programmet. Eventuella brytpunkter och 'Watch'-variabler behålls.

Avsnitt 1 – Sammanfattning

Genom att ha arbetat dig igenom detta inledande avsnitt ska du förhoppningsvis ha skaffat dig grundläggande förståelse för hur en mikroprocessor fungerar. Du har sett exempel på en mikroprocessors *instruktionsuppsättning*, observera dock att dessa exempel inte på något sätt är heltäckande. Förmodligen behöver du framledes konsultera såväl en komplett *instruktionslista* som *fler exempel*. Du har också fått lära dig att hantera grundläggande funktioner hos en programutvecklingsmiljö (Textredigering, Assemblering och Simulering).

Du bör (självständigt) ha utfört alla övningar och har därmed också samlat på dig ett antal källtextfiler, av vilka fler kan komma väl till pass då det är dags för första laborationstillfället.

De påföljande avsnitten kommer att ställa allt högre krav på förståelse, om du känner att du fortfarande inte förstått dina lösningar, eller vad dessa syftade till, bör du därför överväga repetition av de moment som skapade problem.

Om du fortfarande, efter att verkligen ha försökt, tycker att du är osäker, ska du konsultera en kursassistent (eller kursansvarige). Du ska då ha förberett dig genom att komma med konkreta frågor så att din lärare också kan få en rimlig chans att hjälpa dig.

Avsnitt 2

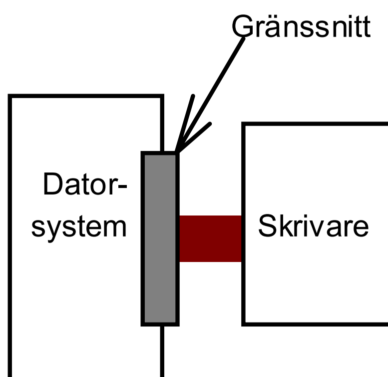
En enkel Skrivarport

Syften:

I detta avsnitt ger vi en introduktion till adressavkodning. Vi studerar synkronisering med hjälp av hårdvara. Vi ger en introduktion till "avbrott".
Den beskrivna "hårdvaran" i form av en enkel "Printer" finns inte som tillbehör till laborationsdatorn. Moment med "printern" är avsedda enbart för övningar med simulatorn.

Målsättningar:

Målsättningen är att du, på egen hand, utfört alla uppgifter i avsnittet.



Inledning

I detta avsnitt koncentrerar vi oss på parallell in- och utmatning. För att göra detta exemplifierar vi med en port (ett *gränssnitt*) till en enkel skrivare. Vi kommer att bygga upp porten bit för bit, både hård och mjukvarumässigt, och studera problem, fördelar och nackdelar med olika lösningar.

Vi måste ge vissa förutsättningar för skrivaren. Vår skrivare är från början en mycket enkel skrivare:

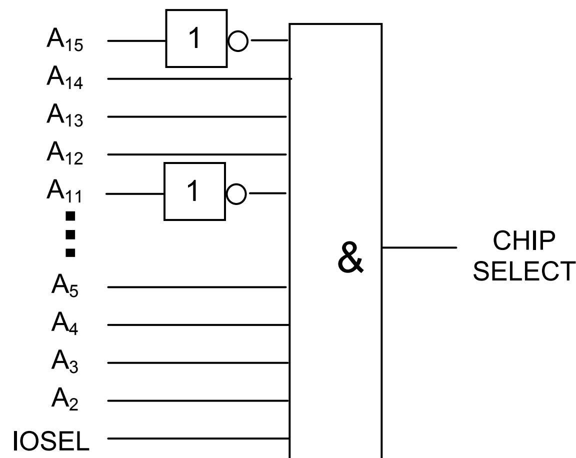
- Den kan endast arbeta med ett tecken i taget. Större skrivare kan hämta in många sidor av ett dokument innan den börjar på utskriften, medan vår hämtar ett tecken i taget för att skriva det innan nästa tecken hämtas.
- Det finns inledningsvis inga handskaknings-signaler från skrivaren som indikerar exempelvis slut på papper (*Paper Out*), klar att ta emot nytt tecken (*Ready*), mm.

Vi förutsätter att skrivaren klarar av att ta emot och skriva 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. Gränssnittet kan även fungera som en *förstärkare* för signaler i den förhållandevis långa kabeln mellan skrivare och mikrodatorsystem.

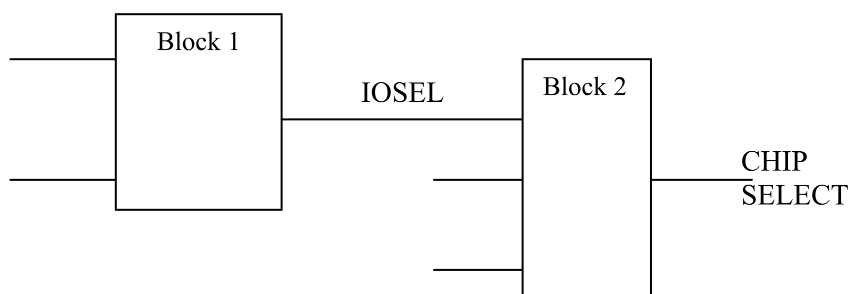
Adressavkodning

Den enklaste formen av ett *gränssnitt*, dvs. en sammankoppling av datorsystemets bussar med någon buss i en yttre (perifer) enhet består av ett register anslutet till processorns bussar. Den nödvändiga hårdvarukopplingen består då av *adressavkodningslogik* och ett *register*. Avkodningslogikens funktion är att bilda en aktiveringssignal, ofta kallad "Chip Select" för registret. Följande figur visar *exempel* på en enkel form av adressavkodningslogik.



Princip för enkel adressavkodningslogik

Principen går ut på att adressbussens signaler grindas samman. I detta fall med hjälp av NOT och AND-logik, på ett sätt som gör att utsignalen CHIP SELECT aktiveras endast för vissa adressintervall. Observera också den nedersta signalen IOSEL, som bildats på ett liknande sätt men med hjälp av adressbussens övre del (A_{16}-A_{23} för MC68000, A_{16}-A_{31} för MC68340). Adressbussen avkodas följakligen i två steg. I det första steget genereras en signal som är aktiv för *samtliga* gränssnitt i systemet. I ett andra steg genererar vi unikt aktiva signaler för varje separat gränssnitt. I ett tredje steg kommer vi senare att lägga till avkodning för de minst signifikanta adressledningarna men just nu nöjer vi oss med de båda första. Följande figur illustrerar en övergripande bild av avkodningslogiken:



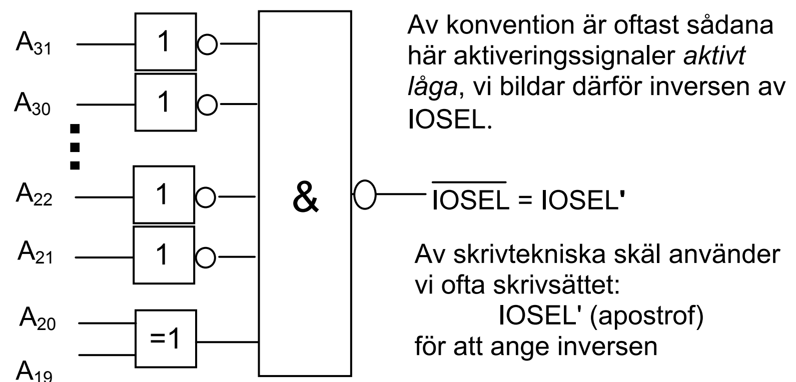
För att generera en signal som aktiverar vårt gränssnitt måste alltså först och främst signalen IOSEL aktiveras, dessutom krävs också att adressledningarna in till block 2 antar värden för det adressintervall vi valt för vårt gränssnitt. Vi ska nu se exempel på hur ett sådant adressintervall bestäms och därmed också hur den fullständiga avkodningslogiken ska se ut.

Signalen IOSEL bildas hos *MC68* för adressintervallet 80000-100000 (hexadecimalt). Följande tabell visar de värden som adressledningar A_{19}-A_{31} måste anta för att detta ska vara uppfyllt.

	A_{31}	A_{30}	A_{29}	A_{28}	A_{27}	A_{26}	A_{25}	A_{24}	A_{23}	A_{22}	A_{21}	A_{20}	A_{19}	A_{18}-
00080000	0	0	0	0	0	0	0	0	0	0	0	0	1	X
00100000	0	0	0	0	0	0	0	0	0	0	0	1	0	X

med X i tabellen menas "don't care", dvs. adressledningen kan anta vilket värde, NOLL eller ETT, som helst.

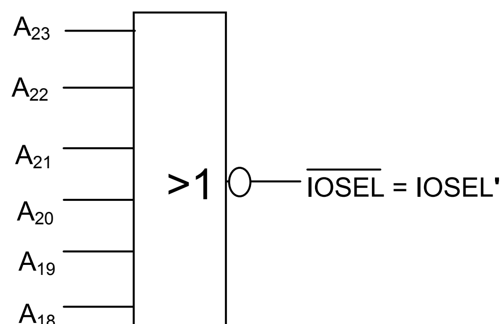
Ett möjligt logikblock, för att bilda aktiveringssignalen IOSEL i adressintervallet är exempelvis:



Signalen IOSEL bildas hos *MD68k* för adressintervallet 400000-FFFFFF (hexadecimalt). Följande tabell visar de värden som adressledningar A_{18}-A_{23} måste anta för att detta ska vara uppfyllt.

	A_{23}	A_{22}	A_{21}	A_{20}	A_{19}	A_{18}	A_{17}-
400000	0	0	0	0	0	1	X
FFFFFF	1	1	1	1	1	0	X

Om någon av adressledningarna A_{23} t.o.m. A_{18} är 1 har vi alltså en adress i vårt angivna adressintervall på bussen. Avkodningslogiken kan enkelt implementeras med NOR-logik:



Vi har sett hur 'Block 1' kan konstrueras för att bilda aktiveringssignal för *något* gränssnitt hos datorsystemet. Den resulterande signalen har vi kallat IOSEL. Den minnesarea som aktiveras på detta sätt kallar vi ofta 'IO-Area'. I ETERMs simulator kan du definiera fyra olika sammanhängande IO-areor. För vår fortsättning nöjer vi oss dock med att använda

standardinställningarna, dvs de som anpassats efter laborationsdatorerna *MC68* respektive *MD68k*.

Vi går nu vidare med avkodningsblock 2. Vi vill placera in vårt skrivargränssnitt någonstans i IO-arean. Vi behandlar först gränssnittet i *MC68* och återkommer strax till *MD68k*.

EXEMPEL
Låt oss för enkelhetens skull välja offset 0 i IO-arean, dvs vårt skrivargränssnitt hamnar på de första adresserna i IO-arean. Hos *MC68* innebär detta adress 80000. Låt oss vidare, redan från början, reservera 8 adresser för gränssnittet, detta innebär att de tre första adressledningarna utelämnas ur denna avkodning, de blir alltså "don't care". Vi kan senare använda dessa adressledningar, exempelvis för att skilja mellan adresser till olika register i gränssnittet. Vi måste alltså avkoda adressledningar från A_3 upp till A_{18} för att kunna bilda en unik aktiveringssignal. Den adress som nu bildas kallar vi BAS-adress för gränssnittet.

	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_3
BAS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

UPPGIFT 2.1
Rita, med hjälp av NOT och NAND-logik en möjlig adressavkodninslogik för detta adressval. Glöm inte IOSEL.

Slut på Uppgift 2.1.

UPPGIFT 2.2

Ange adressledningarnas värden om vi i stället vill placera vårt gränssnitt på adress 8842C (BAS = 842C).

	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_3
BAS															

Rita, med hjälp av NOT och NAND-logik en möjlig adressavkodninslogik för detta adressval. Glöm inte IOSEL.

Slut på Uppgift 2.2.

Vi övergår nu till motsvarande avkodningslogik för *MD68k*. Vi har redan visat hur IOSEL-signalen kan bildas och noterar att adressledningar A_{17} t.o.m A_3 måste ingå. Om vi, precis som i föregående exempel, vill placera gränssnittet först i IO-arean blir då avkodningslogiken för vår CHIP SELECT signal:

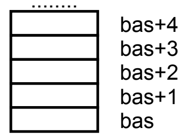
	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_3
BAS	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Som vi ser blir resultatet identiskt om vi bortser från A_{18}, som ju i *MD68k*'s fall redan ingår vid bildandet av IOSEL.

Det tillstöter dock en annan liten komplikation som hänger samman med databussens storlek. Hos *MC68*, som är byggd kring MC68340 är databussen 8 bitar bred, dvs vi har ett direkt *byte*-adresserat minne. Hos *MD68k*, som är byggd kring MC68000 har vi i stället en 16 bitar bred databuss. Ett gränssnitt, med 8 bitars bredd, måste följaktligen placeras på endera D_0-D_7 eller D_8-D_{15}. Hos *MD68k* har denna anslutning gjorts via D_8-D_{15}. Detta betyder ingenting för adressavkodningen men konsekvensen blir att gränssnittet alltid måste byte-adresseras på *udda* adress.

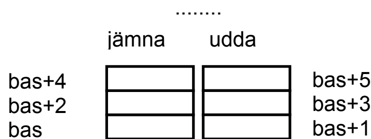
Vi sammanfattar databussbreddens konsekvenser:

MC68 har *byte*-organiserat minne, dvs systemet arbetar med en 8-bitars databuss.



Olika portar i gränssnittet får konsekutiva adresser.

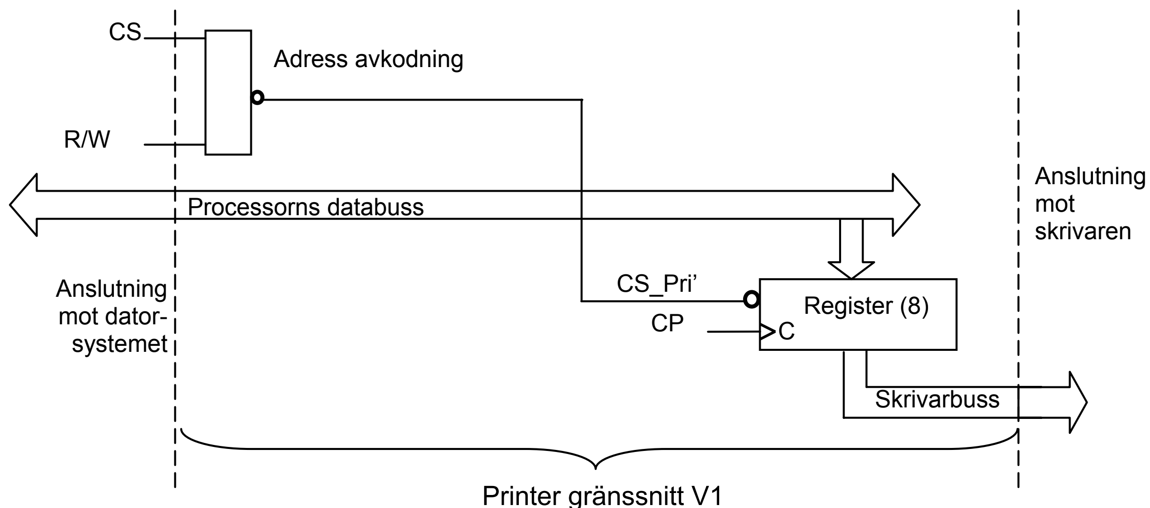
MD68k har *word*-organiserat minne, 8-bitars IO-enheter är anslutna till undre delen av databussen, därför kan endast UDDA adresser uppträda som IO-portar



Olika portar i gränssnittet får alltid udda adresser.

Printergränssnitt V1

Vi har nu bildat även "Block 2" i adressavkodningslogiken och det är dags att se hur dessa signaler används i gränssnittet till vår skrivare. Betrakta följande figur:



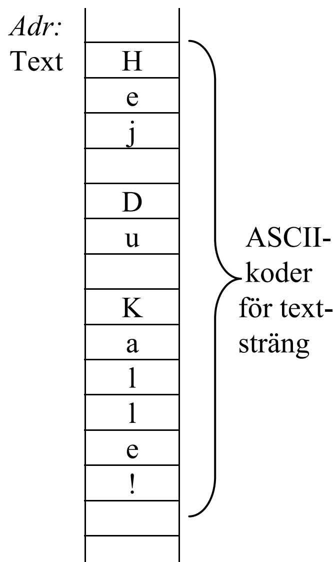
Signalen CS är här den signal som kommer från Block 2 och tillsammans med en aktiv skrivsignal (R/W) i sin tur bildar en aktiveringssignal (CS_Pri) för ett register anslutet mellan datorsystemets databuss och skrivarens buss. Vid en klockpuls och om CS_Pri är aktiv, överförs alltså ett åtta bitars värde från processorns databuss till skrivaren via skrivarbussen.

EXEMPEL
Med portdefinition
```
PRINTER    EQU   $80000
```
och instruktionen
```
      MOVE.B     #$30,(PRINTER).L
```
överförs hexadecimala värdet 30 till skrivaren i ett *MC68* system. Samma sak kan göras i ett *MD68k* system om portdefinitionen ändras enligt:
```
PRINTER    EQU   $400001
```

Innan vi behandlar ett tänkbart utskriftsprogram måste vi anta vissa förutsättningar för hur texten som skall skrivas ut lagrats i minnet (se marginalen). Förutsätt att någon programrutin har placerat ASCII-koderna för texten i minnet och att den är lagrad i sekvens från adressen *Text* och framåt. Exemplet visar texten *Hej Du Kalle!* som vi vill skriva ut. Betrakta följande programsekvens.

```
* Printer V1.0
        ORG        $4000
        MOVEA.L    #Text,A0         Pekare till textsträng
Loop    MOVE.B     (A0)+,D0         Läs tecken
        MOVE.B     D0,(PRINTER).L    Skriv ut
        BRA        Loop
```

I det enkla programförslaget används register A0 för att peka ut tecknet som skall skrivas. När det första tecknet läses i programslingan uppdateras A0 för att peka ut nästa tecken innan detta skickas till skrivaren.



UPPGIFT 2.3

Redigera en källtextfil under namnet `PRINTERV1_0.S68`, Assemblera den och ladda till simulatorn.

Ledning, assemblerdirektiv för att skapa textsträngen:
```
Text    DC    "Hej Du Kalle!"
```

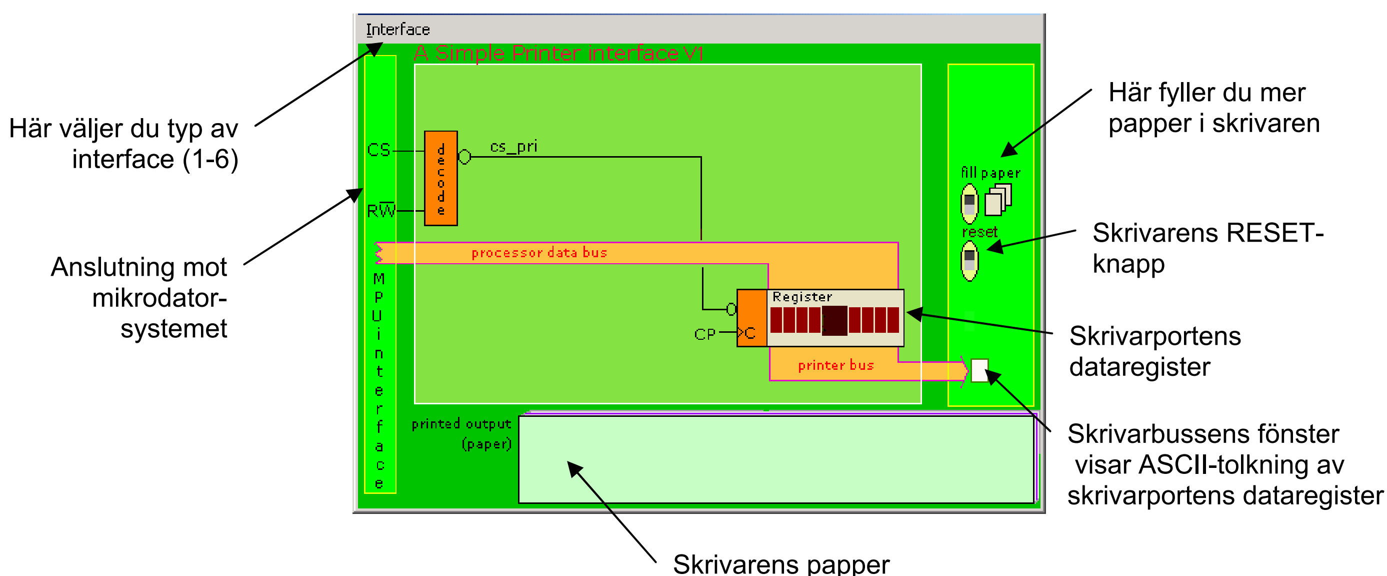
Undersök innehållet i simulatorns minnesfönster och programfönster och gör dig bekant med vart kod och data (textsträngen) placeras. Är du osäker kan du även studera listfilen.

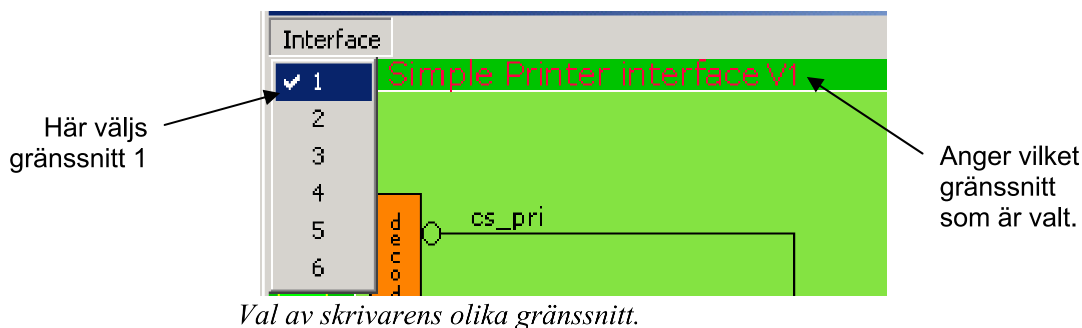
Slut på Uppgift 2.3.

UPPGIFT 2.4

Anslut nu IO-simulatorn 'Simple printer'. Använd adressen $80000 för *MC68* respektive 400001 om du använder *MD68k*, som adress till skrivarens dataregister (`PRINTER`). Studera bilden på skärmen och jämför med följande figur: (nästa sida).

Observera att det finns flera olika gränssnitt (”Interface”) i simulatorn för skrivaren. De är numrerade 1-6 och vi refererar till de olika gränssnitten som ”PrinterV1” osv till ”PrinterV6”. Se alltid till att du använder rätt gränssnitt.





Val av skrivarens olika gränssnitt.

Testa programmet med 'Run'. Studera innehållet i register A0 och att detta hela tiden inkrementeras. Processorn exekverar den lilla programslingan som hämtar tecken och skickar dessa till skrivaren. Tyvärr avslutas inte utskriftsrutinen då hela textsträngen är utskriven och så småningom fylls hela pappret ut, förmodligen med ren skräptext. För att få stopp på skrivaren måste du klicka på skrivarens RESET-knapp.

Då pappret tar slut i skrivaren klickar du på knappen ’fill paper’.

Slut på Uppgift 2.4.

Adr:		
Text	H	ASCII-koder för text-sträng
	e	
	j	
	D	
	u	
	K	
	a	
	l	
	l	
	e	
	!	
	EOT	Slutmarkering

Det krävs uppenbarligen någon form av slutmarkering för textsträngen i minnet. Se marginalen. Vi väljer markeringen **EOT** (End Of Text) som har ASCII-koden $04. EOT ska nu användas av programrutinen för undersöka när det sista skrivbara tecknet skickats till skrivaren:

Vi uppdaterar vårt program Printer V1.0 till Printer V1.1 där vi nu tar hänsyn till slutmarkeringen. Se nedan

```
* Printer V1.1
* Du måste definiera symbolen EOT, exempelvis enligt:

EOT     EQU        04                End Of Text

        ORG        $4000
        MOVEA.L    #Text,A0          Pekare till textsträng
Loop    MOVE.B     (A0)+,D0          Läs tecken
        CMPI.B     #EOT,D0
        BEQ        Stop
        MOVE.B     D0,(PRINTER).L    Skriv ut
        BRA        Loop

Stop    NOP
        BRA        Stop

Text    DC         "Hej Du Kalle!"   Textsträng
        DC.B       EOT               Lägg till slutmarkering
```

PrinterV1_1

Init

Läs nästa tecken

J

Tecken=
EOT?

N

Överför Data
till Skrivaren

Stop

Hej Du Kalle !!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!

Register A0 används fortfarande för att peka ut "nästa" tecken som skall skrivas. Under varje varv i slingan skrivs ett ASCII tecken (en *byte*) ut. När tecknet EOT läses från strängen utförs ett hopp till `Stop`, därefter fortsätter programexekveringen i en oändlig slinga som inte uträttar någonting (Instruktionen `NOP` betyder No OPeration).

UPPGIFT 2.5

Kopiera filen `PRINTERV1.S68` till `PRINTERV1_1.S68` och ändra enligt exemplet ovan.
Assemblera `PRINTERV1_1.S68` och ladda till simulatorn. Välj läget 'Run' när du testar.

Du bör nu se att register A0 inkrementeras så länge som utskriften pågår. När utskriften är klar ”hänger” programmet vid `Stop`.

Slut på Uppgift 2.5.

Förmodligen fylls nu pappret med utropstecken. Betrakta skrivarbussens fönster och värdet som placerats i skrivarens dataregister. Vår enkla skrivare läser hela tiden skrivarbussen värde och skriver detta på pappret. Då registret innehåller ASCII-koden för ett utropstecken kommer därför detta att fylla pappret.

Vi skall ta hand om detta problemet längre fram. Först skall vi studera de olika arbetstakter som våra två system, dator och skrivare, har.

UPPGIFT 2.6

Starta om programmet i 'Run Fast' och studera förloppet.

Vad skrivs på skrivarens papper?? ______________________________

____

Försök förklara varför texten inte skriv ut korrekt _______________

____

____

____

Slut på Uppgift 2.6.

Fundera lite på hur snabbt (hur ofta) vi skriver ett tecken till skrivaren. Programslingan ”Loop” upptar 5 instruktioner vilket inte tar speciellt lång tid jämfört med skrivaren som skriver ut 4 tecken per sekund.

UPPGIFT 2.7

Gör en uppskattning av utskriftshastigheten, alltså hur många tecken per sekund som mikrodatorsystemet skickar till skrivaren.

Antag för enkelhetens skull att varje instruktion tar 4 µs (µs = 10^{-6}s) att utföra.
Programslingan omfattar 5 instruktioner. Hur lång tid tar hela slingan?

____

Hur många tecken skrivs då ut per sekund? ____________________

Jämför detta med skrivarens prestanda och ange hur många gånger snabbare mikrodatorsystemet är än skrivaren.

Hur många gånger snabbare? ______________________________

Slut på Uppgift 2.7.

För att komma till rätta med problemet måste vi alltså synkronisera det snabbare mikrodatorsystemet till den långsamma skrivaren. Man skulle kunna tänka sig att lägga in en fördröjning i utskriftsrutinen för att få denna att "komma i takt" med den långsammare skrivaren. För varje varv i programslingan ”Loop” kan vi utföra ett subrutinanrop till en rutin som fördröjer exekveringen i 250 ms (Jämför avsnitt 1).

Koden borde *i princip* fungera eftersom ett tecken skrivs var 250:e ms, vilket också är skrivarens arbetstakt. Detta gäller under förutsättning att skrivaren är klar att ta emot ett tecken precis när det första skrivs ut. Startögonblicket måste därför *synkroniseras*.

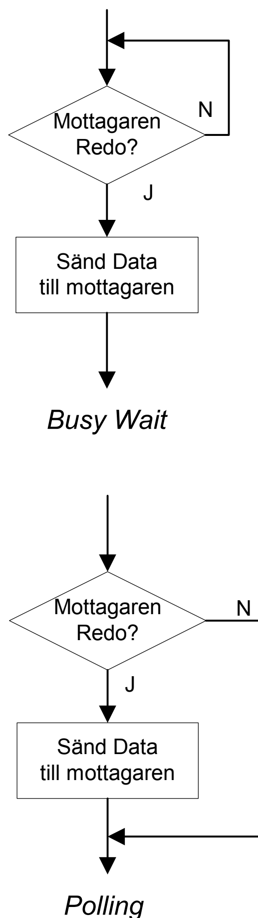
Mikrodatorsystemets klocka går lite snabbare än skrivarens: *Hej Du alle!*

eller långsammare: *Hej Duu Kalle!*.

Om vi skriver ut en *lång* utskrift, även om startögonblicket kan synkroniseras så kommer det sannolikt att bli fel efter ett tag eftersom det är mycket svårt (i praktiken omöjligt) att få två olika klockor (en i skrivaren och en i datorsystemet) att gå synkront (gå exakt lika fort). Felet kommer att uppträda som missade tecken eller dubbletter. Vårt exempel skulle på papperet kunna se ut, `Hej Du alle!` eller `Hej Duu Kalle!`. Problematiken återkommer i alla former av ovillkorlig överföring dvs. då datorsystemet (*sändaren*) skickar data till skrivaren (*mottagaren*) utan att ta hänsyn till om mottagaren är redo att ta emot ett (eller *nästa*) tecken.

Villkorlig överföring

Vi skall nu övergå till olika typer av *villkorlig överföring*. Det krävs då någon form av signalering (handskakning) mellan sändaren (mikrodatorsystemet) och mottagaren (skrivaren).



Vi ska här behandla två principer för villkorlig överföring nämligen ”upprepad statustest” (Busy Wait) och ”rundfrågning” (Polling). Se figurer i marginalen. Den översta betecknas ’Busy Wait’ och innebär att man inväntar att mottagaren skall bli redo att acceptera data. Denna mekanismen innebär att mikrodatorsystemet blir ”hängande” till mottagaren är redo. Direkt efter det att mottagaren är redo skickas data.

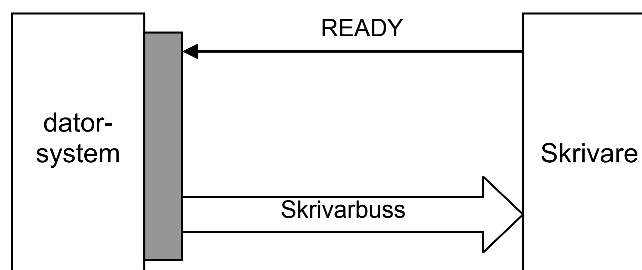
Vid ’Polling’ däremot testas om mottagaren är redo, om den inte är det fortsätter mikrodatorsystemet med annat arbete (exekverar övrig kod). Detta innebär oftast att mottagaren får vänta en kort stund på data efter det att den blivit redo.

I vår konstruktion, så här långt, är vi tvungna att välja det första alternativet, ’Busy Wait’, för vår skrivarport. Orsaken 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 till skrivaren då den blivit redo.

Om vi försöker använda ’Polling’ i vår konstruktion är det högst sannolikt att mikrodatorsystemet är upptaget med andra sysslor (exekverar övrig kod) när skrivaren skall läsa nästa tecken. Detta innebär att skrivaren då läser skrivarbussen som innehåller det ”förra tecknet”. Detta visar sig som dubbletter i utskriften.

Printergränssnitt V2

Vi skapar nu nya förutsättningar för att lösa de problem vi diskuterat. Vi inför en READY-signal i vårt system.

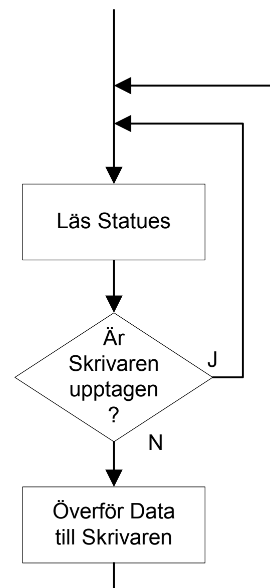


Skrivarport med READY-signal.

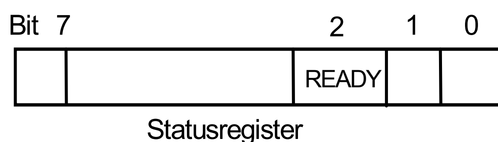
READY-signalen definieras enligt:

- 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.

Med denna konstruktion kan vi nu i stället utforma programmet exempelvis enligt flödesplanen i marginalen. Vi undersöker om skrivaren är upptagen innan nästa tecken skickas till skrivaren.



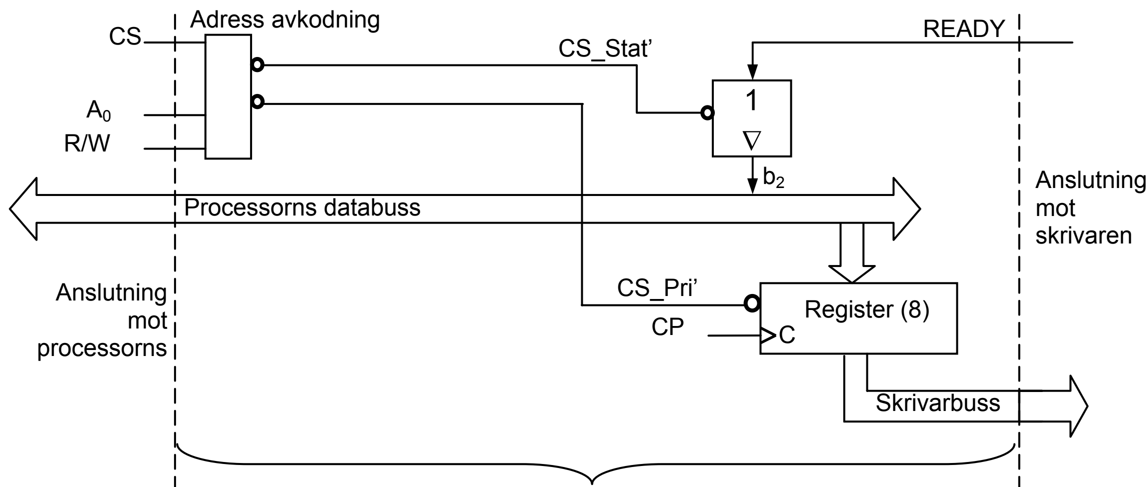
Låt oss införa ett statusregister i skrivarporten där vi kan läsa READY-signalen till processorn. Följande figur visar hur skrivarens READY-signal är ansluten till bit 2 på mikrodatorsystemets databuss. Statusregistret utgörs av en vanlig three-state buffert för bit 2.



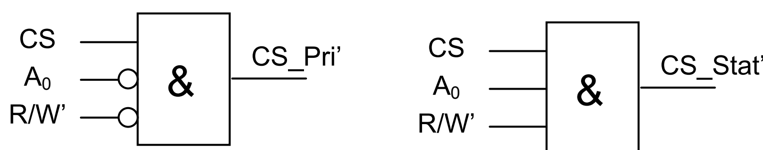
Programmerarens bild av Statusregistret.

Dataregister = basadress
Statusregister = basadress+1

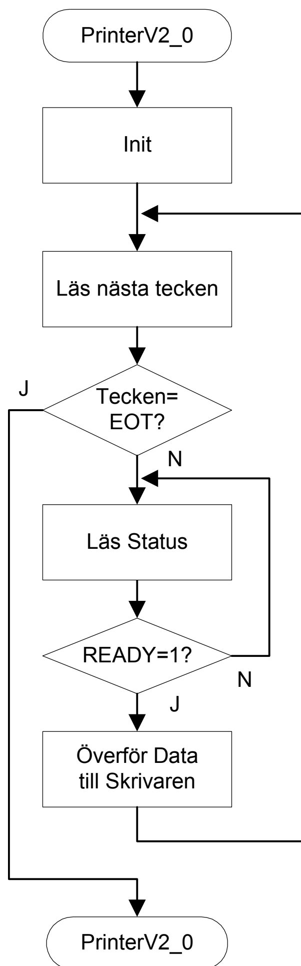
Betrakta figuren 'Printer gränssnitt V2' nedan som visar skrivarporten. Kom i håg att signalen CS från mikrodatorsystemet är aktiv för ett adressintervall som vi tilldelat gränssnittet. Genom att utnyttja den minst signifikanta adressledningen, A_0 i printerportens interna adressavkodningslogik kan vi nu bilda signalerna 'CS_Stat' för statusregistret och 'CS_Pri' för skrivarens dataregister. Se figur '*Intern adressavkodning i skrivaranslutningen*' nedan.



Printer gränssnitt V2



Intern adressavkodning i skrivaranslutningen



Med denna nya hårdvarukonstruktion kan nu en programkonstruktion efter det flödesdiagram som visas i marginalen användas.

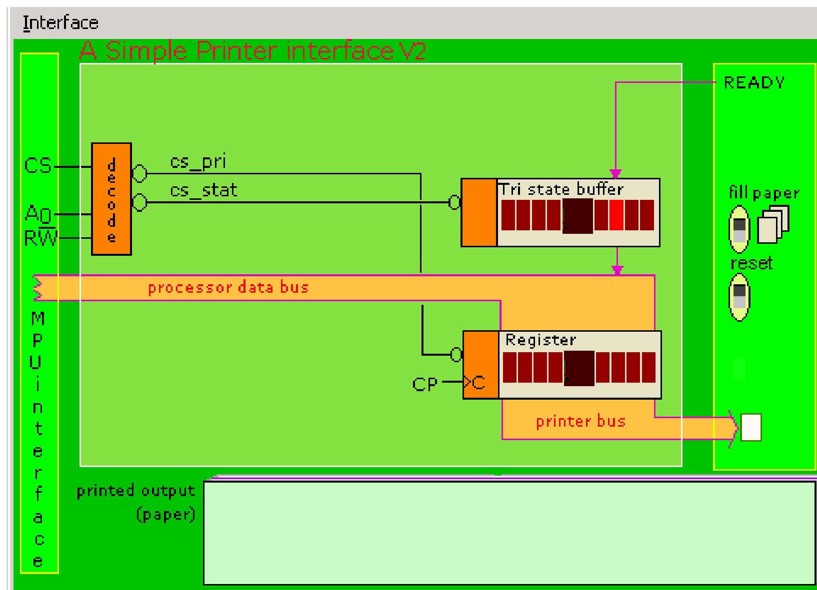
Processorn läser oupphörligt statusregistret och undersöker sedan **READY**-biten (signalen) från skrivaren tills denna indikerar att ett nytt tecken kan skickas till skrivaren. En sådan programkonstruktion kallas alltså "Busy Wait".

```
* Printer V2.0
* Definitioner för MC68
PRINTER           EQU         $80000
PSTATUS           EQU         $80001

* Definitioner för MD68k
PRINTER           EQU         $400001
PSTATUS           EQU         $400003
EOT               EQU         4

      ORG         $4000
      MOVEA.L     #Text,A0
Loop
      MOVE.B      PSTATUS,D0  Läs status
      BTST.B      #2,D0       READY=1 ?
      BEQ         Loop

      MOVE.B      (A0)+,D0
      CMPI.B      #EOT,D0
      BEQ         Stop
      MOVE.B      D0,(PRINTER).L
      BRA         Loop

Stop  NOP
      BRA         Stop

Text  DC          "Hej Du Kalle!"
      DC.B        EOT
```

UPPGIFT 2.8

Skapa en källtextfil `PRINTERV2_0.S68` och redigera enligt ovanstående exempel. Assemblera, rätta eventuella fel och ladda till simulatorn.

Ändra nu skrivarsimulatorn till 'Interface 2' och jämför bilden med figuren på nästa sida.



Simulatorbild för Printer gränssnitt V2

Skrivaren är nu utrustad med en READY-signal. Denna är ansluten till en three-state buffert i gränssnittet som kan läsas via bit 2.

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

Välj nu att starta om simulatorn i läget ’Run’ och testa. Slutligen startar du om på nytt och testar ’Run Fast’.

Du får olika utskrifter beroende på om du använder 'Run' eller 'Run Fast'. Försök förklara varför:

____

____

____

Slut på Uppgift 2.8.

Orsaken till problemet ovan är sannolikt att vi inte testat att skrivaren blivit upptagen efter det att den blivit redo. Studera flödesdiagrammet för PrinterV2.0 på förra sidan. Direkt efter det att skrivaren blivit ledig skickas nästa tecken. Processorn hämtar ”nästa tecken” och omedelbart efter detta testas på nytt om READY=1.

Eftersom skrivaren är betydligt långsammare än mikrodatorsystemet behåller den sin READY-signal hög tillräckligt länge så att när processorn läser statusregistret på nytt så är READY fortfarade ettställd. Detta innebär att mikrodatorsystemet skickar iväg flera tecken åt gången utan att skrivaren hinner läsa dessa. Skrivaren kommer därför att missa ett antal tecken.

Ett nytt programförslag som är anpassat till printerporten med READY-signal visas nedan. Observera att efter vi skrivit ut ett tecken till skrivaren måste programmet invänta att READY går låg

PrinterV2_1

Init

Läs nästa tecken

Tecken= EOT?

J

N

Läs Status

READY=1?

N

J

Överför Data till Skrivaren

Läs Status

READY=0?

N

J

PrinterV2_1

för att inte skriva ut ett nytt tecken direkt. Detta medför att programmet består av två slingor, en som testar att READY går hög och en som testar att READY-signalen går låg. Studera även flödesplanen i marginalen.

```
* Printer V2.1

      MOVEA.L     #Text,A0
Loop
      MOVE.B      PSTATUS,D0
      BTST.B      #2,D0
      BEQ         Loop

      MOVE.B      (A0)+,D0
      CMPI.B      #EOT,D0
      BEQ         Stop
      MOVE.B      D0,(PRINTER).L
Loop2
      MOVE.B      PSTATUS,D0
      BTST.B      #2,D0
      BNE         Loop2
      BRA         Loop

Stop  NOP
      BRA         Stop
```

UPPGIFT 2.9

Redigera en ny källtextfil `PRINTERV2_1.S68` enligt förslaget ovan. Assemblera och rätta eventuella fel.

Testa nu din lösning och verifiera att textsträngen skrivs korrekt på skrivarens papper. Testa både ’Run’ och ’Run Fast’.

Vad händer vid 'Run' ?

Vad händer vid 'Run Fast' ?

Slut på Uppgift 2.9.

Nu har vi synkroniserat datorsystemet till skrivarens arbetstakt och på så sätt kommer inte skrivaren att varken skriva ut dubbletter (vid 'Run Fast') eller att missa tecken. Lösningen är dock ännu ej komplett.

Det sista tecknet som skrivs till printerportens dataregister ligger kvar och kommer att läsas ett antal gånger av skrivaren. Varje gång efter att skrivaren signalerat READY kommer den att läsa utropstecknet, dvs. det sista tecknet i textsträngen, som ju ligger kvar i skrivarportens dataregister. I vårt exempel kommer därför utropstecken att fylla hela papperet.

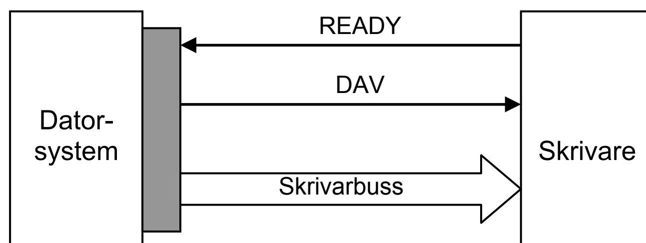
Problemet kan lösas på två olika sätt. Det ena är att vi förutsätter att skrivaren förstår ett antal styrtecken. Exempelvis kan vi då skicka ett stopptecken till skrivaren så att utskriften upphör (jämför med EOT).

```
Hej Du Kalle !!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
```

En annan möjlighet är att vi utrustar skrivaren med en ny handskakningssignal som indikerar att det finns ett nytt giltigt tecken på skrivarbussen som skrivaren kan läsa in och trycka på pappret.
Vi väljer det senare alternativet och övergår därför till printerport Version 3.

Printergränssnitt V3

Vi inför nya förutsättningar för skrivaren och antar att den nu även är utrustad med en insignal **DAV** (Data AVailable).

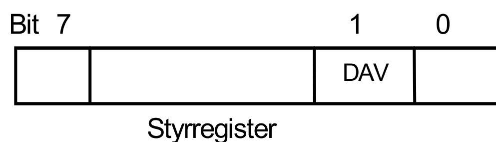


Blockdiagram över printerport version 3.

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

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

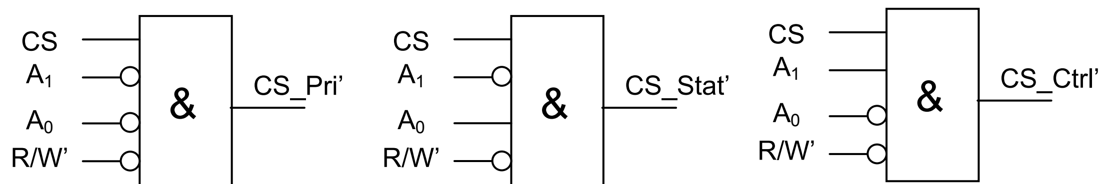
Skrivarporten måste nu förses med en extra utport för DAV-signalen. Utporten utgör skrivarens *styrregister* och vi betecknar detta 'PCtrl' (Printer Control). Se nedanstående figur som visar hur DAV-signalen ansluts via registret till bit1 på mikrodatorsystemets databuss.



Dataregister = basadress
Statusregister = basadress+1
Styrregister = basadress+2

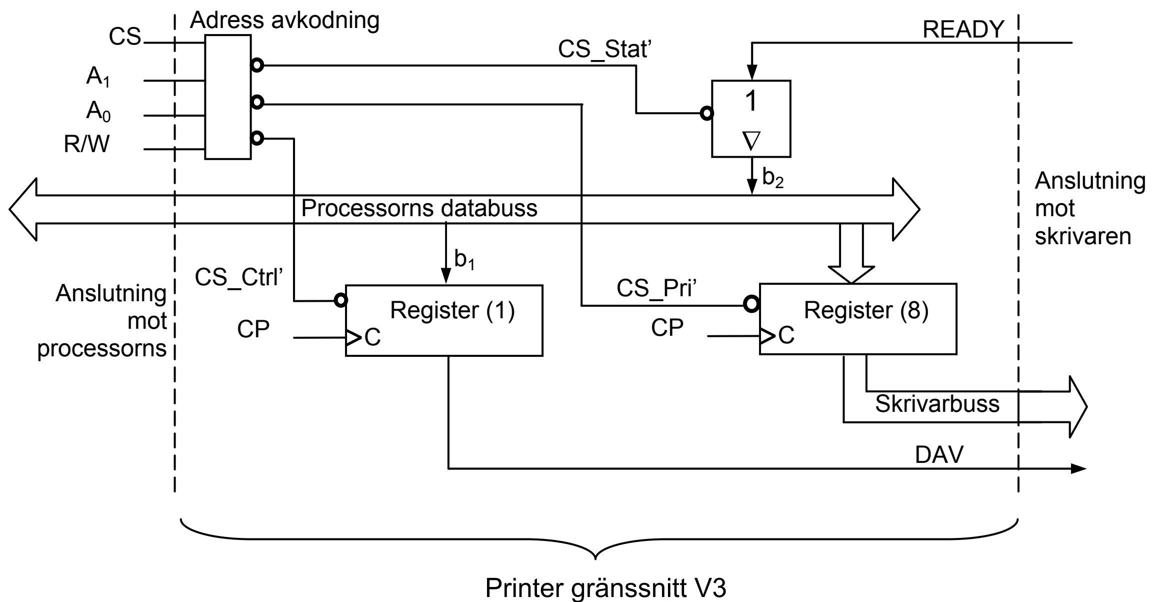
Programmerarens bild av styrregistret

När printerporten nu utökas med ännu ett register så krävs att både adressbitarna A_1 och A_0 ingår i den interna adressavkodnings-logiken.



Intern adressavkodningslogik för printerport V3

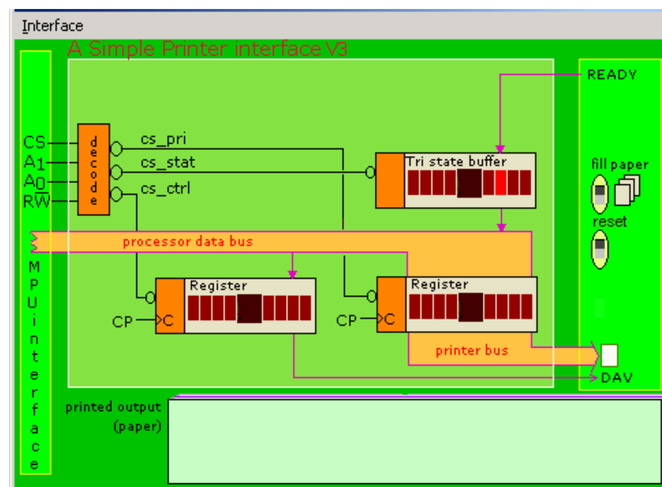
Betrakta hårdvarukopplingen i figuren nedan. Styrregistret har en bit ansluten (b_1) till processorns databuss.



UPPGIFT 2.10

Starta simulatorn för skrivarporten och välj 'Interface 3' och jämför med följande figur. Simulatorbilden är utökad med ett register för DAV-signalen.

Slut på Uppgift 2.10.



Simulator för printerport version 3.

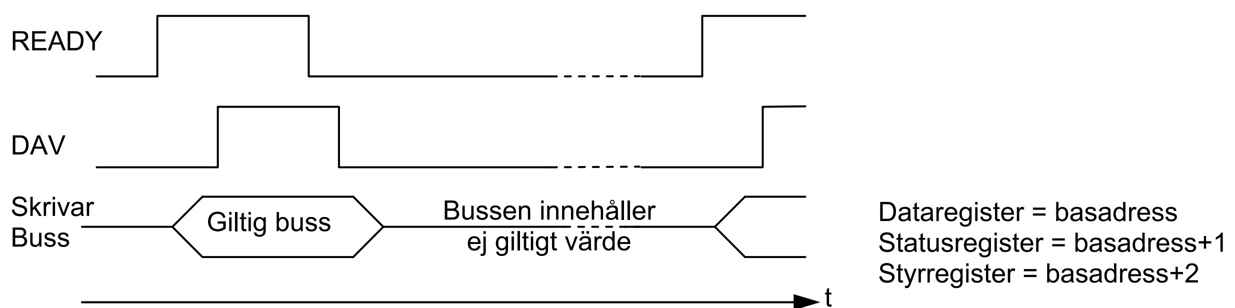
Vi har nu två handskakningssignaler READY och DAV. Följande punkter anger hur dessa två system, den snabba mikrodatorn och den långsamma skrivaren, skall kommunicera för att uppnå en säker överföring av data. På så sätt kommer inga dubbletter eller missade tecken att finnas i utskriften.

Betrakta följande figur som beskriver händelseförloppet i text och nästa figur som illustrerar samma sak i ett tidsdiagram. Vi förutsätter 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
	Inväntar READY=1
2	När READY=1 skrivs nästa tecken till skrivarens dataregister. Sätter DAV=1
	Inväntar READY=0
4	När READY=0 nollställs DAV som indikation på att det inte finns giltigt tecken på skrivarbussens

	Händelser i skrivaren
	Skrivaren är upptagen med att skriva ut ett tecken. READY=0.
1	Skrivaren är redo för nästa tecken och sätter READY=1
	Inväntar DAV=1
3	Ser att DAV=1. Läser nytt tecken från skrivarbussen. Signalerar upptagen, READY=0.
	Skrivaren är upptagen med att skriva ut ett tecken. READY=0.

Händelser i datorsystem och skrivare vid handskakning



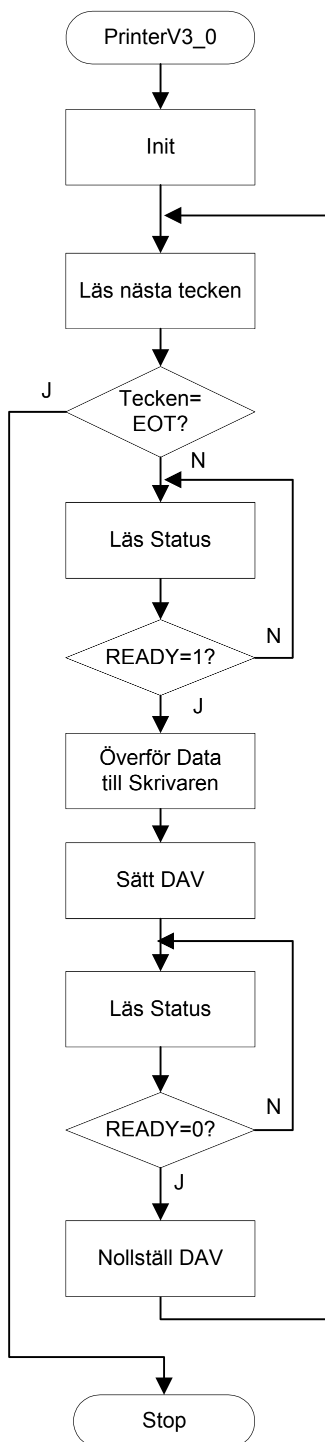
Tidsdiagram över handskakningsförloppet

På det sätt vi definierat signalerna så har skrivarbussen giltigt värde under den tid som visas i tidsdiagrammet.

Observera att skrivarbussen har ett giltigt värde innan DAV ettställs. Observera även att skrivarbussen behåller ett giltigt värde tills DAV nollställs och först efter detta blir skrivarbussen odefinierad.

I vår nästa programlösning däremot så innehåller skrivarbussen dataregistrets värde tills skrivaren signalerar READY och mikrodatorsystemet skriver in ett nytt tecken i dataregistret.

Ett nytt program visas nedan. Vi visar först de uppdaterade printerdefinitionerna enligt:




```
* Definitioner för MC68
PRINTER     EQU   $80000
PSTATUS     EQU   $80001
PCONTROL    EQU   $80002
* Definitioner för MD68k
PRINTER     EQU   $400001
PSTATUS     EQU   $400003
PCONTROL    EQU   $400005
EOT         EQU   4

      ORG   $4000
      MOVEA.L     #Text,A0
Loop
      MOVE.B      PSTATUS,D0
      BTST.B      #2,D0
      BEQ         Loop

      MOVE.B      (A0)+,D0
      CMPI.B      #EOT,D0
      BEQ         Stop
      MOVE.B      D0,(PRINTER).L
      MOVE.B      #2,(PCONTROL).L

Loop2
      MOVE.B      PSTATUS,D0
      BTST.B      #2,D0
      BNE         Loop2
      CLR.B       (PCONTROL).L
      BRA         Loop

Stop  NOP
      BRA         Stop

Text  DC    "Hej Du Kalle!"
      DC.B  EOT
```

UPPGIFT 2.11

Redigera en källtextfil PRINTERV3.S68 enligt förslaget ovan. Assemblera och rätta eventuella fel.

Stega dig genom programmet till det inväntar READY-signalen och studera sedan:

- att READY är ettställd i statusregistret (Tri state buffer).
- Stega dig vidare genom programmet tills DAV ettställs. Detta visas nu på I/O-simulatorns styrregister. Studera även hur READY nollställs.
- Stega dig vidare genom programmet och verifiera att DAV nollställs i styrregistret.
- Stega vidare tills programmet inväntar att READY skall ettställas.

Med detta har du verifierat att programmet fungerar korrekt. Välj slutligen att testa programmet med 'Run' respektive 'Run Fast' och kontrollera att alla tecken skrivs ut.

Slut på Uppgift 2.11.

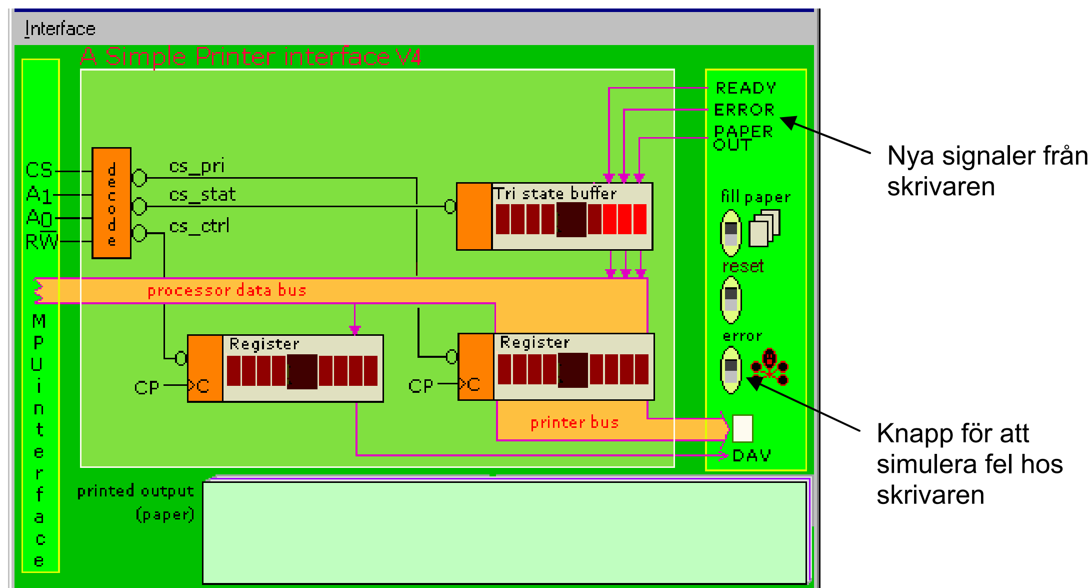
Vi har nu konstruerat ett hårdvarugränssnitt för en mycket enkel skrivaranslutning. Vi har också konstruerat och testat en drivrutin (programvara) för detta gränssnitt. Vi har en konstruktion som skriver ut en hel textsträng felfritt utan att varken missa tecken eller skriva ut dubbletter.

Printergränssnitt V4

De flesta skrivare har någon form för signalering för slut på papper eller att det uppstått ett fel i skrivaren så att man handgripligen måste åtgärda problemet. Vi studerar därför nästa variant av vår skrivarport.

UPPGIFT 2.12

Välj nu 'Interface 4'. Bilden är utökad med en signal som anger slut på pappret och en signal som anger fel i skrivaren. Vidare finns en knapp där vi kan simulera fel hos skrivaren.



Simulatorbild över printer gränssnitt V4

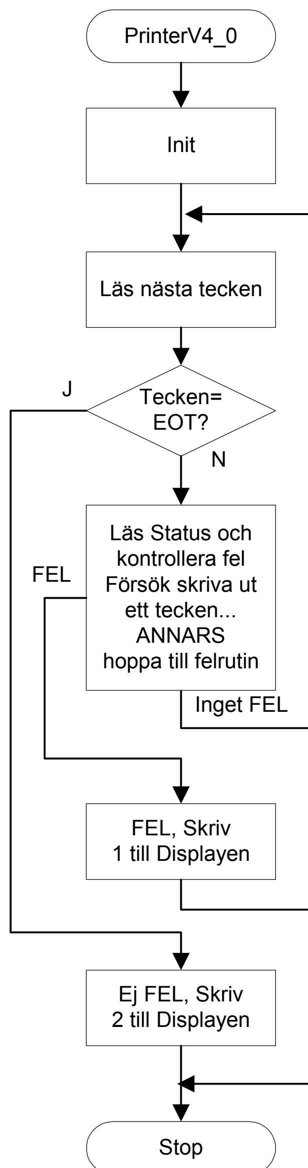
PAPER OUT är ansluten till statusregistrets bit 0 och signalen ERROR till bit 1.

Testa felsignalen genom att klicka på error-knappen och observera att felsignalen nollställs.

Slut på Uppgift 2.12.

Vi måste nu definiera om 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.



UPPGIFT 2.13

Rita en bild av programmerarens bild över det nya statusregistret och välj lämpliga namn för de nya statusbitarna.

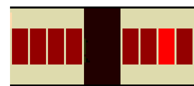
Programmerarens bild över det utökade statusregistret

Slut på Uppgift 2.13.

UPPGIFT 2.14

Du skall nu utöka ditt program så att det tar hänsyn till att något annat fel uppstår hos skrivaren. Du behöver här inte ta hänsyn till pappret (det kommer i nästa uppgift). Redigera en källtextfil `PRINTERV4_0.S68` enligt flödesplanen i marginalen, du kan lämpligen utgå från `PRINTERV3.S68`, assemblera och spara filen.

Observera att hela flödesplanen inte är ritad, jämför med den tidigare flödesplanen PrinterV3.0. Du ska här också använda IO-simulatorn *'ML4 Parallel output'* för att indikera att ett fel upptäckts och därför avbrutit utskriften eller att utskriften är felfri. Efter att ha skrivit ut hela textsträngen ska programmet skriva en 2:a till lysdioderna.



Indikator för felfri utskrift

Slut på Uppgift 2.14.

UPPGIFT 2.15

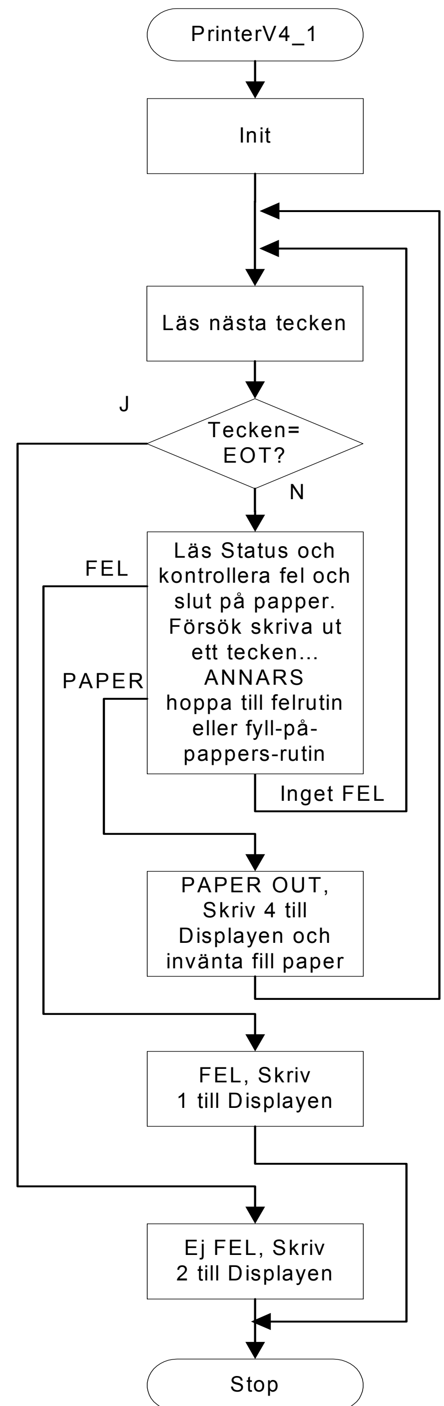
Utgå från din lösning i föregående uppgift och redigera en ny källtext `PRINTERV4_1.S68`. Du skall nu även ta hänsyn till att pappret tar slut så att utskriften fortsätter korrekt på nästa blad.

Studera flödesplanen i marginalen. Vid slut på papper skall utskriften upphöra och en 4:a skall skrivas till lysdioderna som en indikation på att pappret är slut. Programmet skall nu invänta att du klickar på fill paper i simulatorn innan det fortsätter.

Flödesplanen visar att vid slut på papper så läses tecknet in på nytt (boxen ”Läs nästa tecken). Detta är inte nödvändigt men ger en bra struktur på flödesplanen.

Ändra programmet, assemblera det och ladda det till simulatorn. När du testar din lösning för korrekt funktion undersök då noggrant att du inte missar några tecken vid "pappersbyte".

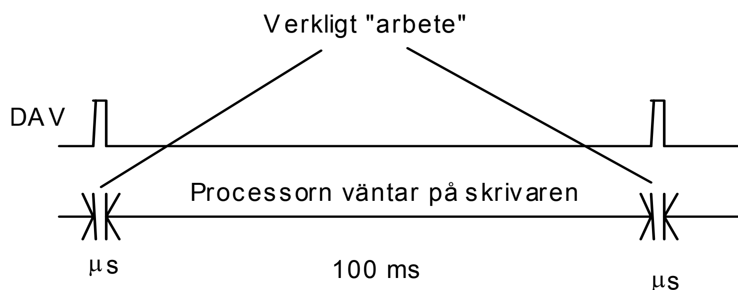
Slut på Uppgift 2.15.



Avslutande resonemang kring printerporten.

Vi har nu utvecklat programvara för att skriva ut en längre text, som löper över flera sidor. Konstruktionen fungerar, om det är intet tvivel, tecken skrivs endast till skrivarens dataregister då den är redo att ta emot ett nytt och skrivaren läser endast dataregistret efter att ett nytt tecken är gett.

En fråga vi bör ställa oss nu är hur effektivt processorn utnyttjas under den tid en utskrift pågår. Funderar vi lite över vilka instruktioner som utförs i de givna programförslagen, och hur lång tid de olika program-slingorna tar att exekvera jämfört med den tid det tar att skriva ut ett tecken så kommer vi fram till att datorsystemet *väntar* i 99.99% av tiden.



Verkligt arbete utfört av processorn.

Ur vår synvinkel utför processorn "nyttigt arbete" endast när den skickar ett tecken till skrivaren men inte under den tid den inväntar att skrivaren skall bli redo att ta emot nästa tecken.

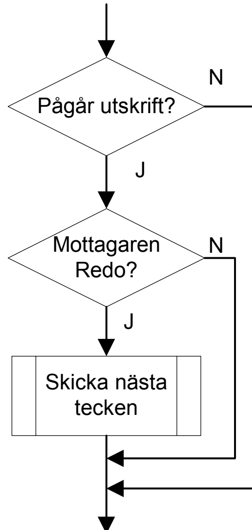
Vi har följaktligen konstruerat ett system som är hopplöst ineffektivt. Processorn är mesta tiden upptagen med att undersöka

om skrivaren kan ta emot ett nytt tecken. Om skrivaren klarar att skriva ut 4 tecken per sekund undersöker processorn READY signalen mer än 10 000 gånger per utskrivet tecken. Detta beror på att vi valt 'Busy Wait'- konstruktionen vilken kan vara praktiskt användbar i en del tillämpningar men knappast till vår printer port där vi har *fullständig handskakning* med DAV och READY signalerna.

Betrakta följande huvudprogram som är tänkt att fungera med denna konstruktion (busy wait).

```
* HUVUDPROGRAM
       ...            Olika initieringar
       ...

Loop   ...
       ...            Annat arbete som
       ...            processorn utför
       ...
       JSR   Print    Skriv ut en fil
       ...
       ...            Ännu mera jobb som
       ...            processorn utför
       BRA   Loop
```

Vi kan förmoda att det finns någon form av programslinga i huvudprogrammet som exekveras oupphörligt. Annat arbete som processorn utför kan exempelvis vara att läsa en temperaturgivare, beräkna vilken temperatur värdet motsvarar, påverka en värmekälla och att skriva temperaturen på en display mm. Skulle vi nu hamna i en lång utskrift medför detta att det blir ett "avbrott" i temperaturmätningen som i vissa sammanhang inte kan tolereras.

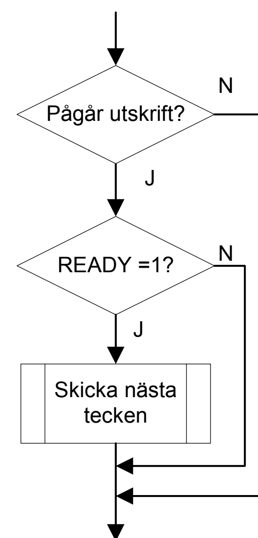


I stället för att i utskriftsrutinen `(Print)` undersöka om skrivaren är redo (BUSY) att ta emot ett nytt tecken kan vi tänka oss att placera denna test i huvudprogrammet. Rutinen som skriver ut en textfil minskas då till att skriva ut *endast ett tecken*. Se nedanstående huvudprogram och subrutin. Programmet är utökat med en flagga (`PFlag`) som indikerar om det överhuvudtaget pågår någon utskrift. Om så ej är fallet fortsätter programmet *utan* att testa READY signalen då utskrift ej pågår.

```
* HUVUDPROGRAM
       ...                Olika initieringar

Loop   ...
       ...                Annat arbete som
       ...                  processorn utför

       TST.B  PFlag       Pågår utskriftsrutin?
       BEQ    NoPrint     .. hoppa om NEJ

       MOVE.B PStatus,D0  Läs Status
       BTST.B #Ready,D0   . och hoppa vidare
       BEQ    NoPrint     .. om upptagen

       JSR    Print1char  Skriv ut ett tecken
NoPrint
       ...
       BRA    Loop
Pflag  DS.B   1           Flagga:Pågår utskrift?
```

Programkonstruktionen vi valt här kallas 'Polling' ("rundfrågning) som går ut på att *ibland* undersöka om en yttre enhet önskar någon form av service. I huvudprogrammet testar vi (exempelvis) en gång varje varv i programslingan om skrivaren är redo att ta emot ett nytt tecken (om någon utskrift pågår).

Allmänt, tar polling mycket processorkraft. Tänk dig ett antal terminaler som är anslutna till *ett* datorsystem på ett företag. Tänk dig vidare *hur ofta* de anställda använder tangentbordet på sin terminal. Om detta datorsystem använder sig av polling för att periodiskt undersöka om någon av användarna trycker på tangentbordet åtgår åtskillig processorkraft för detta. Processorn som arbetar efter polling- principen frågar väldigt ofta i onödan om de olika tangentborden är aktiverade. Polling principen skall *inte* förkastas då den är lätt att förstå, enkel att implementera och därför mycket använd, men kan inte användas urskiljningslöst som exemplet visade.

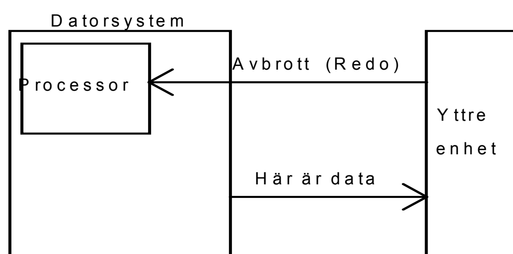


Studera blockdiagrammet och flödesplanen i marginalen. Vi har här hårdvarusignalen READY som är kopplat till printerportens statusregister. Vidare har vi kod, instruktioner, som undersöker om utskrift pågår och om READY är ettställd. Direkt efter instruktionen som utför test av READY-flaggan väljer vi en instruktion som utför ett subrutinanrop till subrutinen `Print1char`.

Det är alltså mjukvaran, programmet, som bestämmer huruvida subrutinanropet skall utföras eller ej.

Tänk dig nu om hårdvaran kunde bestämma när vi skall utföra hoppet till `Print1char`. På så sätt kunde processorn göra en massa annat nyttigt arbete utan att behöva läsa statusregistret. I detta fallet skulle vi kunna avlägsna ett antal instruktioner i vårt program

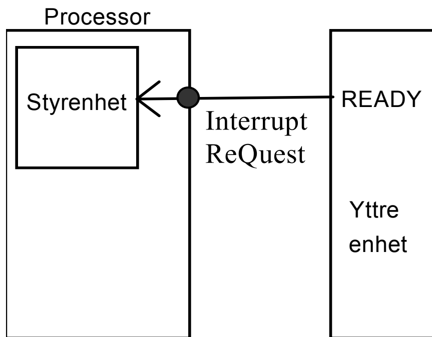
Det finns mekanismer i alla processorer där vi har möjlighet att ansluta exempelvis en READY signal direkt in till processorn för att uppnå ett hopp direkt till i vårt fall utskriftsrutinen `Print1char`. Denna mekanism kallas för *avbrott*.



I vårt fall ansluter vi vår READY-signal direkt till processorn och när READY aktiveras fås direkt ett hopp till utskriftsrutinen.

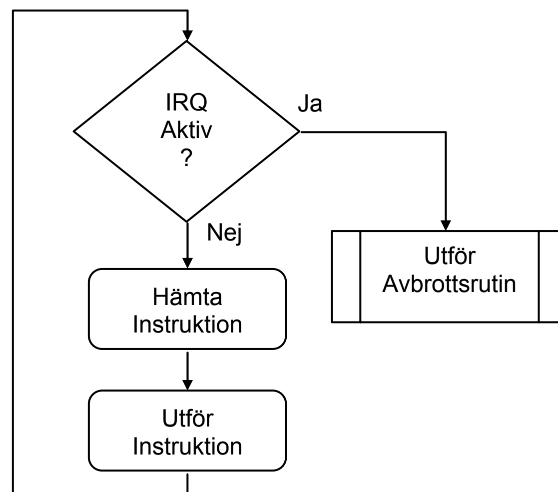
Introduktion till "avbrott"

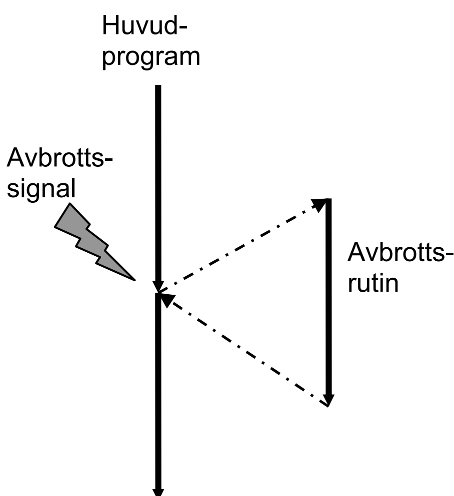
Vi skall nu övergå till att studera en metod där processorn avbryts i sitt normala arbete för att under en kortare tid utföra någon programrutin innan den därefter återupptar sitt normala arbete. Vi skall se mekanismer där en hårdvarusignal används som en signal till processorn som får den att omedelbart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



Låt oss anta att vi har en signal, exempelvis READY-signalen från skrivaren som vi tidigare beskrivit. Vi kopplar nu denna till en *avbrottsingång* 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:





Observera att avbrott aldrig kan inträffa under instruktionsexekvering, dvs. om processorn påbörjat hämtning av instruktion så kommer instruktionen också att utföras innan avbrottsingången kontrolleras nästa gång.

Man skulle kunna jämföra avbrottsrutinen med en subrutin som startas av en signal, snarare än kontrollerat av ett program (JSR), se figur i marginalen.

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.

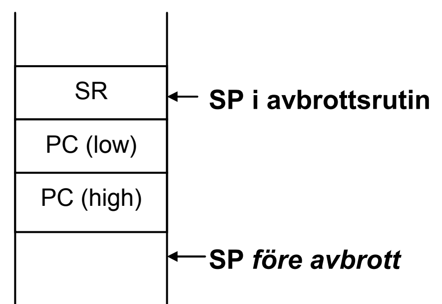
EXEMPEL:
MC68000 (MC68340) kan sägas ha 7 olika avbrottsingångar. Vi kallar dom IRQ1..IRQ7. Till varje avbrottsingång finns en adress kopplad (*autovector*) så att om exempelvis IRQ1 aktiveras så kommer adressen till avbrottsrutinen att läsas från adress 00000064 i minnet.

Eftersom ett avbrott kan inträffa i princip när som helst kan man heller inte i förväg veta vilken del av huvudprogrammet som exekveras då avbrottet uppträder. Detta medför att när återhoppet från avbrottsrutinen ska utföras så måste det finnas information om var huvudprogrammet blev avbrutet. Processorn sparar därför återhoppsadresen på stacken, dvs adressen till den instruktion som stod i tur att hämtas från minnet, precis som vid subrutinanrop (JSR).

Det räcker dock inte med att bara lagra återhoppsadressen. Ett avbrott kan komma när som helst, antag att det kommer mellan instruktionerna:

```
---
CMP.B       Variabel,D0
BHI         Larger
---
```

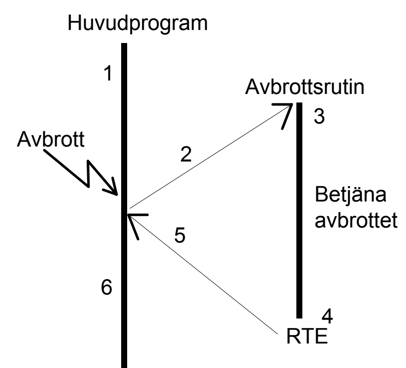
Då avbrottet inträffar precis innan den villkorliga instruktionen (BHI) så har flaggsättningen kunnat komma att ändras av avbrottsrutinen med ett felaktigt programflöde som följd. Vid avbrott sparas därför såväl programräknarens som statusregistrets innehåll på stacken.



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 RTE (*ReTurn from Exception*) för detta ändamål. RTE återställer *både* statusregister och PC från stacken.

Vi summerar händelseförloppet vid ett avbrott.

1. Processorn känner att IRQ är aktiverad.
2. Processorn sparar återhoppsadress och statusregistrets innehåll på stacken. Därefter läser processorn startadressen för avbrottsrutinen från den aktuella avbrottsvektorn, denna startadress placeras i PC
3. Avbrottsrutinen startas
4. Avbrottsrutinen avslutas med instruktionen RTE dvs återhoppsadress och statusregistrets innehåll återställs från stacken.
5. Återhopp till huvudprogram.
6. Därmed återstartas huvudprogrammet där det blev avbrutet.



En komplikation man måste tänka på då man skriver en avbrottsrutin är att innehållet i processorns register inte får ändras av avbrottsrutinen. Betrakta följande exempel som belyser problematiken

```
---
MOVE.B      Summa,D0
ADD.B       Belopp,D0
MOVE.B      Summa
---
```

Om avbrottet inträffar precis efter additionsinstruktionen:

```
ADD.B       Belopp,D0
```

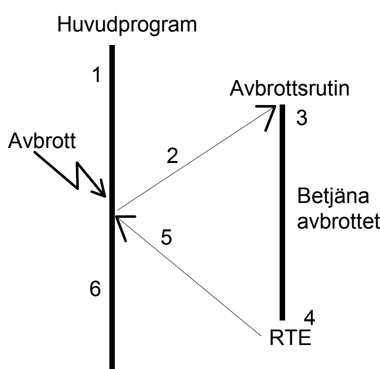
i programsekvensen och om avbrottsrutinen använder sig av register D0 kommer registerinnehållet att ändras. Detta medför att vår summa kommer att bli fel när denna skrivs sist i programexemplet. Alltså, för att undvika detta, och relaterade problem så måste de register som används i avbrottsrutinen först sparas. Omedelbart före utträde ur avbrottsrutinen återställs registerinnehållen så att det avbrutna programmet inte påverkats.

Vi övergår nu till att beskriva de handskakningsmekanismer som krävs mellan processor och yttre enhet vid avbrott. Den yttre enheten signalerar alltså med en avbrottsignal, IRQ, till processorn. Denna signal måste dock avlägsnas på något sätt under utförandet av avbrottsrutinen annars är den fortfarande aktiv när processorn återgår till huvudprogrammet. Detta skulle leda till upprepade avbrott från samma avbrottskälla.

På något sätt måste därför avbrottsrutinen *kvittera avbrottet* mot den yttre enheten så att denna kan inaktivera sin avbrottsbegäran. I bland sker detta automatisk då processorn läser från inporten (alternativt skriver till en utport), vilket den yttre enheten då uppfattar som en kvittens. En annan möjlighet är att processorn skriver i ett speciellt register i den yttre enheten för att kvittera avbrottet.

I processorns statusregister finns ofta flera statusbitar som har en viktig uppgift i avbrottssammanhang. Genom att manipulera dessa bitar kan programmeraren nämligen tillåta eller stänga ute ("maska bort") avbrott, s.k. *avbrottsmask*.

EXEMPEL
Hos MC68000 finns det tre bitar i statusregistret som utgör avbrottsmask. Masken kan alltså sättas till något av värdena 0-7. Om värdet sätts till 7 kommer inte processorn att utföra några andra avbrott än IRQ7. Om avbrottsmasken sätts till 0 kommer alla avbrott att betjänas.



Processorn ändrar automatiskt denna avbrottsmask vid avbrottsbetjäning. När den börjar utföra avbrottsrutinen (punkt 2 i marginalen) är fortfarande avbrottsbegäran aktiv från den yttre enheten. Alltså, vid punkten 2 *efter* det att processorn sparat registerinnehållen, inklusive statusregistret, så ändrar processorn sin avbrottsmask i statusregistret. På så sätt så stängs fortsatta avbrott från samma avbrottskälla ute.

Vid punkten 5 *efter* det att avbrottsrutinen har kvitterat avbrottsbegäran mot den yttre enheten, återställs registerinnehållen från stacken och på så sätt tillåts nytt avbrott.

Du ska nu utföra några enkla uppgifter med avbrott så att du blir bekant med funktionen och hur du bör använda simulatorn.

UPPGIFT 2.16

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

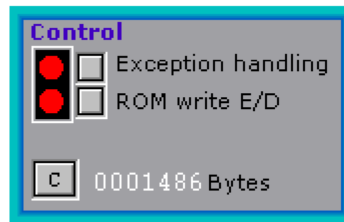
Efter initieringen utför programmet en slinga där vi ökar en variabel med ett för varje varv i slingan. Variabeln skrivs till en parallell utport så att vi ser förloppet på lysdioderna.

Avbrottsrutinen är utformad så att en annan variabel ökas med ett. Denna variabels värde skrivs till en annan parallell utport så att vi ser förloppet.

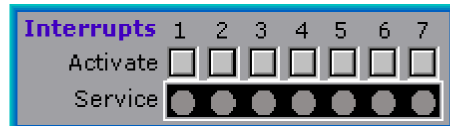
```
* irq1.s68
*

* MC68
PARPORT1    EQU   $80000
PARPORT2    EQU   $80001

* MD68K
PARPORT1    EQU   $400001
PARPORT2    EQU   $400003

      ORG   $4000
* Nollställ våra variabler
      CLR.B (Var1).L
      CLR.B (Var2).L
* Initiera avbrottsvektor IRQ1
      MOVE.L      #IrqR,($64).L
* Sätt om avbrottsmasken hos processorn
      ANDI  #$2000,SR

* I huvudprogrammet skrivs båda variabelvärdena
* till olika utportar.
* Endast 'Var1' ökas dock för varje varv i slingan
Loop
      MOVE.B      (Var1).L,(PARPORT1).L
      ADDI.B      #1,(Var1).L
      NOP
      MOVE.B      (Var2).L,(PARPORT2).L
      NOP
      BRA         Loop


* Avbrottsrutin
* Ökar 'Var2' med 1
IrqR
      ADDI.B      #1,(Var2).L
      RTE
```

Redigera en källtextfil `IRQ1.S68` enligt ovan, assemblera och rätta eventuella fel. Starta simulatorn.

Koppla PARPORT1 respektive PARPORT2 (adresserna beror på om du använder *MC68* eller *MD68k*) till två 'ML4 Parallel Output'.



Kontrollera att 'Exception handling' är aktiverad (ljust röd).



Här kan du aktivera olika avbrott. OBS Använd endast '1' för detta exempel eftersom programmet inte initierar för övriga avbrott. Då du klickar på knappen '1' genereras ett avbrott IRQ1 i simulator och motsvarande avbrottsrutin utförs. Notera speciellt att under avbrottsrutinen är dioden 'Service 1' tänd.

Välj först 'Run' och observera hur den första parallella utporten räknas upp.

Klicka på knappen '1' och observera beteendet.

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

Högerklicka med musen och prova att sätta en brytpunkt på avbrottsrutinens första instruktion. Välj 'Enable break on IRQ'. När du nu simulerar nästa avbrott kommer programexekveringen att stoppa inför den första instruktionen i avbrottsrutinen.

Slut på Uppgift 2.16.

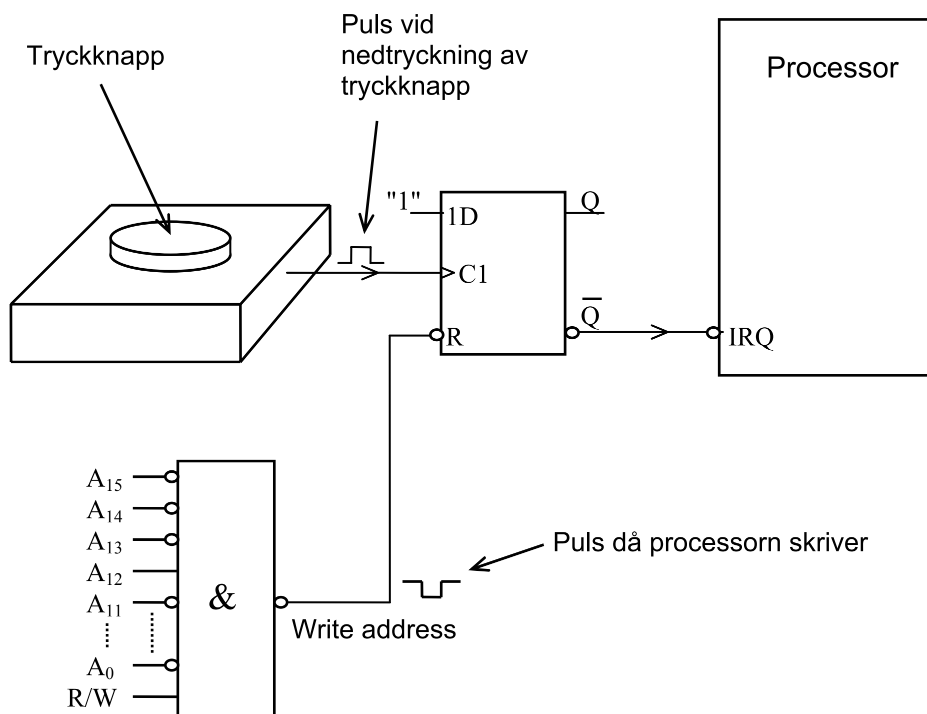
I den första uppgiften använde vi simulatorns inbyggda funktion för att "generera" (simulera) avbrott. Vi går nu vidare och exemplifierar en yttre enhets signal helt enkelt genom att införa en enkel tryckknapp utanför processorn.

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 införa 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.

Principen för kopplingen visas i nästa figur.



Koppling för yttre avbrottssignal

När vi trycker på knappen klockas en etta in från D-ingången på vippan. Detta medför att vippan sätts och Q' blir därmed noll. Då Q' är ansluten till processorn avbrottsingång IRQ fär processorn på detta sätt en avbrottsbegäran.

I avbrottsrutinen måste nu avbrottsbegäran in till processorn avlägsnas. Vi måste därför se till att ge vippan en RESET-signal för att återställa denna så att Q' blir ett. Till hjälp här har vi den stora NAND-grinden i figuren. I princip är detta adressavkodningslogik och grindens utgång aktiveras när och endast när 'Write address' uppträder på adressbussen och processorn samtidigt gör en skrivning.

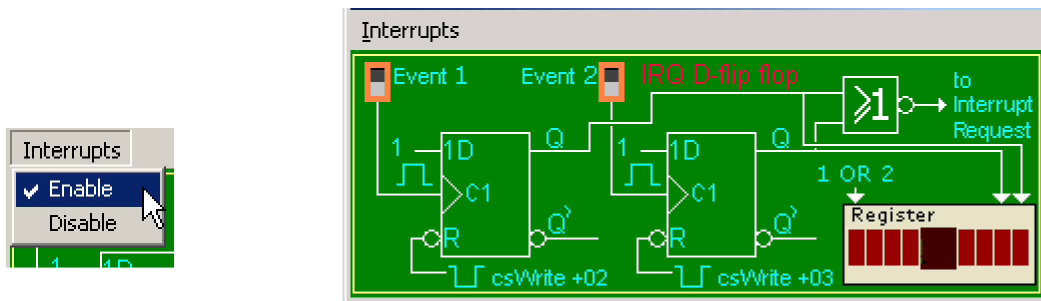
UPPGIFT 2.17

Kopiera filen `IRQ1.S68` till `IRQ2.S68` och arbeta vidare med denna fil. Lägg till följande portdefinitioner:

```
* MC68
IrqStat    EQU  $89C00     Status IRQ-sim
IrqRes1    EQU  $89C02     Nollställ event 1
IrqRes2    EQU  $89C03     Nollställ event 2

* MD68k
IrqStat    EQU  $413801    Status IRQ-sim
IrqRes1    EQU  $413805    Nollställ event 1
IrqRes2    EQU  $413807    Nollställ event 2
```

Anslut nu IO-simulatorn 'IRQ Flip Flop' och välj dess basadress till
`89C00` om du använder *MC68*
`413801` om du använder *MD68k*

Studera figuren. Klicka INTE på några knappar ännu. Denna simulator har två avbrottsvippor med gemensam anslutning via NOR-grinden till processorns avbrottsingång. Vidare kan vippornas Q-utgångar läsas via bit 0 och 1 i ett statusregister. Detta visas på lysdioderna längst ner till höger i simulatorn.



Simulator för avbrottskällor

Kontrollera att avbrott aktiverats hos simulatorn

Tidigare simulerade du avbrott genom att klicka på en knapp i processorn simulator. Du skall här utnyttja knappen 'Event 1' i figuren ovan för att simulera avbrott. Först måste du dock ändra i ditt program `Irq2`.

I initieringen av programmet skall avbrottsvippan för Event 1 nollställas. Detta är för att förhindra oönskade avbrott. Du nollställer vippan genom att skriva på adressen basadress+2. Instruktionen

```
CLR.B (IrqRes1).L     Nollställ vippa 1
```

löser detta. (Gör samma initiering även för 'Event 2' då vi behöver detta längre fram). Lägg slutligen till en nollställning av avbrottsvippa 1 i avbrottsrutinen.

Assemblera och testa programmet. Starta programmet med 'Run' kontrollera att lysdioderna för parallellport 1 ändras som tidigare.

Klicka nu på Knappen 'Event 1' för att simulera avbrott. Du ska nu ha samma händelseförlopp som du hade för programmet `Irq1`.

Sätt en brytpunkt i avbrottsrutinen. Detta gör du enklast genom att välja 'Enable break on IRQ' när du högerklickar. Starta om simulatorn i läge 'Run'. Simulera sedan ett avbrott 'Event 1'. Studera bitarna i statusregistret hos IO-simulatorn 'IRQ Flip Flop'. Bit 7 är en gemensam bit som anger att antingen 'Event 1' eller 'Event 2' har inträffat. Bit 0 och bit 1 anger att 'Event 1' respektive 'Event 2' inträffat.

Stega nu instruktionsvis tills vippan nollställs.

Slut på Uppgift 2.17.

UPPGIFT 2.18

Använd samma testprogram som ovan och starta med 'Run'. Välj att simulera några 'Event 1'. Klicka därefter på 'Event 2'. Beskriv nu vad som händer och förklara varför ingen av utportarna längre ändras.

____

____

____

Lägg nu till nödvändiga instruktioner i avbrottsrutinen så att du får en uppräkning av utport 1 oberoende om du klickar på 'Event 1' eller 'Event 2'.

Redigera, testa och rätta eventuella fel. Spara filen under namnet `IRQ3.S68`.

Slut på Uppgift 2.18.

UPPGIFT 2.19

Jobba vidare med programmet, men nu med filnamnet `IRQ4.S68`. Det skall fungera som tidigare fast klickar du på 'Event 2' skall parallellport 2 nollställas. Klickar du däremot på 'Event 1' skall parallellport 2 räknas upp med ett i vanlig ordning.

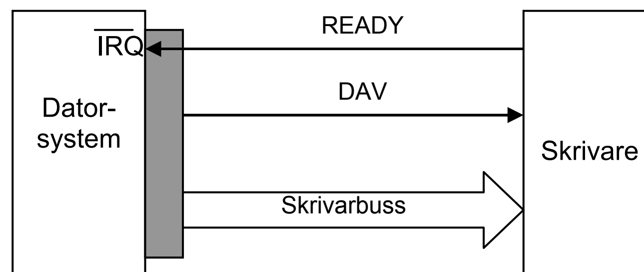
Du måste här utnyttja bitarna i statusregistret för att ta reda på huruvida det är Event 1 eller Event 2 som är orsaken till avbrottet.

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

Slut på Uppgift 2.19.

Avbrottsdriven Printerport

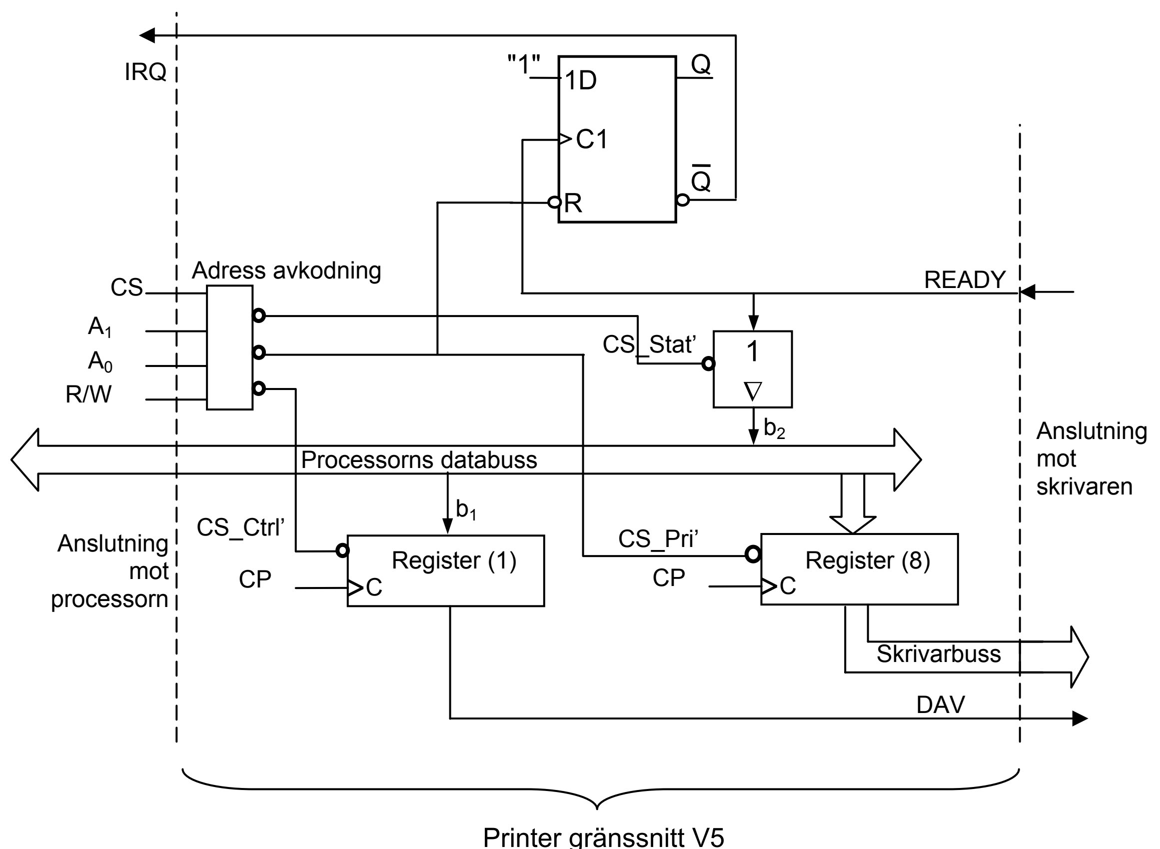
Vi ska nu återgå till vårt skrivarexempel och ansluta READY-signalen till processorns avbrottsingång.



Vi måste se till att avbrottssignalen kan avlägsnas när avbrottsrutinen utförs. För detta ändamål kan vi även här utnyttja en "vippa".

Printergränssnitt V5

Följande figur illustrerar hur printerporten nu kan generera *avbrott*.



Printer gränssnitt V5

När READY går hög kommer D-vippan att klocka in en etta vilket medför att Q' går låg. Q' är direkt ansluten till processorns IRQ-ingång. På så sätt styr skrivarens READY-signal när avbrott skall ske.

För att avlägsna avbrottssignalen till processorn används CS_pri'-signalen för att påverka RESET-ingången på vippan. Detta medför att när avbrottsrutinen skriver ett nytt tecken till skrivaren kommer

vippan att nollställas och därmed ett ställs Q' och IRQ in till processorn.

Vi förutsätter att skrivaren nu utnyttjar flanktriggning av DAV-signalen. När den känner av en positiv flank på DAV innebär detta att skrivarbussen har ett giltigt värde. Utöver detta skall skrivarbussen behålla värdet tills skrivaren på nytt signalerar READY. Först efter detta får processorn på nytt skriva till skrivaren dataregister. Vi betraktar nu DAV-signalen som en *strobe*

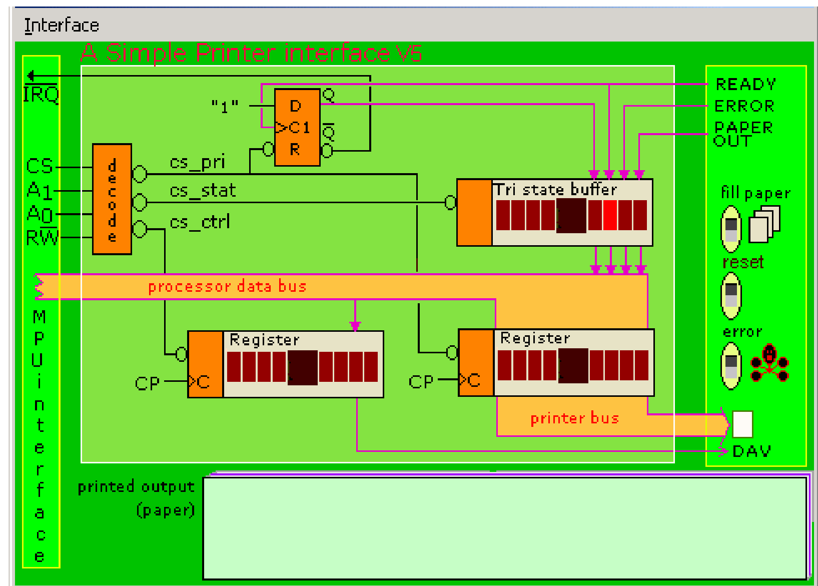
Låt oss se hur en avbrottsdriven tillämpning för detta skrivargränssnitt kan se ut. I applikationen använder vi även diodrampen 'ML4 Parallel Output' - vi låter huvudprogrammet ändra utdata till diodrampen och vi låter en avbrottsrutin ta hand om utskriften till skrivaren.

```
* Avbrottsrutin Printer V5.0
      ORG   $4000
start
*     Initiering
      CLR.B       dummy       räknarvariabel
      CLR.B       TextP       aktuellt tecken

*     Initiera för avbrott ...
      MOVE.L      #IrqRut,($64).L   Avbrottsvektor
      ANDI        #$2000,SR   acceptera avbrott

*     HUVUDPROGRAM
*     Simulera 'nyttigt arbete'
Loop
      MOVE.B      (dummy).L,(ML4_OUT).L
      ADDI.B      #1,(dummy).L
      NOP
      NOP
      BRA         Loop

* Avbrottsrutin
IrqRut
      MOVEA.L     #Text,A0
      CLR.L       D0
      MOVE.B      TextP,D0
      MOVE.B      (A0,D0),D1
      CMPI.B      #EOT,D1
      BEQ         Stop
      ADDI.B      #1,D0
      MOVE.B      D0,TextP

      MOVE.B      D1,(PRINTER).L
      MOVE.B      #2,(PCONTROL).L   sättDAV
      CLR.B       (PCONTROL).L      reset DAV

Stop  RTE

Text  DC    "Hej Du Kalle!"
      DC.B  EOT

TextP DS.B  1
dummy DS.B  1
```

Observera hur avbrottsbegäran in till processorn nollställs samtidigt som `MOVE.B D1,(PRINTER).L` utförs. Detta medför att RESET-ingången på D-vippan aktiveras och IRQ blir ett ställd.

UPPGIFT 2.20

Starta först simulatorn för skrivaren och välj 'Interface 5' och jämför med följande figur.



Simulator för printerinterface V5

Figuren illustrerar hur avbrottsvippan nollställs genom en skrivning till printerportens dataregister.

Redigera nu ett program enligt det givna exemplet ovan. Du behöver inte ta hänsyn till att pappret tar slut eller att fel kan uppstå i skrivaren i detta moment.

Huvudprogrammets slinga utgör en inkrementering av värdet på parallellporten så att vi ser att det hela tiden händer något. Anslut därför lysdioderna.

Assemblera, testa och rätta fel tills du får önskad funktion. Spara programmet (`PRINTERV5_0.S68`).

Slut på Uppgift 2.20.

Tänk dig att du använt detta system för att skriva ut ett dokument. Vilket tillstånd är systemet i när hela dokumentet är utskrivet?. Skrivarens READY-signal är hög och avbrottsmasken i SR-registret är nollställd och processorn utför huvudprogrammet.

UPPGIFT 2.21

Prova ovanstående genom att starta en utskrift och invänta att den är slut. Klicka därefter på error-knappen på skrivaren. Observera att READY nollställs. Klicka slutligen på reset-knappen på skrivaren

och observera förloppet. Vad händer? ________________________

____

och varför? ____

____

____

Slut på Uppgift 2.21.

Vi har nyss simulerat hur någon stänger av skrivaren av någon orsak och sätter på den på nytt. När skrivaren är avstängd är READY-signalen nollställd. När nu skrivaren på nytt blir påslagen så signalerar den att den är redo att ta emot ett tecken. READY ettställs vilket medför att D-vippan klockas och vi får ett avbrott från skrivaren vilket *inte* är önskvärt.

Problemet kan lösas genom att manipulera avbrottsmasken. Detta sker då lämpligast i avbrottsrutinen direkt efter det att slutmarkeringen (EOT) läses från textsträngen i minnet i vår befintliga avbrottsrutin:

```
        .....
      CMPI.B      #EOT,D1
      BEQ         Stop
      ADDI.B      #1,D0
      MOVE.B      D0,TextP

      MOVE.B      D1,(PRINTER).L
      MOVE.B      #2,(PCONTROL).L   sättDAV

      CLR.B       (PCONTROL).L      reset DAV
      BRA         Exit

* Sätt högsta avbrottsmask hos sparat SR på
stacken...
Stop  ORI.W       #$700,(A7)

Exit  RTE
        ...
```

När processorn nu utför RTE kommer den att hämta ett SR-register där avbrottsmasken är ett ställd och därmed accepteras inte avbrott i fortsättningen.

UPPGIFT 2.22

Ändra i programmet enligt ovan och spara (`PRINTERV5_1.S68`). Testa sedan att klicka på error och reset på skrivaren.

Uppför sig systemet på ett bättre sätt nu? ______________________

Försök nu att besvara frågan: Skulle denna lösning fungera om vi haft två skrivaranslutningar och vi önskade skriva ut på båda skrivarna samtidigt?

____

____

____

____

Slut på Uppgift 2.22.

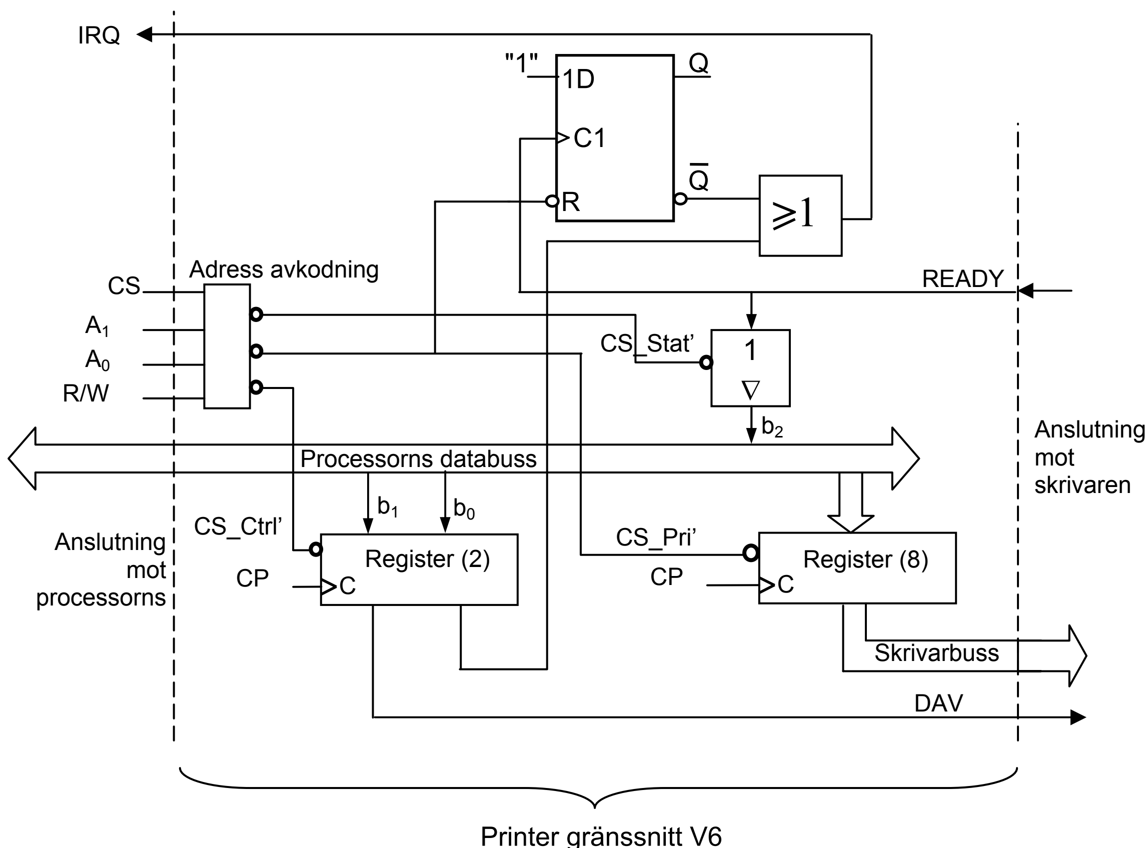
Lösningen ovan fungerar utmärkt så länge vi bara har en avbrottskälla. Vi kan då manipulera avbrottsmasken i SR-registret och utestänga vidare avbrott utan problem.

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.

Printergränssnitt V6

Gränssnitt 6 visar en bättre lösning i hårdvara. Notera styrregistret 'PCtrl' som nu utökats till 2 bitar. Den nya biten är anslutet till en OR-grind. Den andra ingången på OR-grinden är den tidigare IRQ-signalen. Om vi nu skriver en etta till b_0 på PCtrl-registret så kommer aldrig IRQ att gå låg. Detta är alltså ett sätt att programvarumässigt stänga av enhetens avbrottsmekanism.



Printer gränssnitt V6

Instruktionen

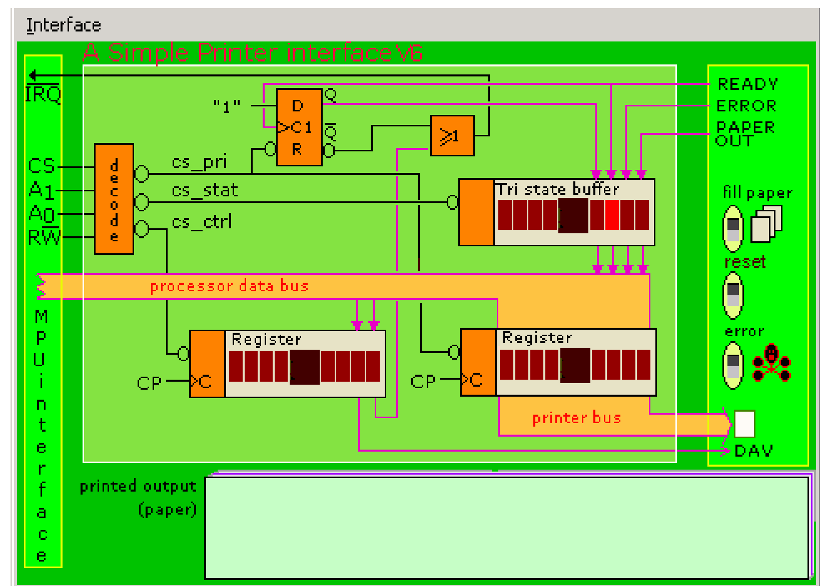
```
Stop  ORI.W       #$700,(A7)
```

vi provat tidigare kan nu bytas ut mot

```
Stop  ORI.B       #$1,PCtrl         ettställ bit 0
```

UPPGIFT 2.23

Välj nu 'Interface 6' för skrivaren och studera detta. Observera OR-grinden som tillkommit. Vi definierar den nya styrsignalen (bit 0) som *IrqOff*. Värdet vid RESET hos denna bit sätts till 1, dvs "*Interrupt Disable*".



Simulator för printerinterface V6

Skapa en ny källtextfil `PRINTERV6_0.S68` utgående från din lösning av förra uppgiften. Ändra programmet enligt ovan så att när du skrivit sista tecknet i avbrottsrutinen så manipulerar du bit 0 i styrregistret. Observera att du måste ta hänsyn till *IrqOff* även vid initieringen av systemet.

Utför ändringarna och kontrollera funktionen.

Slut på Uppgift 2.23.

UPPGIFT 2.24

Använd strömbrytarna (Dip switch input) för att starta en utskrift. Välj bit 0=1 på strömbrytarna som indikator på att en hel fil skall skrivas ut eller inte.

Spara filen under namnet `PRINTERV6_1.S68`.

Slut på Uppgift 2.24.

UPPGIFT 2.25

Ändra programmet PrinterV6.1 så att du kan inte bara starta men även starta om utskriften med strömbrytaren. Så att när du skrivit texten en gång klickar du på nytt på strömbrytaren och erhåller en ny utskrift.

Redigera och testa under filnamnet `PRINTERV6_2.S68`.

Slut på Uppgift 2.25.

Avsnitt 2
Sammanfattning

Vi har nu konstruerat ett enkelt parallellt gränssnitt och ingående provat och utvärderat aspekter på hårdvara såväl som programvara.

Vi har introducerat avbrott och sett exempel på några enkla tillämpningar.

Avancerade gränssnitt med dessa och många fler möjligheter finns som färdiga komponenter och kan ofta, mycket enkelt, anslutas till en mikroprocessors bussystem. Samlingsnamnet på sådana komponenter är *periferikretsar*.

En modern *micro-controller* innehåller såväl mikroprocessor (CPU) som en rad olika typer av gränssnitt. Det är ingen principiell skillnad på dessa från vad vi här presenterat. Olikheten ligger i att man nu kan placera fler komponenter i en enda kapsel.

Avsnitt 3

Tangentbord och Display

Syften:

I detta avsnitt ställs du inför uppgiften att skapa en större tillämpning än i de tidigare avsnitten. Ett viktigt syfte här är att ge dig kännedom om hur programsystem kan, och bör delas in i delar som är alltigenom överskådliga (modularisering). Avsnittet beskriver därför en metod att och dela upp källtexten i olika källtextfiler men också hur du kan utnyttja villkorlig assemblering för att göra dina program generella och därmed också återanvändbara.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.

Målsättningar:

Du ska kunna redogöra för samtliga lösningar du presterat till uppgifter i detta avsnitt. Det betyder dock inte att du självständigt ska ha utarbetat dom, du får gärna samarbeta med någon kamrat och lösningarna får naturligtvis då vara lika (gemensamma) du måste dock vara beredd att svara på frågor beträffande dina lösningar.

Om du ska utföra laborationer kring detta avsnitt (MC68 eller MD68k)
(tala med kursansvarige och konsultera kurs-PM)
Du har då ett speciellt laborations-PM till denna laboration. Uppgifterna i detta avsnitt är förberedande för laborationen,du ska därför spara samtliga källtextfiler och se till att du har dom tillgängliga vid laborationstillfället. Du kan knappast räkna med att genomföra laborationen inom utsatt tid om du inte dessförinnan förberett dig genom att ha arbetat i genom detta avsnitt.

Inledning

I detta avsnitt kommer du att se exempel på ytterligare former av gränssnitt. Du kommer att se att de i grunden inte skiljer sig från det du studerade i avsnitt 2 (skrivargränssnittet) men att de samtidigt ändå är så olika att de faktiskt kräver ny och speciell hårdvara för att fungera som de ska. Gemensamt med det tidigare visade gränssnittet är att det finns någon form av IOSEL-signal Basadresserna inom denna IO-area kan i princip väljas godtyckligt när du arbetar med simulatorn. Om du förbereder dig för laborationer gör du dock klokt i att redan från början välja riktiga adresser. Dessa ges efter hand i texten men du finner också en sammanställning av alla laborationskorts basadresser i appendix A.

De kringenheter vi studerar i detta avsnitt kallas Tangentbord *ML2* och Displaymodul *ML3*. Till varje kringenhet finns det tre olika gränssnitt. Du kommer att konfronteras med två utav dem.

ML2/ML15 Funktion

ML2 och *ML15* finns som enhet i IO-simulatorn. Gränssnittet utgörs av två konsekutiva register, vi behöver dock endast använda det första, för denna uppgift. Börja med att ansluta enheten *ML2* till simulatorn:

Om du använder MC68
Anslut till adress $89C00
Om du använder MD68k
Anslut till adress $413801

Kontrollera att
'Interface' = *ML15*
'Interrupts' = Disable

All avkodning utförs av hårdvara. Resultatet av avkodningen kan alltid avläsas från ett register på den anslutna adressen. Bitarna i registret har följande funktion:

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

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

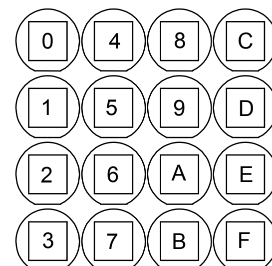
Bit 6-4, 0: Används ej

Bit 3-0, B3-B0: Tangentnummer; Anger aktuell (eller senaste) tangent.
En hexadecimal (0-F) anger aktuell (senaste) tangentnedtryckning.

UPPGIFT 3.1:
Skriv en enkel instruktionssekvens:

```
start:
      MOVE.B     (KEYBOARD).L,D0
      BRA        start
```

stega igenom sekvensen ett antal gånger. Prova med olika tangentnedtryckningar mellan varje gång. Komplettera också följande tabell:



Nedtryckt tangent	*Avläst värde (register)*
ingen tangent	
0	
7	
E	

SLUT PÅ UPPGIFT 3.1.

UPPGIFT 3.2:
Konstruera en subrutin, *CheckKbd*, enligt följande specifikation:

```
* Subrutin CheckKbd
* Läser tangentbord via ML15
* Returvärde i D0:
*  D0 = 0..F (hexadecimalt), tangentkod
*  D0 = $FF om ingen tangent är nedtryckt
```

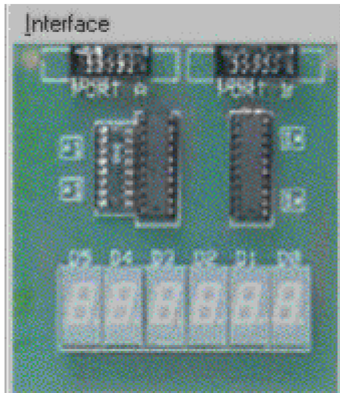
Kontrollera att den fungerar enligt specifikationen

SLUT PÅ UPPGIFT 3.2.

ML3/ML15 Funktion

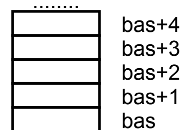
Det finns även möjlighet att ansluta display-modulen *ML3* till simulatorn. Observera dock att modulen upptar två adresser i minnet. Vi döper dessa register till 'MODE' respektive 'DATA'.

Anslutning av IOSIMULATORNS 'ML3 Display' till simulatorn



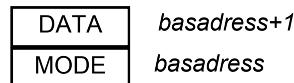
Om du använder MC68

MC68 har *byte*-organiserat minne, dvs systemet arbetar med en 8-bitars databuss.



Anslut *ML3* Keyboard till adress $89C02

MODE hamnar på basadress medan DATA hamnar på basadress+1.

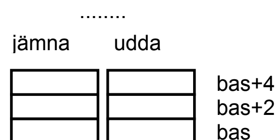


För att göra det hela tydligare kan du göra följande definitioner:

```
MODE EQU   $89C02
DATA EQU   $89C03
```

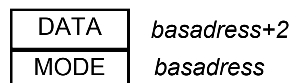
Om du använder MD68k

MD68k har *word*-organiserat minne, 8-bitars IO-enheter är anslutna till undre delen av databussen, därför kan endast UDDA adresser uppträda som IO-portar



Anslut *ML3* Keyboard till adress $413805

MODE hamnar på basadress medan DATA hamnar på basadress+2.



För att göra det hela tydligare kan du göra följande definitioner:

```
MODE EQU   $413805
DATA EQU   $413807
```

För att tända sjusifferindikatorerna krävs nu:

1) Att visningskretsen på *ML15* initieras
2) Att 8 bytes data skrivs till DATA-registret

först därefter tänds displayen.

Förfarandet kan beskrivas enligt följande:

Initieringssekvens:
```
1     ->    MODE
$90   ->    DATA
0     ->    MODE
```

kretsen är nu beredd att ta emot data...

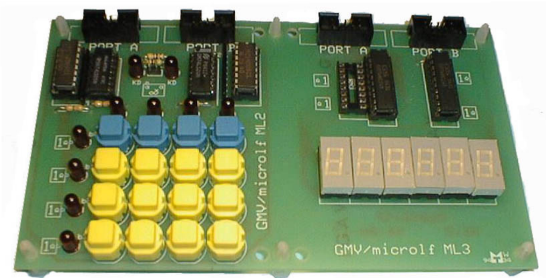
```
i = 0;
do(
     data[i] -> DATA;
     i = i+1;
)while( i < 8 )
```

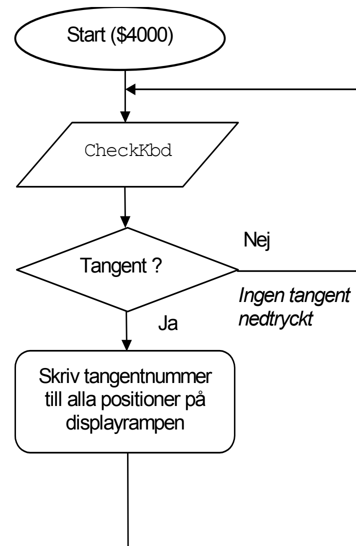
UPPGIFT 3.3:
Skriv en enkel instruktionssekvens som visar siffrorna
```
1 2 3 4 5 6
```
på sifferindikatorerna.

SLUT PÅ UPPGIFT 3.3.

UPPGIFT 3.4:

Konstruera ett litet program som läser av tangentbordet *ML2* (via *ML15*) och skriver den lästa tangentbordskoden till samtliga giltiga siffror (6 st.) på *ML3*. Om exempelvis tangent 3 trycks ned på *ML2* ska alla sifferindikatorerna på *ML3* visa siffran 3. Använd din subrutin CheckKbd från tidigare uppgift.





Programmet illustreras av flödesdiagrammet i marginalen:

SLUT PÅ UPPGIFT 3.4.

Villkorlig assemblering

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

Ett antal speciella direktiv används för att styra villkorlig assemblering. Samtliga direktiv inleds med tecknet '#'.

```
#define    SYMBOL
#ifdef     SYMBOL
#elif      SYMBOL
#endif
#else
```

Vi illustrerar användningen av dessa direktiv med följande exempel:

```
#define    MC68
...

#ifdef     MC68
* Om symbolen 'MC68' definierats tidigare..
MODE       EQU        $89C02
DATA       EQU        $89C03
#else
* i alla andra fall...
MODE       EQU        $413805
DATA       EQU        $413807
#endif
```

....

I detta fall kommer portadresser för *MC68* att användas vid den fortsatta assembleringen. Vill man däremot använda adresser för *MD68k* kan man antingen ta bort raden som definierar 'MC68' *eller* helt enkelt kommentera bort den enligt följande:

```
*#define   MC68
```

En annan möjlighet är att använda följande konstruktion:

```
#ifdef     MC68
* Om symbolen 'MC68' definierats tidigare..
MODE       EQU        $89C02
DATA       EQU        $89C03
#elif      MD68K
* Om symbolen 'MD68k' definierats tidigare..
MODE       EQU        $413805
DATA       EQU        $413807
#else
     Ingen portadress definierad
#endif
```

Om ingen av symbolerna 'MC68' eller 'MD68K' definierats med ett 'define'-direktiv kommer raden "Ingen portadress definierad" att assembleras och generera ett felmeddelande.

Du kan också använda `define`-direktivet för att definiera en konstant på samma sätt som med EQU..

Du kan alltså använda följande konstruktion:

```
#define    MODE  $89C02
```

i stället för :

```
MODE       EQU   $89C02
```

UPPGIFT 3.5:
Skapa en källtextfil `LABDEFS.S68` som använder villkorlig assemblering för att definiera portadresser för:

```
ML15_KeyBoardRegister
ML15_DisplayMode
ML15_DisplayData
```

Adresserna ska definieras för såväl *MC68* som *MD68k*. De aktuella adresserna finner du i appendix A.

SLUT PÅ UPPGIFT 3.5.

Resten av detta avsnitt ska vi ägna åt ett mindre programmerings-projekt. Vi kommer att använda tangentbordet och sifferindikatorn och vi kommer dessutom att introducera ett nytt gränssnitt, *ML5*. Gränssnittet är helt olikt det tidigare använda *ML15* och detta kommer att innebära avsevärda skillnader i programmet. Vi kommer därför att noggrannt beskriva detta nya gränssnitt.

Vi ska redan från början konstruera programmet på ett välstrukturerat och överskådligt sätt. Som vi ska se kommer detta att innebära att vi kan ’återanvända’ kod på ett sätt som inte alltid är möjligt om man inte tänker igenom sin programkonstruktion innan man börjar koda.

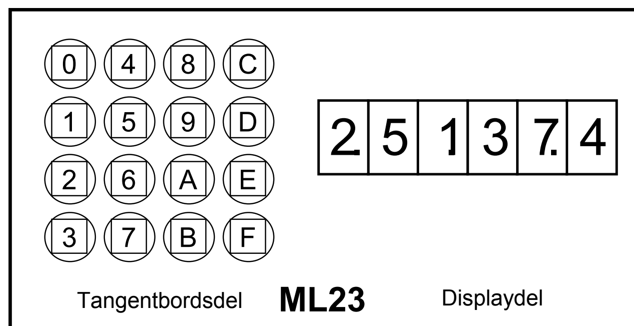
Programmeringsprojekt: Stoppur

Detta är det första i en serie av moment där vi konstruerar ett mikrodatorbaserat stoppur. Den ”hårdvara” vi har till förfogande består av: *laborationsdator*, gränssnitt *ML15* och tangentbord/display-kort *ML23*. Samtliga uppgifter utför vi dock här med hjälp av simulator och IO-simulator.

Vi är inte noga med ”verklig tid” i denna övning. Då vi säger ”stoppur” menar vi att konstruktionen ska fungera som ett stoppur. Vi kan dock inte kräva att vårt stoppur ska visa korrekt tid...

Anledningen till detta är att vi arbetar med simulatorer som inte kan ge verkliga realtidsegenskaper.

Laborationerna består i att konstruera, testa och demonstrera ett programpaket skrivet i MC68000 assemblerspråk. Programmet gör det möjligt att använda den beskrivna hårdvaran som ett elektroniskt stoppur. Tiden skall visas på display-delen och tangentbords-delen skall användas för att starta, stoppa och nollställa klockan. Vi använder alltså bara 3 av de 16 tangenterna. Följande illustration visar hur stoppuret har gått i 2 timmar 51 minuter 37 sekunder och 4 tiondelar.



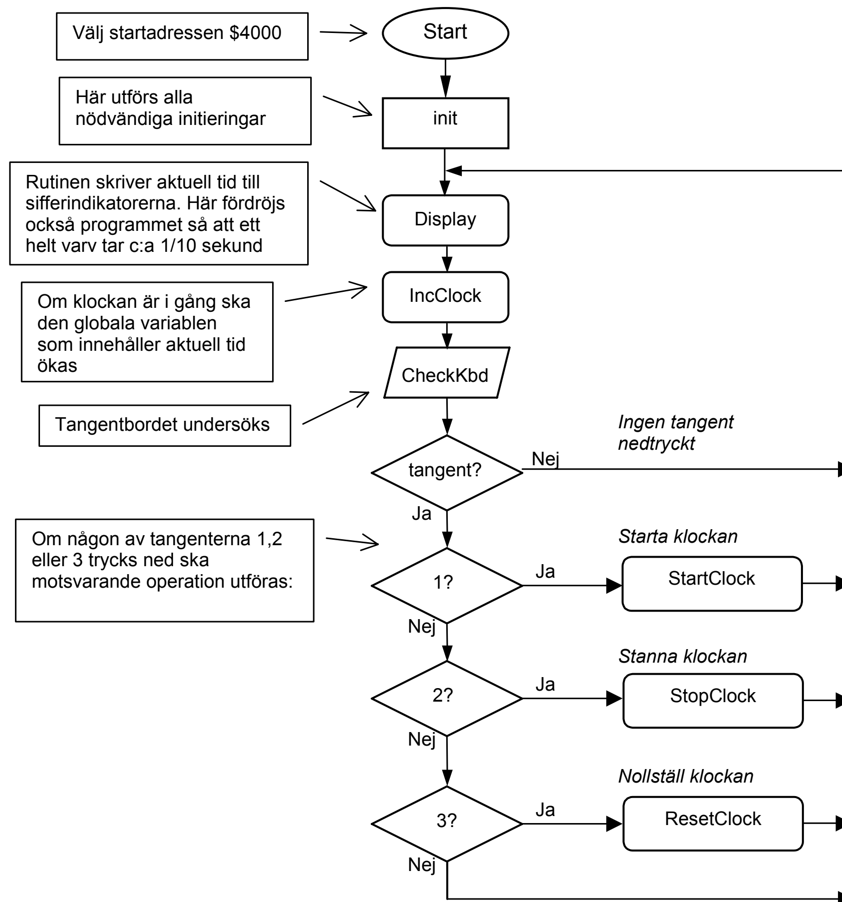
Med det färdiga programmet kan utrustningen mäta tid med en upplösning av tiondels sekunder. Tiden visas på en ramp (*ML3*) bestående av sex stycken så kallade 7-segmentsdisplayer (sifferindikatorer). Med ett 16-tangenters tangentbord (*ML2*), kan klockan startas (eller eventuellt återstartas), stoppas och nollställas. Såväl tangentbord som sifferindikatorer (*ML23*) ansluts till laborationsdatorn via gränssnittet *ML15*.

Programpaketet konstrueras i följande steg (moment):

- Anslutning av display-kort.
- Konstruktion av programvara för att visa siffror på display.
- Konstruktion av en “mjukvaru-klocka”
- Anslutning av tangentbord.
- Implementering av ”stoppur”.

Du måste strikt följa de specifikationer som ges och du ska också tillämpa den arbetsmetod som beskrivs.

För att ytterligare illustrera programpaketets funktion visas i figuren nedan en flödesplan av programmet. Laborationsmomenten kommer att bit för bit realisera programpaketet enligt denna beskrivning.



Vi lägger stor vikt vid en välstrukturerad lösning. Vi kommer därför att konstruera ett antal subrutiner för att implementera vårt stoppur, dessa antyds av flödesplanen ovan. Vi kommer dessutom att arbeta med tre olika källtextfiler:

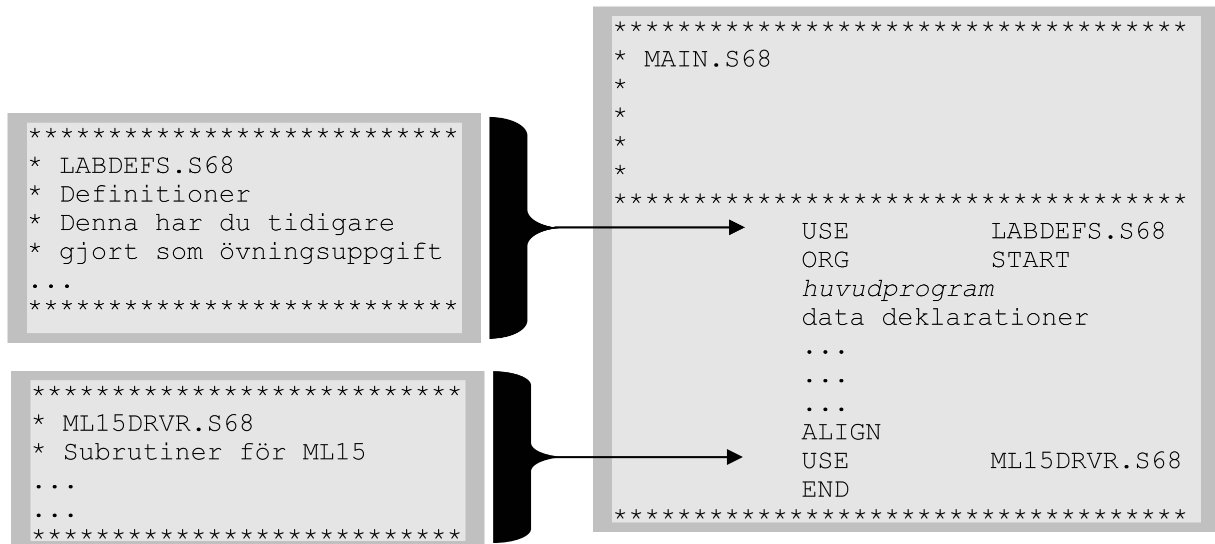
- `LABDEFS.S68`, som du redan börjat på, här samlar vi alla konstantdefinitioner (EQU-satser). Observera att varken programkod eller datadefinitioner får finnas i denna fil.
- `MAIN.S68`, här skriver vi huvudprogrammet, här deklareras också globala variabler (data definitioner).
- `ML15DRVR.S68`, här samlar vi de delar av programmet som är direkt kopplade till gränssnittet *ML15*.

Några delmoment har du utfört som tidigare uppgifter. Det kan därför underlätta att rekapitulera tidigare lösningar.

Huvudprogrammet

Huvudprogrammet skrivs i filen `MAIN.S68`, men vi samlar alla definitioner i filen `LABDEFS.S68` och vi samlar de funktioner som är direkt beroende av gränssnittet *ML15* i filen `ML15DRVR.S68` (vitsen med detta är att det blir enklare att byta ut *ML15* mot *ML5* i senare moment). För att inkludera filerna i `MAIN.S68` använder vi USE-direktivet:

```
      USE        <filnamn>
```



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 ökar överskådligheten och du ser bättre var, i minnet, de olika programdelarna hamnar.
- Endast definitioner (EQU-satser) får föregå det första ORG-direktivet.
- Om du deklarerar data på BYTE-form (direktiven `DC.B` och `DC`) i filen `MAIN.S68` måste du låta dessa deklarationer efterföljas av direktivet `ALIGN`. Annars kan deklarationerna resultera i att första instruktionen i `ML15DRVR.S68` hamnar på udda adress. Assemblatorn tillåter inte detta utan genererar då ett flemeddelande *("Bad Alignment...")*.

Tänk också på att det är filen `MAIN.S68` som ska assembleras. Du kan visserligen också assemblera de andra filerna men den resulterande laddfilen är förstås meningslös.

Vi börjar med att konstruera själva huvudprogrammet `MAIN.S68`. Utforma detta så att det huvudsakligen innehåller subrutinanrop. Det ska senare kompletteras med datadeklarationer. Observera att de subrutiner som inte ska placeras i `ML15DRVR.S68` (se nedan) ska placeras i `MAIN.S68`. Eftersom subrutinerna inte är konstruerade ännu utformar du dessa som "dummies".

EXEMPEL:
Om vi i huvudprogrammet har ett anrop av en ännu inte färdig subrutin *IncClock*:

```
      BSR   IncClock
```

definierar vi en dummyversion av subrutinen

```
IncClock:  RTS
```

för att så småningom fylla i koden för denna rutin.

UPPGIFT 3.6:
Skapa källtextfilen `ML15DRVR.S68`. Implementera en 'dummy-version' av subrutinen:

```
      Display
```

använd din tidigare version av

```
      CheckKbd
```

Observera att du, i denna fil, ska använda de symboliska adresser du definierat i `LABDEFS.S68.`

SLUT PÅ UPPGIFT 3.6.

UPPGIFT 3.7:
Konstruera ett huvudprogram enligt flödesplanen i början av detta moment. Skriv källtextfilen `MAIN.S68`. Använd DUMMY-versioner av alla subrutiner.
Assemblera och testa huvudprogrammet, kontrollera att det utförs enligt flödesplanen

SLUT PÅ UPPGIFT 3.7.

Specifikation av subrutinerna

Vi ska här ge specifikationer av alla de subrutiner som ska ingå i programpaketet. Vi ger också en del tips som är avsedda att förenkla arbetet.

Starta eller stanna klockan

Två rutiner ska konstrueras. `StartClock` och `StopClock`.
Tips: Inför en global variabel *Running* som sätts till 1 då klockan startas och som sätts till 0 då klockan stannas.

Öka eller nollställa klockan

Rutinen `IncClock` ska administrera den aktuella tiden. Om klockan går ska tiden alltså uppdateras 10 gånger i sekunden (vi är dock inte speciellt noga med att klockan går rätt). Vi tänker oss här att *IncClock* faktiskt anropas 10 gånger per sekund, därmed ska tiden uppdateras med en tiondels sekund vid varje anrop.
Tips: Deklarera en klockvariabel *clock*, 8 bytes (ty då kan denna skrivas direkt till display-rampen 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. Glöm inte att det går tio tiondelar på en sekund och 60 sekunder på en minut...

Adress	Data	Kommentar
clock	$02	Timmar
	$05	Tiominuter
	$01	Minuter
	$03	Tiosekunder
	$07	Sekunder
	$04	Tiondelar

Variabeln *clock*

Rutinen `ResetClock` ska återställa tiden till 0.
Tips: Nollställ klockvariabeln.

Tangentbordsrutin

Rutinen `CheckKbd` ska kontrollera om någon tangent är nedtryckt. I så fall ska tangentens kod (0-$F) returneras. Annars ska talet $FF returneras av rutinen. Du ska redan ha implementerat denna. *Använd register **D0** för returvärdet*.

Displayrutin

Rutinen *Display* skriver ut aktuell tid till displayrampen. I förra momentet löste du i princip uppgiften. Omforma denna lösning till en subrutin.

UPPGIFT 3.8:

Här ersätter du ”dummy”-rutinerna...

1. Börja med att färdigställa *Display*-rutinen. Ändra till några olika värden i *clock*-variabeln och övertyga dig om att visningen fungerar korrekt.
2. Implementera rutinen *IncClock* och testa programmet med en ”startad” klocka.
3. Implementera rutinerna *StartClock*, *StopClock ResetClock* .
4. Anropa *IncClock* en gång för varje varv i huvudprogrammet. På så sätt ser du att klockan uppdateras korrekt.

SLUT PÅ UPPGIFT 3.8.

Styrning av ML3 via ML5

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

Vi återanvänder programmet från föregående moment men ska här byta ut filen `ML15DRVR.S68` mot `ML5DRVR.S68`. Enklast gör du detta genom att helt enkelt spara `ML15DRVR.S68` under det nya filnamnet. I detta första moment ska vi byta ut Display-rutinen.

Förberedelser

Spara filen `ML15DRVR.S68` under namnet `ML5DRVR.S68`. Ändra också namnet i filen så du inte blandar ihop dessa längre fram. Ändra, i filen MAIN.S68 direktivet

```
      USE ML15DRVR.S68
```

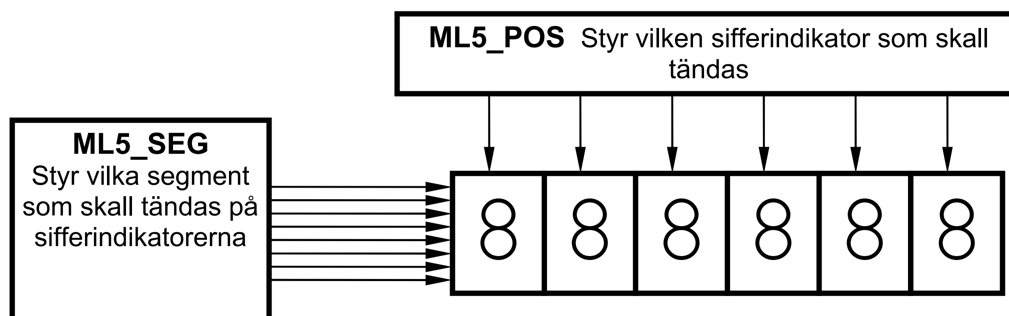
till

```
      USE ML5DRVR.S68
```

Tänk också på att du måste ändra inställningar i IO-simulatorn för korten *ML3* respektive *ML2*. I båda dessa fall ska du ändra 'Interface' från ML15 till ML5.

Subrutinen Display för ML5

Programvarumässigt utgörs nu gränssnittet av två 8-bitars ut-portar *ML5_SEG* som kopplats till PORT A på *ML3* respektive *ML5_POS* som kopplas till PORT B på *ML3*.



Du har använt en sifferindikator tidigare (avsnitt 1) på *ML4*-kortet. Sifferindikatorerna på *ML3* fungerar på samma sätt. Dock är segmentkoderna annorlunda.

Läs nu vidare om IO-simulatorns *ML2* och *ML3* i appendix C.

UPPGIFT 3.9:
Komplettera filen `LABDEFS.S68` med adressdefinitioner:
`ML5_SEG`
`ML5_POS`
Du finner dessa adresser i appendix A.
Som tidigare använder du villkorlig assemblering för att skilja adressdefinitionerna för *MC68* respektive *MD68k*.

SLUT PÅ UPPGIFT 3.9.

UPPGIFT 3.10:
Skapa nu en enkel testsekvens för att kontrollera att portadresser och inställningar är korrekta. Det kan exempelvis se ut som följande:

```
*

#define      XXXXX              ersätt med MC68 eller MD68K

 USE         labdefs.s68        inkludera portdefinitioner

 ORG         $4000
start:
 MOVE.B      #$FF,D1            mönster för att tända samtliga segment
 MOVE.B      #%11011111,D0      mest signifikanta position

repetera:
 MOVE.B      D0,(ML5_POS).L     tänd segment
 MOVE.B      D1,(ML5_SEG).L     segmentsmönster
 MOVE.B      #$FF,(ML5_POS).L   släck alla indikatorer
 ASR.B       #1,D0              nästa indikator
 BCS         repetera           tills samtliga pos. har visats

 MOVE.B      #$0,(ML5_POS).L    tänd alla indikatorer

 BRA         start
```

Om du använder MC68
Anslut *ML3 Display* till adress $8C002
Om du använder MD68k
Anslut *ML3 Display* till adress $418005

Sätt 'Interface' till ML5.

Stega igenom testsekvensen och kontrollera funktionen.

SLUT PÅ UPPGIFT 3.10.

Som du ser gäller det mönster du skriver till *ML5_SEG* för *alla* aktiva indikator.

Hur ska vi då kunna visa olika siffror på olika indikatorer med detta gränssnitt? Exempelvis 2.51.37.4 som motsvarar 2 timmar 51 minuter 37 sekunder och 4 tiondelar.

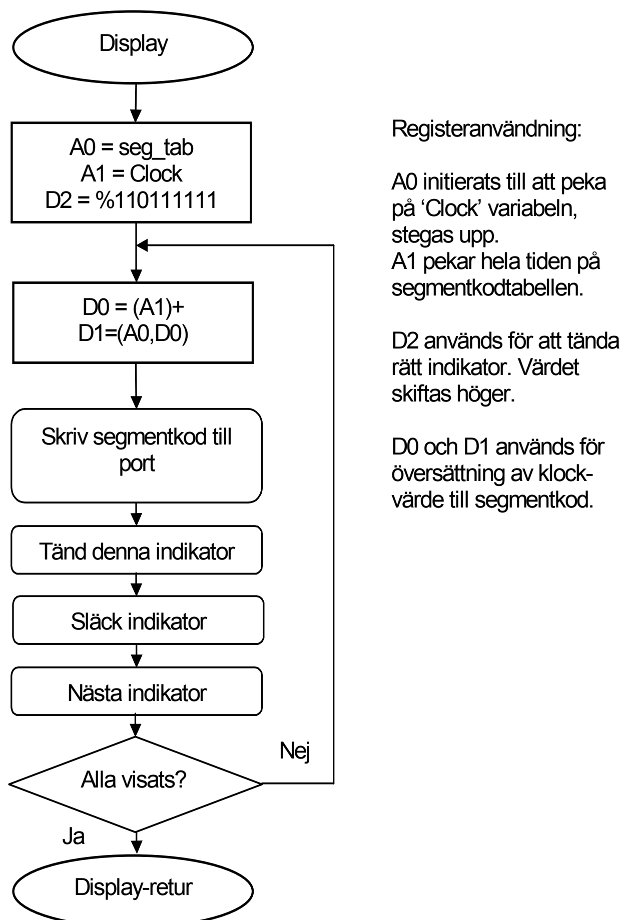
Let's make a movie...

Om vi skriver mönstret för att visa "tvåan" för timmar till *ML5_SEG* så kommer detta att vara aktivt för alla sifferindikatorer, vi bör därför endast visa en siffra åt gången.
Lösningen blir att göra "film" av det hela. Vi skriver ett program som:

- skriver ut mönstret för en 2:a på *ML5_SEG* och tänder endast den vänstra sjusegment indikatorn via ML5_POS. (“Tvåan” i timmar)
- sedan skriver ut mönstret för en femma på *ML5_SEG* och tänder nästa sjusegment indikatorn via *ML5_POS*. (“Femman” i 51 minuter)
- och sedan skriver ut mönstret för en 1:a på *ML5_SEG* och tänder den nästa sjusegment indikatorn via *ML5_POS*. (“Ettan” i 51 minuter)
- osv.

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 börjar vi om från början med den vänstra indikatorn och tänder upp en 2:a på nytt. I och med att detta förfarande upprepas i snabb takt gör vi en “film” av det, det ser ut som om alla indikatorerna lyser samtidigt med olika siffror. Det första som krävs är en programslinga som kan tända upp en sjusegment indikator i taget. Den vänstra indikatorn skall tändas först och den högra till sist. Observera att endast EN indikator i taget skall lysa. För att få till detta krävs att en “nolla” roterar på *ML5_POS*. Jämför med testsekvensen du gjorde tidigare.

Vi har tidigare (i avsnitt 1) sett hur man enkelt översätter decimala siffror till motsvarande segmentsmönster (som skall skrivas till *ML5_SEG*). Du hittar segmentkoder för *ML3* i appendix C. Det är nu dags att specificera *ML5*’s *Display*-rutin.



Registeranvändning:

A0 initierats till att peka på ‘Clock’ variabeln, stegas upp.
A1 pekar hela tiden på segmentkodtabellen.

D2 används för att tända rätt indikator. Värdet skiftas höger.

D0 och D1 används för översättning av klock-värde till segmentkod.

UPPGIFT 3.11:
Redigera nu filen ML5DRVR.S68. Implementera rutinen *Display* enligt anvisningarna. Testa din *Display*-rutin exempelvis enligt följande:

```
*
#define MD68K

     USE   labdefs.s68       portdefinitioner...

* Huvudprogram för att testa den första varianten av 'Display'

     ORG   $4000
main:
     BSR   Display
     BRA   main


* definiera ett godtyckligt klockvärde
Clock:   DC.B  1,2,3,4,5,6

     USE   ml5drvr.s68
```

Testa din lösning genom att stega igenom Display-rutinen.
Studera registerinnehållen för varje varv som utförs i program snurran: (Testa åtminstone två- tre varv)

- kontrollera att rätt klockvärde läses och ....
- att detta översätts till korrekt segmentkod och....

att när “nästa” sifferindikator tänds så visas korrekt decimalsiffra.

Då du övertygat dig om att rätt värde visas, prova med ’Run’ respektive ’Run Fast’.

SLUT PÅ UPPGIFT 3.11.

Kan du tydligt uppfatta varje siffra (vid ’Run Fast’) eller blinkar det bara?’. Det krävs förmodligen någon form av fördröjning av den tid som varje siffra visas. Denna fördröjning ska alltså utföras *en gång varje varv* i programslingan i *Display*.

UPPGIFT 3.12:
Komplettera rutinen *Display* med en instruktionssekvens som fördröjer exekveringen mellan upptändning av varje indikator.
NOT: Eftersom IO-simulatorn är betydligt långsammare än hårdvara kommer du inte att uppfatta alla indikatorer som tända samtidigt. Snarare kommer det att bli någon form av ”rinnande ljus”. Försök anpassa fördröjningen så att det tar c:a 0,25 sek att genomföra visning av alla sex siffror då du använder ’Run Fast’.

SLUT PÅ UPPGIFT 3.12.

UPPGIFT 3.13:
Kombinera Display för *ML5* med *CheckKbd* för *ML15* (kopiera kod från ML15DRVR.S68 till ML5DRVR.S68) och kontrollera att stoppuret fungerar även med *Display* för *ML5*.

SLUT PÅ UPPGIFT 3.13.

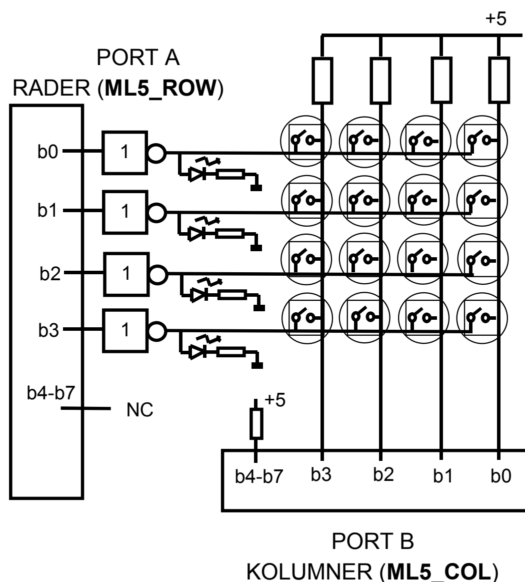
Anslutning av ML2 via ML5

Det är nu dags att ersätta tangentbordsrutinen *CheckKbd* för *ML15* med en tangentbordsrutin för *ML5*. Du fortsätter arbeta med samma källtextfiler som i föregående moment, dvs `LABDEFS.S68`, `MAIN.S68` och `ML5DRVR.S68`.

Under detta moment kommer vi att göra en programmerad avsökning av tangentbordet.

Tangentbordsrutin för ML5

Studera principskissen för kopplingen mot tangentbordet. När ingen tangent är nertryckt så läses endast ettor från *ML5_COL*. För att känna av en tangentnertryckning måste en rad (*ML5_ROW*) aktiveras. Som framgår av figuren är PORT A en skrivport för laborationsdatorn. PORT B är en läsport.



Tangentbordet är bestyckat med 16 tangenter. Dessa är uppsatta i fyra rader med fyra tangenter i varje rad.

Tangentbordets rader aktiveras via PORT A och alla kolumner kan läsas samtidigt via PORT B.

En HÖG nivå på ML5_ROW aktiverar en rad.

En LÅG nivå på ML5_COL anger att en tangent är nertryckt

Du kan läsa mer om IO-simulatorns enhet *ML2*/*ML5* i appendix C.

Om du använder MC68
 Anslut *ML2 Keyboard* till adress $8C000
Om du använder MD68k
 Anslut *ML2 Keyboard* till adress $418001

Sätt 'Interface' = ML5

UPPGIFT 3.14:
Skriv en instruktionssekvens som aktiverar den nedersta raden på tangentbordet, läser kolumnerna och sedan upprepar detta. Använd portdefinitioner från `LABDEFS.S68`.

SLUT PÅ UPPGIFT 3.14.

Låt tangenterna motsvaras av koder enligt följande figur:

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

För implementeringen av stoppuret använder vi denna gång tangenterna 3, 7 och B. Observera att detta val av tangenter innebär att endast en rad måste undersökas (aktiveras). De övriga raderna kan utelämnas tills vidare.

UPPGIFT 3.15:
Ange vilket värde som skall skrivas till *ML5_ROW* och vilka värden som läses från *ML5_COL* för att detektera respektive tangentnedtryckningarna 3, 7 och B?

Tangent	ML5_ROW	ML5_COL
3		
7		
B		

SLUT PÅ UPPGIFT 3.15.

UPPGIFT 3.16:
Färdigställ nu programpaketet där du använder såväl tangentbordet *ML2* som displayrampen *ML3* via *ML5*. Kontrollera att stoppuret fortfarande fungerar som det ska.

SLUT PÅ UPPGIFT 3.16.

Med detta har vi nu fullgjort uppgiften att implementera ett ”stoppur” med hjälp av *ML2*/*ML3*/*ML5* och *ML15*. Vi har dock fuskat en smula genom att välja enklast tänkbara tangentbordskodning.

I det avslutande momentet ska vi lösa uppgiften då en tangentbords-kodning tvingar oss att avsöka hela tangentbordet. Med detta får vi möjlighet att införa ytterligare funktionalitet, nämligen att sätta klockans startvärde från tangentbordet.

Generell tangentbordsavkodning

Vi fortsätter under detta moment med avkodning av *hela* tangentbordet. Därefter konstruerar vi en rutin för att kunna ställa klockan till ett bestämt startvärde.

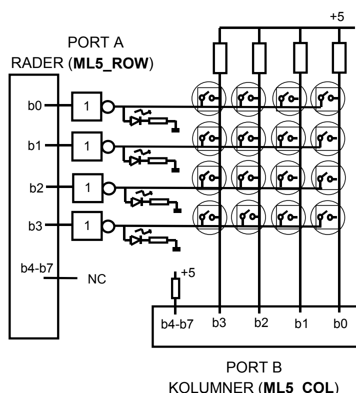
Svårighetsgraden i detta moment är något högre än i de förra momenten.

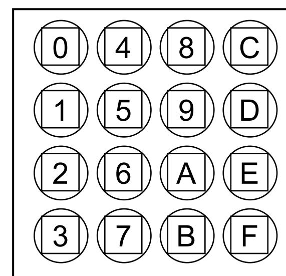
Du har nu större frihet att själv utforma lösningen på uppgiften.

Tekniken som används för tangentbordsavsökning kallas *koincidensavsökning*. Denna metod, som har varit mycket vanlig har allt mer förlorat principiell betydelse då tangentbordsavkodning nu enklast görs av speciellt konstruerade kretsar, jämför *ML15*. Inte desto mindre utgör momentet ett utmärkt exempel på hur yttre enheter kan anslutas och ges en programstyrd funktion.

Avsökning av tangentbord

Betrakta principschemat av tangentbordet. PORT A associeras till tangentbordets RADER och PORT B associeras till tangentbordets KOLUMNER.





Observera att 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.

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. Tangentkoderna definieras av figuren i marginalen.

UPPGIFT 3.17:
Skriv en instruktionssekvens som aktiverar rad för rad på tangentbordet. För varje rad ska kolumnerna läsas av. Därefter upprepas förfarandet. Använd portdefinitioner från `LABDEFS.S68`.

SLUT PÅ UPPGIFT 3.17.

UPPGIFT 3.18:
Ange i följande tabell, med hjälp av lösningen till föregående uppgift, 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 PÅ UPPGIFT 3.18.

UPPGIFT 3.19:
Modifiera subrutinen *CheckKbd* (i `ML5DRVR.S68`) där tangentbordet avkodas, enligt följande specifikation.

OBSERVERA
Denna subrutin utgör en omfattande programmeringsuppgift.

Tänk noga igenom din lösning.

Försök finna olika sätt att lösa uppgiften på. (Olika *implementeringar* av samma specifikation).

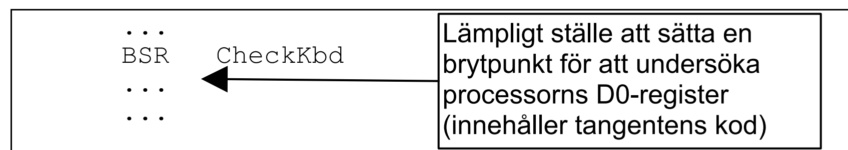
Specifikation:

Subrutin `CheckKbd` (check keyboard) skall avsöka hela tangentbordet (rad för rad) en gång.
Om ingen tangent är nedtryckt ska rutinen returnera `$FF` i processorns D0-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 D0 register

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

Du skall ha något av värdena 0-$F eller $FF i D0 då programmet stannar vid brytpunkten.

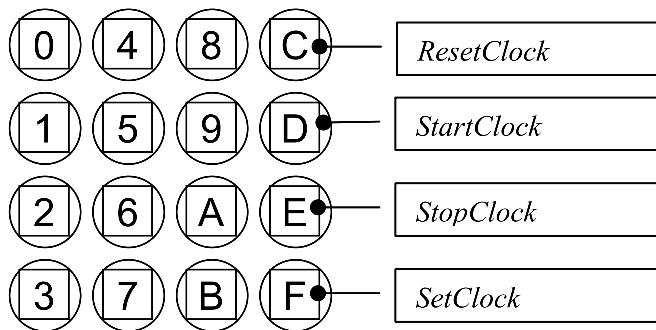


- Kontrollera att ALLA tangentnedtryckningarna ger korrekt värde i register D0.
- Kontrollera också att subrutinen "hänger" så länge en tangent är nedtryckt.

SLUT PÅ UPPGIFT 3.19.

När du fått programmet med den nya *CheckKbd*-rutinen att fungera som tidigare kan du starta med nästa uppgift, att "ställa" klockan.

UPPGIFT 3.20:
Utöka huvudprogrammet med en ny valbox och en ny subrutin *SetClock* (dummyversion) som ska ställa klockan. När du testar ditt huvudprogram så se till att subrutinen *SetClock* verkligen anropas då dess tangent trycks ned. Följande figur beskriver hur du nu ska disponera tangentbordets tangenter (A och B används ej):



SLUT PÅ UPPGIFT 3.20.

UPPGIFT 3.21:
När huvudprogrammet fungerar som det ska skriver du subrutinen *SetClock* enligt följande specifikation

Specifikation:
Subrutinen *SetClock* skall:

- startas av att tangent 'F' på tangentbordet, trycks ned
- stanna klockan (om den går) och släcka hela displayen
- läsa ett nytt klockvärde från tangentbordet. Placera detta i variabeln *clock*.
  - Klockvärdet skall matas in enligt: timme, minuter, sekunder och tiondelar.
  - Displaymodulen skall visa ändringarna (siffra för siffra) när ny tid matas in.
  - När hela klockvärdet (sex siffror: T.MM.SS.t, har matats in) är givet skall subrutinen *SetClock* avslutas.

Felhantering: (frivillig uppgift)

- Endast decimala siffror får placeras i variabeln *clock* (hexadecimala siffrorna $A-$F ignoreras vid inmatnings-kontroll)
- Kontroll skall ske så att klockvärden för minuter och sekunder inte överskrider 59 (exempelvis får 74 minuter inte förekomma). Felaktiga värden ignoreras vid inmatningskontroll.

SLUT PÅ UPPGIFT 3.21.

Avsnitt 3 Sammanfattning

Under detta avsnitt har du fått tillfälle att arbeta med flera vanliga problemställningar vid maskinnära programmering. IN- UT-matning, avkodning och synkronisering till en mänsklig operatör i realtid.

Du har sett hur en applikation kan skapas och kombineras med olika typer av hårdvara (*ML15* och *ML5*). Du har också sett hur villkorlig assemblering och av USE-direktiv kan användas och bidra till att källtexter kan organiseras på ett effektivt sätt.

Avsnitt 4

Programmeringsprojekt: Borrmaskinen

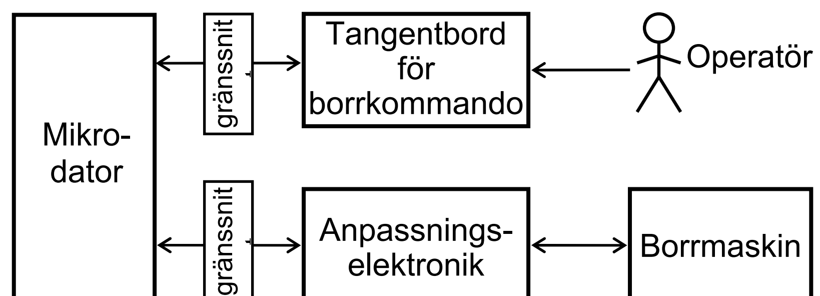
Syften:
I detta avsnitt ska du konstruera en "arbetsrobot" som borrar hål i ett arbetsstycke enligt ett givet schema. Roboten består av en borrmaskin, en styrpanel (ett tangentbord), anpassningselektronik och ett mikrodatorsystem. Du förutsätts nu ha de grundläggande färdigheter som krävs för att själv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

Målsättningar:
Målsättningen är att du, på egen hand, utfört alla uppgifter i avsnittet.

Inledning

Ett mikrodatorsystem ska användas för att styra och övervaka 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. *Hela* systemet (operatör och hårdvara) beskrivs av följande illustration.



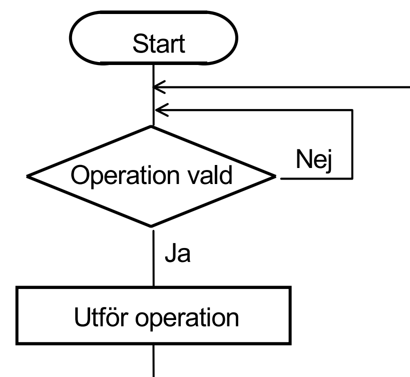
Systemets *funktion* realiseras med hjälp av programvara. Större delen av detta avsnitt kommer att handleda dig då du konstruerar denna programvara. Vi ska dock först bekanta oss lite bättre med systemet.

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:

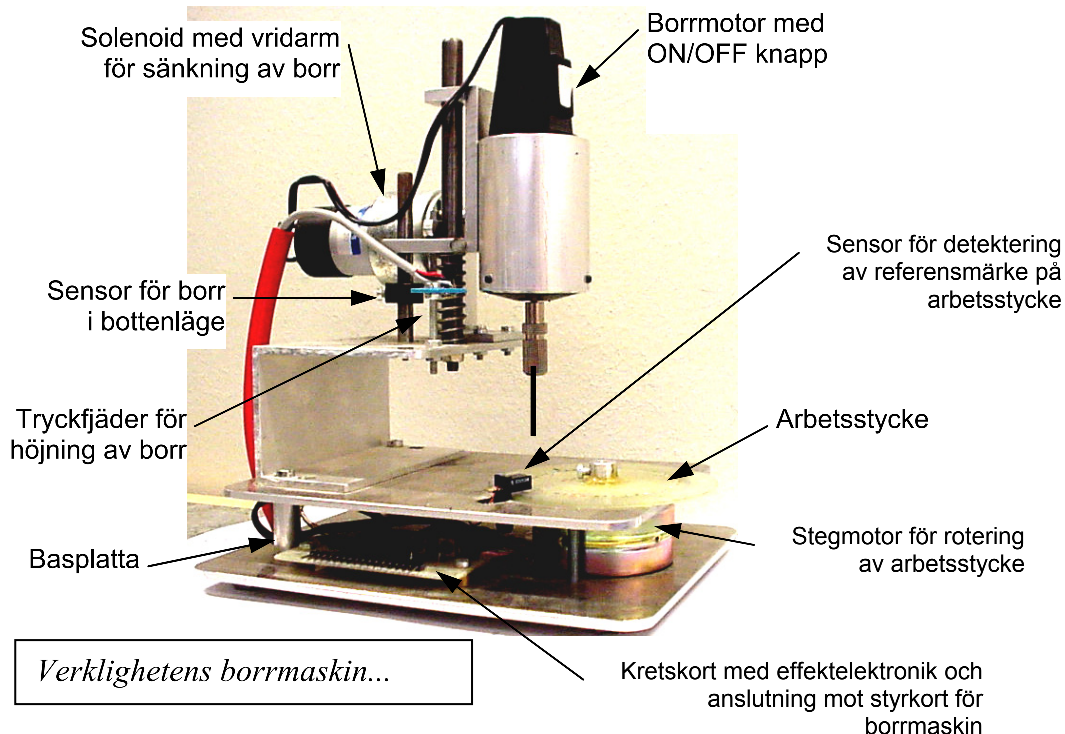
- *Solenoid*, en vridmagnet används för att sänka borrmotorn. När solenoiden inte är aktiverad pressar en tryckfjäder borrmotorn uppåt.
- Borrmotor, så att borret roterar
- Sensor som anger att borret har nått bottenläget (borrat genom arbetsstycket)
- Stegmotor för att kunna vrida arbetsstycket i steg om 7,5° vridning.
- Sensor som anger att arbetsstycket är i referensposition (startläge).
- ”Alarmindikator” kan användas exempelvis som larm vid fel eller som indikering på att arbetsstycket är färdigborrat.

Systemet skall i allt väsentligt vara operatörsstyrt. Se figur med flödesplanen i marginalen. Vi kan se detta som en övergripande specifikation.

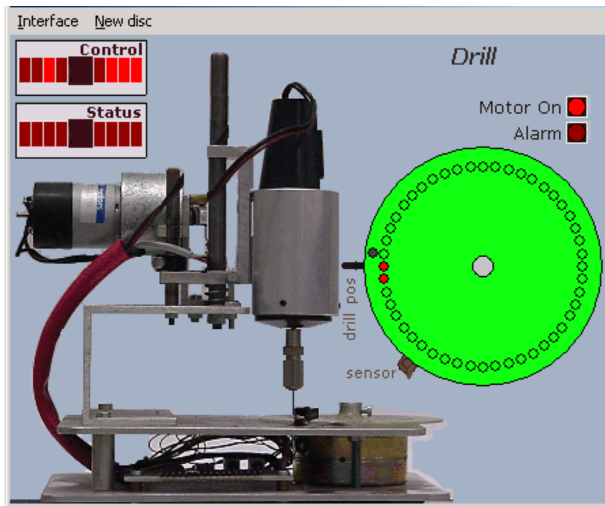


En första analys av uppgiften ger att systemet måste ha kommandon för följande operationer:

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



Verklighetens borrmaskin...



När du genomför detta programmeringsprojekt utnyttjar du en *borrmaskinsimulator*. Simulatorn innehåller förutom borrmaskinen, ett cirkulärt arbetsstycke och indikatorer för styr- och statusregister.

I de kommande momenten beskrivs de rutiner som krävs för att få systemet att uppfylla specifikationen.

Observera att när du arbetar med simulatorn kommer vissa realtidsaspekter inte att uppfyllas. Detta innebär att exempelvis vissa fördröjningsrutiner måste ändras beroende på om du utnyttjar simulatorn i läget ”Run”, Run Fast”. Vi återkommer speciellt till detta.

Borrmaskinsimulatorn

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

UPPGIFT 4.1:

Skapa först en underkatalog DRILL i ditt arbetsbibliotek. Du bör sedan spara samtliga filer i denna katalog. Redigera nu en fil med portadresser (`DRILLDEFS.S68`):

Om du använder MC68

```
DSInput EQU  $FFFFF011  Dip Switch Input
DCtrl   EQU  $80000     Drill Control Output
DStatus EQU  $80001     Drill Staus Input
```

Om du använder MD68k

```
DSInput EQU  $3E0013  Dip Switch Input
DCtrl   EQU  $3E0001  Drill Control Output
DStatus EQU  $3E0003  Drill Staus Input
```

Redigera också en fil `DTEST1` enligt följande:

```
* dtest1.s68

        DEFINE     MC68               eller MD68k…
        USE        DRILLDEFS.S68

        ORG        $4000

Loop    MOVE.B     DSInput,D0         Läs strömbrytarna
        MOVE.B     D0,DCtrl           Skriv till borrmaskinen
        MOVE.B     DStatus,D0         Läs statusregistret
        BRA        Loop
```

Assemblera `DTEST1.S68` och ladda till simulatorn.
Anm: Du kan visserligen assemblera DRILLDEFS men ingen användbar laddfil genereras av det.

SLUT PÅ UPPGIFT 4.1.

UPPGIFT 4.2:
Koppla nu IO-simulatorns *ML4 Dip switch input* och *Drill* enligt:

Om du använder MC68

Koppla *ML4 Dip switch input* till $FFFFF011
Koppla *Drill* till $80000

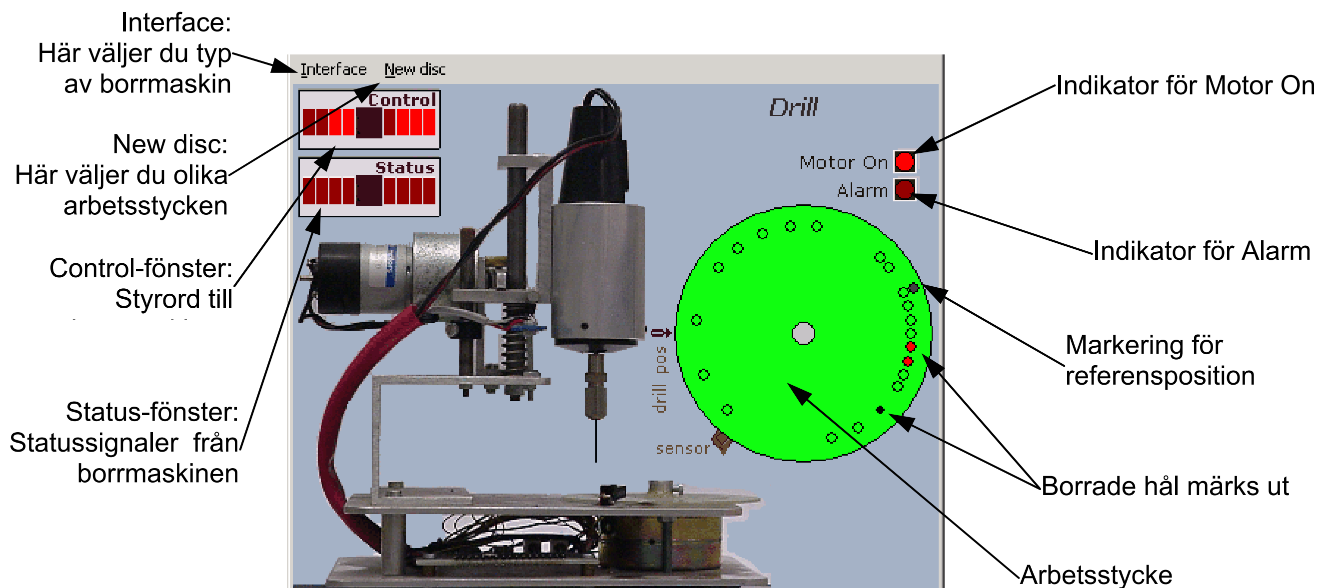
Om du använder MD68k

Koppla *ML4 Dip switch input* till $3E0013
Koppla *Drill* till $3E0001

Spara konfigurationen under namnet `DRILL1`.

Kontrollera att *Current target setup* är DRILL1.

Studera bilden som dyker upp på skärmen och jämför med följande figur. Överst till vänster har du möjlighet att välja *typ* av borrmaskin. Välj här CTH/CE som gränssnitt.

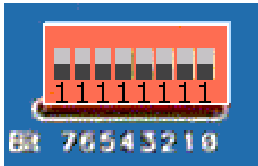


Välj olika *arbetsstycken* genom att klicka på 'New disc'. Klicka upprepade gånger på ett och samma arbetsstycke exempelvis 'Disc 4' och observera att markören för referensposition hamnar på olika ställen.

Du kan läsa mer om IO-simulatorn's borrmaskin i appendix C.

SLUT PÅ UPPGIFT 4.2.

UPPGIFT 4.3:

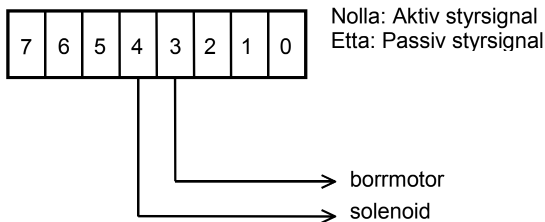


Innan du startar programmet DTEST1, som du tidigare laddat till simulatorn så måste du ge passiva styrsignaler till borrmaskinen. Passiva styrsignaler från processorn startar ingen aktivitet hos borrmaskinen. Här är en ETTA en passiv styrsignal. Välj därför att ettställa alla strömbrytarna på Dip-Switchen. På så sätt kommer $FF att läsas och skrivas till borrmaskinen enligt programmet.

```
Loop    MOVE.B      DSInput,D0        Läs strömbrytarna
        MOVE.B      D0,DCtrl          Skriv till borrmaskinen
        MOVE.B      DStatus,D0        Läs statusregistret
        BRA         Loop
```

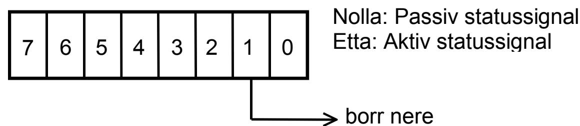
Starta nu programmet DTEST1 genom att välja 'Run' i simulatorn.

Utport: Drill Control

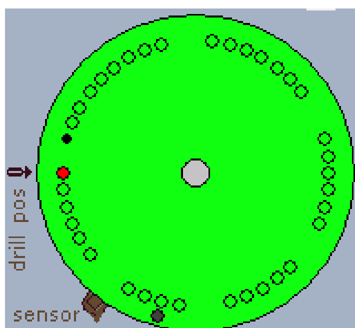


Bit 3 = 0: Borrmotorn startar
Bit 3 = 1: Borrmotorn stannar
Bit 4 = 0: Borret sänks
Bit 4 = 1: Borret höjs

Inport: Drill Status



Bit 1 = 0: Borret är i bottenläge
Bit 1 = 1: Borret är inte i bottenläge



RÖD MARKERING: Ett hål är borrat på ett fördefinierat ställe (korrekt ställe)
SVART MARKERING: Ett hål är borrat på utanför ett fördefinierat ställe (fel ställe)

Studera borrmaskinens styrord ('Drill Control' i marginalen). Starta borrmotorn genom att ställa in styrordet $F7 på DIP-SWITCH'arna. Klicka på strömbrytare 3 och observera hur indikatorn för bit 3 släcks i Control-fönstret på borrmaskinsimulatorn. Notera också att indikatorn för 'Motor On' tänds i borrmaskinsimulatorn.

Lägg märke till värdet i processorns register D0.

Borrmaskinens statusregister läses till register D0. Studera även statusregistret i borrmaskin-simulatorn och observera att bit 1 är tänd. Är denna lysdiod släckt indikerar detta att borret är i bottenläge. (Se 'Drill Status' i marginalen).

Klicka nu på strömbrytare 4 på DIP-SWITCH'arna för att ge styrordet $E7 och observera hur borret sänks. Efter en stund har borret arbetat sig genom arbetsstycket och indikatorn för bit 1 släcks i Status-fönstret. Borret är nu i bottenläge.

Ge slutligen styrordet $FF för att sätta passiva styrsignaler till borrmaskinen. Observera hur borret höjs och att borrmotorn stannar och att bit 1 i statusregistret tänds. Observera även hur en markering för ett borrat hål finns på skivan.

Betrakta nu arbetsstycket som innehåller ett antal cirklar. Dessa är markeringar för ett tänkt borrschema. När du borrar ett hål ges en röd markering när du borrar enligt det tänkta schemat för detta arbetsstycket. Borrar du utanför markeringarna markeras dessa hål som svarta och indikerar på så sätt ett felaktigt borrat hål.

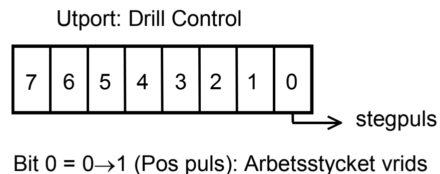
Du kan som sagt välja olika arbetsstycken genom att välja 'Disc 1' till 'Disc 4' i borrmaskinsimulatorn.

SLUT PÅ UPPGIFT 4.3.

Nu när du sett hur du kan borra *ett* hål ska vi se hur vi kan vrida arbetsstycket för att borra ett nytt hål i en annan position.

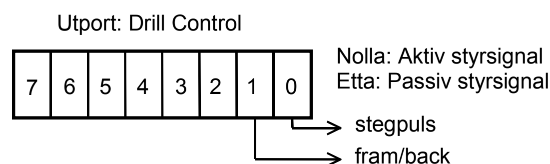
UPPGIFT 4.4:

För att vrida arbetsstycket krävs en positiv flank på bit 0 i styrordet till borrmaskinen. Utnyttja samma testförfarande och testprogram som i förra uppgiften och starta simulatorn med 'Run'. Ge styrorden $FF respektive $FE genom att upprepade gånger klicka på bit 0 på DIP-SWITCH'arna.



Observera att arbetsstycket vrids när register D0 ändras från $FE till $FF vilket alltså motsvarar en positiv flank.

Vrid nu arbetsstycket ett par steg till och försök på nytt att borra ett hål genom att ge styrordssekvensen $F7, $E7. Du kommer nu att få en ny markering på arbetsstycket som indikerar ett hål.

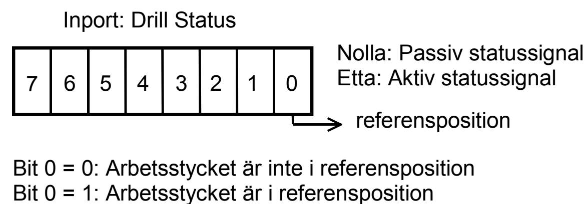


Arbetsstycket kan vridas både med- och moturs. Riktningen styrs via bit 1 i borrmaskinens styrord.

Bit 0 = 0→1 (Pos puls): Arbetsstycket vrids
Bit 1 = 0: Medurs vridningsriktning
Bit 1 = 1: Moturs vridningsriktning

Ge nu styrordet $F9 till borrmaskinen. Klicka sedan enbart på bit 0 på DIP-SWITCH'arna för att generera stegpulser. Observera nu hur arbetsstycket roterar medurs.

Avslutningsvis skall du undersöka hur man bestämmer *vart* 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 markeringen.



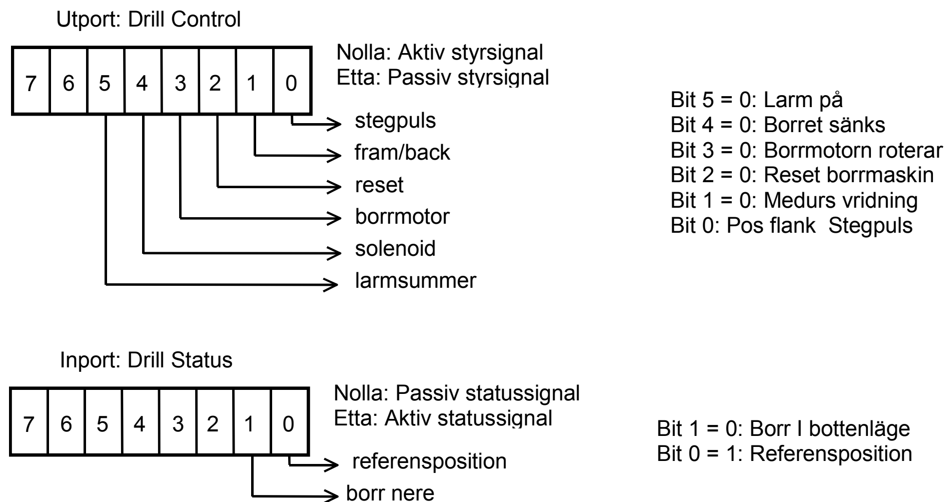
Bit 0 = 0: Arbetsstycket är inte i referensposition
Bit 0 = 1: Arbetsstycket är i referensposition

När arbetsstycket vrids upprepade gånger kommer markeringen förr eller senare att passera sensorn som är ansluten till bit 0 i borrmaskinens statusregister. Ge ett antal stegpulser genom att klicka på bit 0 på DIP-SWITCH'arna och observera att bit 0 i Status-fönstret tänds på borrmaskinen när arbetsstyckets referensposition passerar sensorn.

Kontrollera även hur en etta läses till register D0 när arbetsstycket är i referensposition. Om du stegat för långt kan du ändra bit 1 i styrregistret för att stega tillbaka.

SLUT PÅ UPPGIFT 4.4.

Du har nu undersökt flera av borrmaskinens styr- och statussignaler. En sammanställning av borrmaskinens styrregister och statusregister följer här.



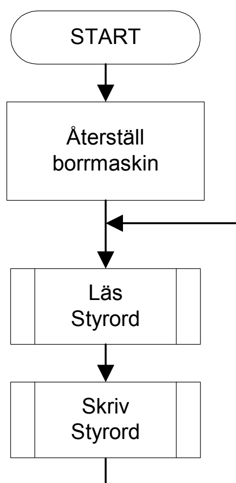
De enda signaler vi hittills inte behandlat är bit 2 och bit 5 i styrregistret. Bit 2 används för att utföra *Reset* på borrmaskinsimulatorn och bit 5 används för alarmfunktionen.

UPPGIFT 4.5:

Använd samma testprogram och testförfarande som tidigare och:

- Ändra bit 5 ett antal gånger och observera hur indikatorn för Larm ändras i simulatorn.
- Nollställ bit 2 och observera vad som händer med borrmaskinen när du försöker att starta och sänka borret.

SLUT PÅ UPPGIFT 4.5.



När du nu fortsättningsvis skall skriva program för borrmaskinen bör du ge en *Reset* signal till borrmaskinen innan de första styrorden ges. Se flödesdiagrammet i marginalen.

Efter att ha testat borrmaskinsimulatorns alla funktionerna är det nu dags att skriva ett enkelt testprogram som utför borrning i ett arbetsstycke.

För testprogrammet ger vi följande specifikation:

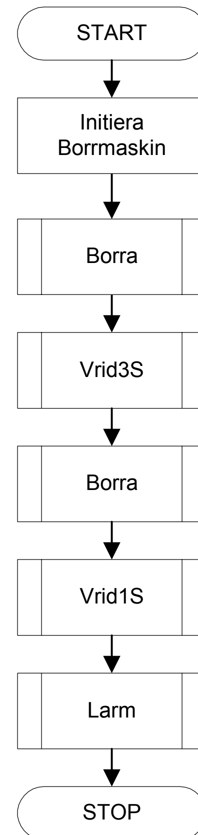
a) Ett hål borras
b) Arbetsstycket vrids *medurs* tre steg
c) Ett hål borras
d) Arbetsstycket vrids *medurs* ett steg
e) En larmsignal ges som indikation på att uppgiften är klar.

Specifikationen visas även i marginalen (nästa sida) som ett flödesdiagram. Det är här lämpligt att utforma flödesdiagrammet som ett antal subrutinanrop. Det gör att testprogrammet blir mer översiktligt och enklare att verifiera och prova.

UPPGIFT 4.6:

Redigera ett program i en ny källtextfil `DTEST2.S68` enligt följande, jämför med flödesplanen i marginalen. Spara programmet.

```
      USE          DRILLDEFS.S68

      ORG          $4000
      MOVE.B       #$FB,DCtrl
      JSR          Borra
      JSR          Vrid3S   Vrid 3 steg
      JSR          Borra
      JSR          Vrid1S   Vrid 1 steg
      JSR          Larm
Loop  BRA          Loop
```



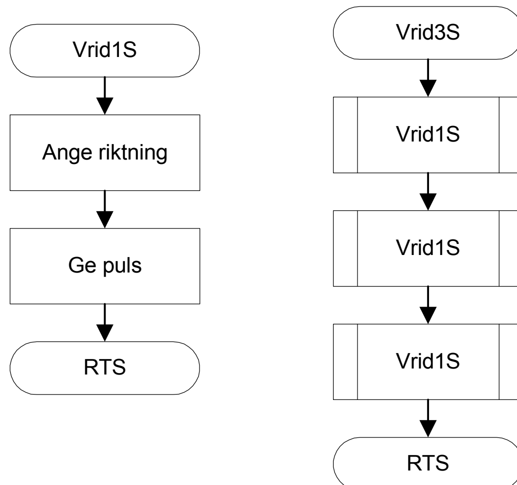
SLUT PÅ UPPGIFT 4.6.

UPPGIFT 4.7:
Betrakta igen punkterna a-e i specifikationen men på en mer detaljerad nivå med utgångspunkt från vad du sett i tidigare uppgifter.

Punkt b) och d), att vrida arbetsstycket, löstes genom att ge sekvensen

- Ange vridningsvinkel
- Ge stegpuls

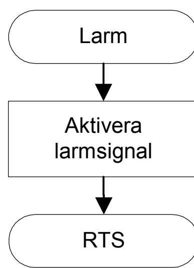
Lämpligtvis skrivs en subrutin som utför en vridning som anropas tre gånger för att åstadkomma en tre-stegs vridning. Se följande flödesplaner.



Skriv subrutinerna Vrid1S och Vrid3S i filen `DTEST2.S68`.

SLUT PÅ UPPGIFT 4.7.

UPPGIFT 4.8:



Larmsignalen i punkt e), att ge en larmsignal, realiseras genom att

- Starta larmet

Se även flödesdiagrammet i marginalen.

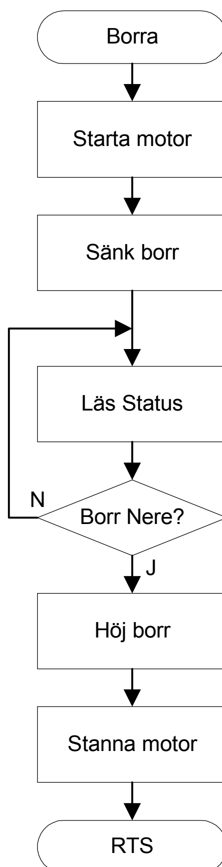
Skriv subrutinen `Larm` i filen `DTEST2.S68`.

SLUT PÅ UPPGIFT 4.8.

UPPGIFT 4.9:

Punkt a) och c), att borra *ett* hål, utför vi med sekvensen

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



Se även flödesdiagrammet i marginalen.

Skriv subrutinen `Borra` i filen `DTEST2.S68`.

SLUT PÅ UPPGIFT 4.9.

UPPGIFT 4.10:

Assemblera din källtext och ladda till simulatorn. Använd 'Disc 1' när du testar ditt program. Rätta eventuella fel så att subrutinerna `Borra`, `Vrid1S`, `Vrid3S` och `Larm` fungerar som avsett.

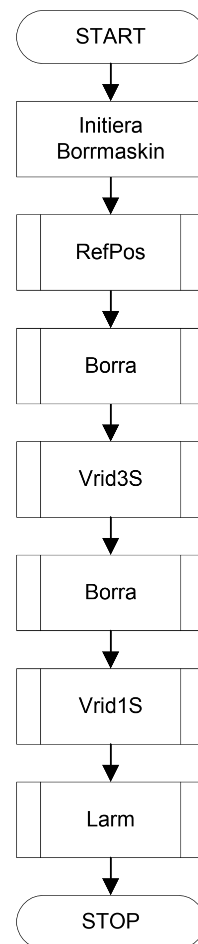
SLUT PÅ UPPGIFT 4.10.

Föregående testprogram är specificerat så att det vid uppstart kommer att borras ett hål direkt utan att någon hänsyn tas till arbetsstyckets position.

Vi ändrar nu specifikationen så att vi tar hänsyn till vart första hålet skall borras på kortet.

Vi gör en ny specifikation. Studera flödesplanerna i marginalen:

a) Arbetsstycket vrids till referensposition
b) Ett hål borras
c) Arbetsstycket vrids medurs tre steg
d) Ett hål borras
e) Arbetsstycket vrids medurs ett steg
f) En larmsignal ges som indikation på att uppgiften är klar.



Det nya som tillkommit är att vi vill vrida arbetsstycket till referensposition innan borrningen startar.

UPPGIFT 4.11:
Redigera din källtext `DTEST2.S68` och utöka denna med ett *anrop* av subrutinen `RefPos` i nästa uppgift ska du koda rutinen.

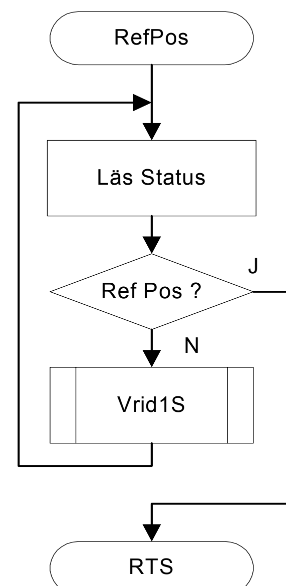
SLUT PÅ UPPGIFT 4.11.

Rutinen `RefPos` som tillkommit i den nya specifikationen beskrivs nu. Denna skall alltså söka upp *referenspositionen* på arbetsstycket. Du såg tidigare hur bit 0 i statusregistret anger huruvida arbetsstycket är placerat i referensposition eller inte.

Att söka upp referenspositionen på arbetsstycket, kan alltså utföras genom att

- Ge en stegpuls
- Undersöka om indikatorn för referenspositionen är satt (om bit 0 = 1)
- Om bit 0 = 0, börja om från början

UPPGIFT 4.12:
Redigera rutinen `Refpos` i källtexten. Assemblera! Ladda till simulator och testa. Rätta eventuella fel.



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

SLUT PÅ UPPGIFT 4.12.

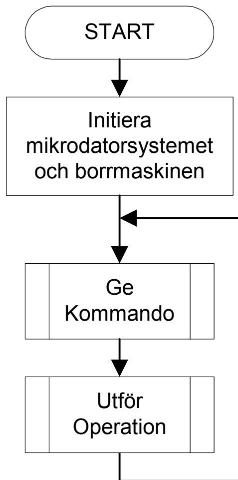
Du har nu skrivit ett program som först letar upp referenspositionen på arbetsstycket, borrar ett hål, vrider arbetsstycket tre steg och slutligen borrar det sista hålet.

Detta har du förmodligen gjort genom att använda kodsekvenser liknande:

```
* Ge Styrord till borrmaskinen
  MOVE.B          #StyrOrd,DCtrl
```

Detta fungerar utmärkt om ditt program följer en given flödesplan och därmed en given sekvens av styrord till borrmaskinen.

Längre fram skall vi skriva ett mer generellt program där en operatör kan ge kommandon och därmed styra borrmaskinen. Se flödesplanen i marginalen



I ett operatörsstyrt system kan *olika* sekvenser av styrsignaler komma att ges till borrmaskinen. Vi måste därför ha kontroll över vilka styrsignaler som är aktiva respektive passiva ut till borrmaskinen när ”nästa kommando” ges eftersom vi vanligtvis bara vill ändra en bit i taget i borrmaskinens styrregister (DCtrl) och måste låta övriga bitar vara opåverkade. Detta kan utföras med AND- och OR-instruktioner för att nollställa respektive ettställa bitar i borrmaskinens styrregister.

Betrakta följande kodsekvens, jämför med den tidigare använt.

För att nollställa en bit används:

```
MOVE.B DCtrl,D0     Läs nuvarande styrord
ANDI.B #%xxxxxxx,D0 Nollställ önskat bit
MOVE.B D0,DCtrl     Skriv nytt styrord
```

eller för att ettställa en bit används:

```
MOVE.B DCtrl,D0     Läs nuvarande styrord
ORI.B  #%xxxxxxx,D0 Ettställ önskat bit
MOVE.B D0,DCtrl     Skriv nytt styrord
```

Anm.
MC68000 tillhandahåller speciella instruktioner för bitmanipulering. Läs om instruktionerna BTST, BSET, BCLR och BCHG i instruktionslistan.

Observera att uppgiften i stora drag handlar om att undan för undan *ändra en bit åt gången* i borrmaskinens styrregister.

Kodsekvenserna ovan fungerar tyvärr *inte* i vårt fall eftersom styrregistret (DCtrl) 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). Denna utport är av type "Write Only" och då vi försöker läsa från den (`MOVE.B DCtrl,D0`) får vi ett nonsensresultat.

Det krävs därför att du skapar en variabel i minnet som är innehåller en kopia av styrordet. Detta gör du enklast genom att utnyttja `DS`-direktivet. Ge kopian symbolnamnet 'DCCopy' enligt:

```
DCCopy     DS.B  1       Drill Control Copy
```

Följande exempel visar på hur du nu skall ändra styrbitar till borrmaskinen.

För att nollställa en bit används nu:

```
MOVE.B DCCopy,D0    Läs kopian av styrordet
ANDI.B #%xxxxxxx,D0 Nollställ önskat bit
MOVE.B D0,DCtrl     Skriv nytt styrord
MOVE.B D0, DCCopy   Spara nya kopian
```

Tips.
Gör samma sak med BSET och BCLR instruktionerna.

för att ettställa en bit används:

```
MOVE.B DCCopy,D0    Läs kopian av styrordet
ORI.B  #%xxxxxxx,D0 Nollställ önskat bit
MOVE.B D0,DCtrl     Skriv nytt styrord
MOVE.B D0, DCCopy   Spara nya kopian
```

Realtid och fördröjningsrutiner

Vi skall nu studera realtidsaspekter och fördröjningsrutiner. Du har sett hur du kan köra simulatorn på tre olika sätt, nämligen:

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

UPPGIFT 4.13:

Testa på nytt den förra uppgiften du sparade. Välj 'Run Fast'.

Vad händer? Försök förklara varför.

____

____

____

SLUT PÅ UPPGIFT 4.13.

Troligen "hänger" programmet i rutinen `RefPos` där vi ger ett antal pulser till stegmotorn för att vrida arbetsstycket fram till referenspositionen.

Följande kodsekvens som utför en vridning av arbetsstycket medurs är hämtad från subrutinen `Vrid1S`

```
Vrid1S
*     Medurs, Bit0=0
      MOVE.B #%11111100,DCtrl

*     Bit0=1=Pos Flank
      MOVE.B #%11111101,DCtrl      Medurs, Bit0=0
      RTS
```

Observera
Resonemanget vi för här kan också tillämpas på verklig hårdvara. Skillnaden är då att processorn exekverar betydligt fler instruktioner/sekund än vad simulatorn gör.

I kodsekvensen sker bland annat följande:

- en negativ flank ges
- direkt därefter ges en positiv flank för att få arbetsstycket att rotera.

Borrmaskinssimulatorn tar 200 ms att utföra en vridning. Simulatorn utför c:a 10 instruktioner vid 'Run' och betydligt fler vid 'Run Fast'.

Vid 'Run Fast' kommer pulserna till stegmotorn med såpass 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.

Nästan oberoende av vad man vill göra med borrmaskinen så krävs någon fördröjning innan man kan utföra nästa moment. Några exempel är

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

När vi nu är på väg att påbörja arbetet med att skriva det riktiga programmet för borrmaskinen så måste vi beakta de realtidsfördröjningar som kan komma att krävas. Alltså, vi bör ha två *olika* fördröjningsrutiner. En fördröjningsrutin som är tänkt att användas med 'Run' och en annan, som lämpar sig vid 'Run Fast'.

Fördröjningsrutinen bör utformas på ett sådant sätt att den kan skapa olika fördröjningar. Vi väljer här att specificera en rutin som kan skapa ett antal fördröjningar om 500 ms. Antalet sådana fördröjningar överförs till rutinen i register D0.

```
*********************************************
*SUBRUTIN - DELAY
* Beskrivning:  Skapar en fördröjning om
* ANTAL x 500 ms.
* Anrop:   MOVE.B #6,D0
*    Fördröjning 6*500 ms = 3 s
*          JSR   DELAY
* Indata:  Antal intervall,om 500 ms i D0
*
* Utdata:  Inga
* Register-påverkan:  D0
* Anropad subrutin:   Ingen.
*********************************************

DELAY      MOVE.L     D1,-(SP)
           TST.L      D0
           BEQ        DERET

DELOOPO
* inre slinga
           MOVE.L     #DCONST,D1
DELOOP
           DBEQ       D1,DELOOP
* yttre slinga
           DBEQ       D0,DELOOPO

DERET      MOVE.L     (SP)+,D1
           RTS
```

Anm.
I stället för
`DBEQ     D1,DELOOP`
kan kodsekvensen
```
SUBI.L  #1,D1
BNE     DELOOP
```
användas

UPPGIFT 4.14:

Skriv in DELAY-rutinen ovan under filnamnet `DELAY.S68`. Spara filen. Observera att fördröjningskonstanten (inre slingan) är odefinierad. Du ska strax få tillfälle att bestämma denna.

SLUT PÅ UPPGIFT 4.14.

Tidigare, då du testade nya rutiner använde du strömställarna på IO-simulatorns DIP-SWITCH input för att ge olika indata till dina kodsekvenser eller subrutiner. Utdata från dina rutiner kan du sedan presentera via exempelvis *ML4 parallel output*. Detta är ett enkelt att sätt kontrollera att programmet fungerar korrekt.

Att testa rutinen DELAY är inte lika enkelt. Följande uppgift ger dig övning hur du kan testa DELAY-rutinen och liknande kodsekvenser.

UPPGIFT 4.15:
Fördröjningskonstanten i inre slingan ska bestämmas. Följande program kan då vara behändigt.

```
      ORG         $4000
      CLR.L       D1
mloop:
      CLR.L       D0
      MOVE.B      Inport,D0
      JSR         TDELAY
      NOT.B       D1
      MOVE.B      D1,Utport
      BRA         mloop

TDELAY            TST.L D0
      BEQ         TDERET

TDELOOP
      DBEQ        D0,TDELOOP

TDERET
      RTS
```

Skapa en källtext `TDELAY.S68` och redigera enligt ovanstående. Använd IO-simulatorns DIP-SWITCH'ar för att ge test-indata till rutinen DELAY (Inport) . Använd *ML4 Parallel Output* som Utport för att ge indikation på att ett pass av DELAY-rutinen är klar.

Redigera, assemblera och ladda programmet till simulatorn. Anslut IO-simulatorerna och välj lämpliga adresser som du i vanlig ordning definierar med EQU-direktiv.

Se till att alla strömställarna på DIP-SWITCH input är nollställda så att du läser NOLL från inporten. Pröva dig fram med olika fördröjningskonstanter (såväl vid 'Run' som 'Run Fast') så att indikatorerna blinkar i rätt takt (De ska tändas och släckas 20 ggr under en 10 sekunders period).

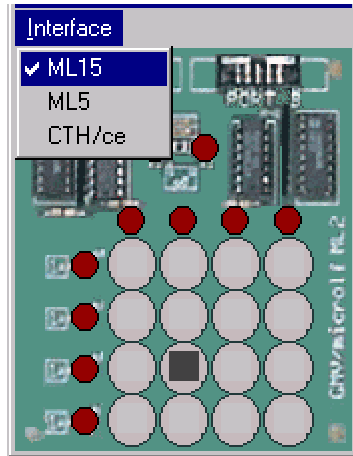
Observera att vi inte är väldigt noga med noggrannheten här, det är ju trots allt en simulering.

Då du bestämt fördröjningskonstanterna i de olika fallen kan du göra färdigt DELAY, exempelvis med följande:

```
#ifdef RUNFAST
DCONST      EQU   ????
#else
DCONST      EQU   ???
#endif
```

SLUT PÅ UPPGIFT 4.15.

Tangentbord och borrmaskin



Du har i tidigare avsnitt bekantat dig med det enkla tangentbordet *ML2*. Detta skall vi använda för att styra vår borrmaskin.

Vi väljer här att utnyttja gränssnittet *ML15* eftersom detta tillhandahåller färdig avkodningslogik för tangentnertryckningar. När vi läser från *ML15* får vi tangentnumret som ett binärtal [0000,1111] direkt från tangentbordet.

Vi repeterar programmerarens bild av gränssnittet *ML15*.

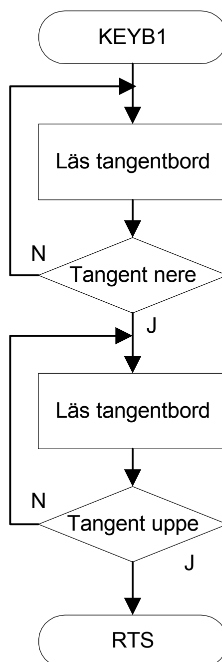
7	6	5	4	3	2	1	0
DAV	0	0	0	B3	B2	B1	B0

Bit 7, DAV: Data Valid; Statusbit som anger nedtryckt tangent
b_7=1: Ingen tangent är för tillfället aktiverad på tangentbordet.
b_7=0: En tangent är aktiverad

Bit 6-4, 0: Används ej.

Bit 3-0, B3-B0: Tangentnummer; Anger aktuell tangentnedtryckning.

Observera att du inte får ha två aktiverade tangenter på samma gång. Observera också att om ingen tangent trycks ner så ettställs bit 7.



I marginalen finner du ett flödesdiagram över den tangentbordsrutin du skall använda tillsammans med borrmaskinen. Observera att rutinen består av två slingor. I den första slingan inväntas en tangentnedtryckning och i den andra slingan avvaktas att tangenten skall släppas upp igen. Detta innebär att när vi anropar tangentbordsrutinen så kommer programmet att ”hänga” där tills att en tangent trycks ner och sedan släpps. På så sätt förhindras att EN tangentnedtryckning startar samma aktivitet mer än EN gång hos borrmaskinen.

UPPGIFT 4.16:
Redigera i en ny fil `KEYML15.S68`. Skriv en tangentbordsrutin enligt följande specifikation. Testa på lämpligt sätt och övertyga dig om att den fungerar enligt specifikationen.

```
***********************************************
* SUBRUTIN KEYB1
* Beskrivning:
* Rutinen används vid simulering
* Tangentbordet ML2 använder interfacet ML15.
* Anrop:     JSR KEYB1
* Indata:    Inga
* Utdata:    Tangentnummer (0-F) för nedtryckt
*            tangent i register D0.
* Reg-påverkan: D0
* Anr subr:     Ingen
***********************************************
```

SLUT PÅ UPPGIFT 4.16.

UPPGIFT 4.17:
Den önskade numreringen (kodningen) av tangenterna för de fortsatta uppgifterna visas i figuren till vänster nedan. Tangentbordet med gränssnittet *ML15* vi använder har en annan numrering (tangentkodning). Undersök denna numrering genom att utnyttja ett av testprogrammen du skrivit tidigare och fyll i *ML15's* numrering nedan till höger.

Kolumn

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

Kolumn

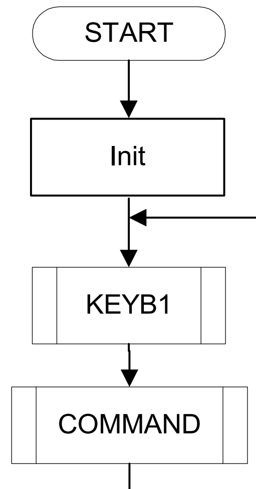
Rad	7	6	5	4
3				
2				
1				
0				

Du kan enkelt göra en konvertering med hjälp av en tabell som innehåller de *önskade* tangentkoderna *i den ordning* de uppträder hos *ML15*.

Modifiera nu rutinen KEYB1 (i filen `KEYML15.S68`) på lämpligt sätt så att returvärdet i register D0 motsvaras av den *önskade* tangentbordskodningen. Assemblera och testa den nya versionen av KEYB1. Du kommer att få användning för denna fil under kommande moment.

SLUT PÅ UPPGIFT 4.17.

Inledning till borrmaskinsstyrning



Här kommer nu specifikationen till den första riktiga versionen av borrmaskinen. Som vi beskrev i ett tidigare kapitel så skall borrmaskinen styras från ett tangentbord.

Av flödesdiagrammet i marginalen framgår huvudprogrammets struktur. När systemet väl startats, kommer det hela tiden att utföra den uppgift som operatören ger kommando om. Programmet läser av tangentbordet och utför därefter den valda operationen.

Innan programslingan startas måste systemet initieras. Här sker all nödvändig initiering av mikrodatorsystem och borrmaskin.

De olika val av operationer som kan göras visas i följande tabell.

tangent nr	Operation	subrutin
0	starta borrmotorn	START
1	stoppa borrmotorn	STOP
2	sänk borret	DOWN
3	höj borret	UP
4	rotera arbetsstycket medurs ett steg	STEP
5	borra ett hål	DRILL
6	stega arbetsstycket till referensposition	REFPO
7	borra hål längs cirkeln enligt mönster	AUTO

Kolumn

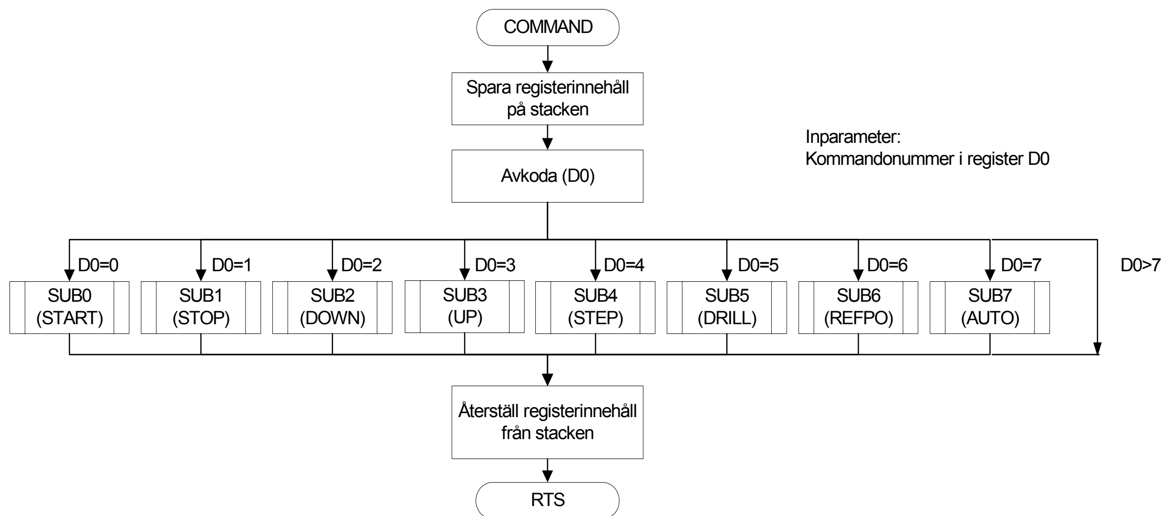
Rad	7	6	5	4
3	0	1	2	3
2	4	5	6	7
1	8	9	A	B
0	C	D	E	F

Tangentbordsrutinen har du klarat av under en tidigare uppgift. Tangentbordet ska ha den kodning av de olika tangenterna som visas i figuren i marginalen. Trycker du på tangenten överst till höger på tangentbordet skall alltså borret *höjas,* enligt tabellen ovan.

Betrakta på nytt flödesdiagrammet över huvudprogrammet. Först initieras systemet, sedan läses tangentbordet och därefter utförs önskat kommando.

Vid initieringen ges stackpekaren ett lämpligt värde och borrmaskinen sätts i ett *passivt* tillstånd. Med passivt tillstånd menas att alla styrsignaler till borrmaskinen skall sättas passiva dvs ETTOR skrivs till borrmaskinens styrregister.

Register D0 innehåller efter tangentbordsrutinen alltså information om önskad operation. Rutinen COMMAND som utför operationen använder register D0 som inparameter. En flödesplan över subrutinen COMMAND visas härnäst.



Som flödesplanen visar sparas nödvändiga registerinnehåll på stacken innan ett hopp utförs till någon av de olika subrutinerna enligt tabellen ovan. För en tangentnedtryckning vars värde överstiger sju skall inget kommando utföras. Slutligen återställs registerinnehållen innan återhopp sker.

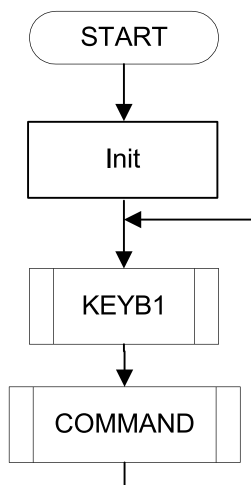
För att få ett överskådligt program *skall* hopp till de olika subrutinerna som utför vald operation ske via en *hopptabell*. Vi illustrerar principen för en sådan genom att exemplifiera rutinen COMMAND.

```
*************************************************
*SUBRUTIN - COMMAND
*Beskrivning:     Rutinen avgör vilken
*kommandosubrutin som skall anropas och anropar
*denna.
*Anrop:           JSR   COMMAND
*Indata:          Kommandonummer i reg A
*Utdata:          Inga
*Reg-påverkan:    Ingen
*Anrop subr:      SUB0 - SUB7
*************************************************
*************************************************
MAX    EQU    7

COMMAND      MOVEM.L     D0/A0,-(SP) spara...
*
      CMPI.L      #MAX,D0
      BHI         COMEX
*
      ASL.B       #2,D0
      MOVEA.L     #JUMPTAB,A0
      MOVEA.L     (A0,D0),A0
      JSR         (A0)
*
COMEX MOVEM.L     (SP)+,D0/A0 återställ
      RTS

************************************************
*     Tabell med subrutinadresser
JUMPTAB
      DC.L SUB0,SUB1,SUB2,SUB3,SUB4,SUB5,SUB6,SUB7
**********************************************
```

```
*************************************************
* Subrutiner
* SUB0,SUB1,etc byts senare ut mot START,STOP,etc

SUB0  MOVE.B      #0,ParOut
      RTS

SUB1  MOVE.B      #1,ParOut
      RTS

SUB2  MOVE.B      #2,ParOut
      RTS

SUB3  MOVE.B      #3,ParOut
      RTS

SUB4  MOVE.B      #4,ParOut
      RTS

SUB5  MOVE.B      #5,ParOut
      RTS

SUB6  MOVE.B      #6,ParOut
      RTS

SUB7  MOVE.B      #7,ParOut
      RTS
```

Subrutinen COMMAND sparar först registerinnehåll på stacken och kontrollerar därefter att inparameter (kommandonumret i register D0) inte överstiger 7. Därefter multipliceras kommandonumret med fyra eftersom varje adress i JUMPTAB upptar fyra bytes. Slutligen placeras adressen till den rutin som motsvaras av kommandonumret i register A0 och därefter utförs subrutinanropet.

De olika subrutinerna representerar var sin operation som operatören kan utföra med borrmaskinen. Subrutinnamnen SUB0 och SUB1 i hopptabellen byter vi senare ut mot rutinerna START och STOP etc, allt eftersom rutinerna implementeras.



De dummy-subrutiner som används inledningsvis skriver bara önskat numret för "sin" funktion till en utport. Syftet är att underlätta testen av huvudprogrammet. Du ansluter lämpligen *ML4 parallell output* som IO-simulator. På så sätt kan du enkelt se att korrekt kommandonummer skrivs till utporten av de olika subrutinerna när du testar huvudprogrammet.

Rekapitulera nu huvudprogrammet som anropar COMMAND. Först skall mikrodatorsystem och borrmaskin initieras, därefter väntar programmet i tangentbordsrutinen KEYB1 tills någon tangent aktiverats (och släpps upp) innan önskad operation utförs via COMMAND.

Att initiera mikrodatorsystemet består här i att ge stacken ett värde. Välj här $2000 som BOS (*Bottom Of Stack*).

```
      MOVEA.L    #$2000,SP
```

Att initiera borrmaskinen är nästa steg....

```
* Sätt passiva styrsignaler till borrmaskinen
      MOVE.B     #$FF,DCtrl
* och kopian
      MOVE.B     #$FF,DCCopy
```

Följande adresser skall användas för borrmaskinuppgiften:

```
* Adress till tangentbord för MC68
Key     EQU   $89C00
* Adress till tangentbord för MD68K
Key     EQU   $413801

* Adresser till borrmaskinens register för MC68
DCtrl   EQU   $80000          Drill Control Register
Dstatus EQU   $80001          Drill Status Register

* Adresser till borrmaskinens register för MD68K
DCtrl   EQU   $3E0001         Drill Control Register
Dstatus EQU   $3E0003         Drill Status Register

* Parallell IO för MC68
DSInput EQU   $FFFFF011       Dip Switch Input (ML4)
ParOut  EQU   $FFFFF019       Parallel Output (ML4)
* Parallell IO för MD68K
DSInput EQU   $3E0013         Dip Switch Input (ML4)
ParOut  EQU   $3E0011         Parallel Output (ML4)

* Övriga Definitioner
Start   EQU   $4000           Program Startadress
BOS     EQU   $2000           "Bottom Of Stack"
```

UPPGIFT 4.18:

Redigera adressdefinitioner och spara i filen `IODEFS.S68`. Använd villkorlig assemblering, så som du lärde dig i avsnitt 3, för att skilja på maskintyp.

SLUT PÅ UPPGIFT 4.18.

Rutinen KEYB1 du skrev tidigare skall vara sparad i filen `KEYML15.S68`. Huvudprogrammet kan nu utformas enligt följande

```
        USE   IODEFS.S68      adressdefinitioner
        ORG   Start

        ---           Initiera mikrodatorsystem
        ---           Initiera borrmaskinsimulator
Loop    JSR   KEYB1   Invänta vald operation
        NOP
        JSR   COMMAND Utför vald operation
        BRA   Loop


        USE   KeyML15.s68     Lägg till KEYB1

        ---           Här följer rutinen COMMAND
        ---           ... och hopptabell
        ---           ... och Dummysubrutiner
* Placera alla datadefinitioner sist
DCCopy  DS.B  1       Kopia av styrord
```

NOP-instruktionen kan vara värdefull att ha vid test av programmet. Det kan exempelvis vara lämpligt att sätta en brytpunkt på just denna adress innan programmet går vidare till COMMAND-rutinen.

Observera var din kopia av styrordet (DCCopy) placeras. När du senare assemblerat ditt huvudprogram kan du undersöka med hjälp

av listfilen vilken adress variabeln placerats på. Denna kan vara värdefull att undersöka under test och felsökning.

UPPGIFT 4.19:

Redigera en fil med namnet `MAIN1.S68` där du skriver in huvudprogrammet. (Vi skall i fortsättningen skapa ett antal filer; Main2, Main3, etc).

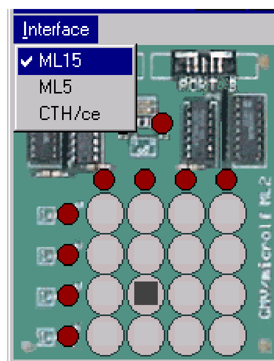
SLUT PÅ UPPGIFT 4.19.

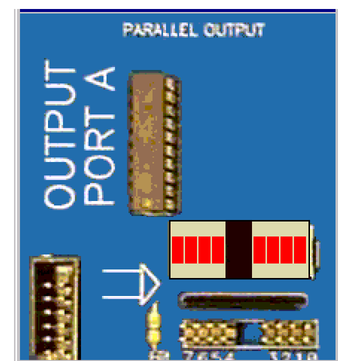
UPPGIFT 4.20:

Assemblera nu filen `MAIN1.S68`. Rätta eventuella fel och ladda denna sedan till simulatorn. Koppla nödvändiga IO-simulatorer *ML4 Parallel output* och *ML2 Keyboard* där du väljer gränssnittet *ML15*.

När du testar huvudprogrammet ska du se följande IO-simulatorer framför dig:

Rad \ Kolumn	7	6	5	4
3	0	1	2	3
2	4	5	6	7
1	8	9	A	B
0	C	D	E	F





Starta huvudprogrammet genom att välja 'Run' i simulatorn. När du sedan trycker ner (och släpper upp) tangenter på *ML2* ska du kunna utläsa ”vald operation” hos lysdioderna på den parallella utporten. Tangentbordets nummerordning visas igen i marginalen.

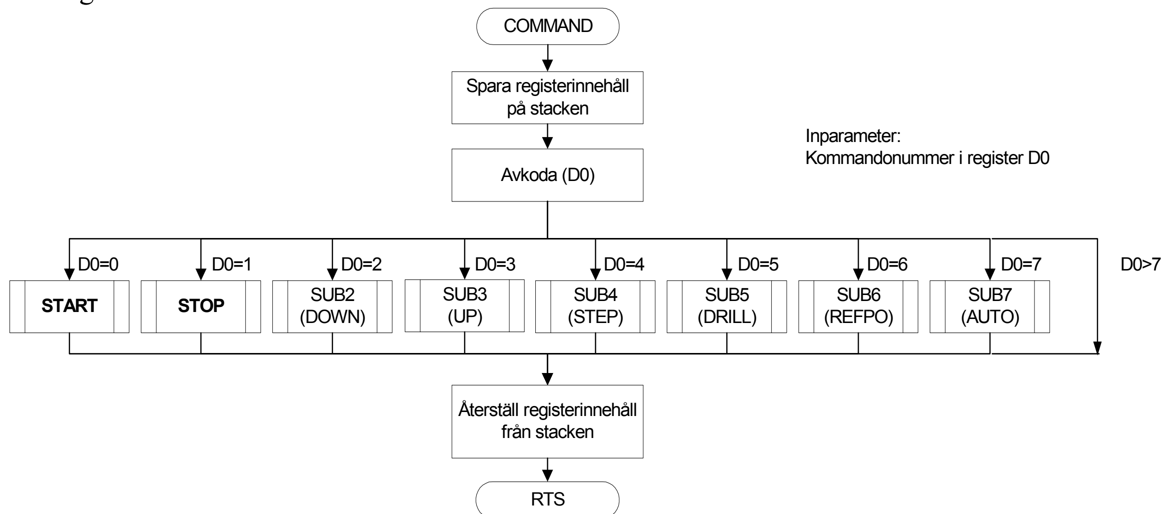
Trycker du på tangent nummer 7 väljer du att utföra operation nummer 7 och då ska subrutin nummer 7 utföras. Motsvarande dummy-subrutin skriver värdet 7 till utporten som därför skall visas binärt här. Väljer du ett tangentnummer högre än 7 skall detta inte visas.

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

SLUT PÅ UPPGIFT 4.20.

Specifikationer av subrutiner till borrmaskinen

Här följer nu de specifikationer som är nödvändiga för att kunna styra borrmaskinen. Vi börjar med rutinerna Start och Stop. Detta är som bekant de två första rutinerna i COMMAND. Betrakta på nytt flödesdiagrammet för COMMAND.



Vi har tidigare sett hur vi manipulerar borrmaskinens styrregister genom att bitvis nollställa eller ettställa en enstaka bit och samtidigt underhålla en kopia av det senast skrivna styrordet. Vi ska först och främst isolera denna kod i två subrutiner, OUTONE respektive OUTZERO. Dessa rutiner kommer sedan att användas från andra rutiner för att utföra olika kommandon.

Subrutinen **OUTZERO** sköter utmatningen av styrsignaler (styrordet) till borrmaskinen, ifall en av styrsignalerna skall nollställas. En kopia av styrordet lagras på minnesadressen **DCCopy**. Subrutinen **OUTZERO** har följande specification:

```
* Subrutin OUTZERO.
* Nollställer en av bitarna i kopian DCCopy och
* utporten DCTRL
* Vilken bit som nollställs bestäms av innehållet
* i D1-registret (0-7) vid anrop av subrutinen.
* Om innehållet > 7 utförs ingenting.
*
* Anrop:             MOVE.L     #5,D1
*                    JSR        OUTZERO
* Här skall bit nummer 5 i styrordet nollställas
*
*Indata:             D1-registret skall innehålla
                     ett tal 0-7, som är numret på
                     den bit i styrsignalordet som
                     skall nollställas. Bitnumrering
                     framgår av följande figur.
```

```
* Utdata:                Inga
* Registerpåverkan:      Ingen
* Anropade subrut:       Inga
```

Flödesdiagram över OUTZERO

UPPGIFT 4.21:

Rita ett flödesdiagram över subrutinen OUTZERO.
Skapa en källtextfil `SUB2.S68` och skriv subrutinen OUTZERO.
(Hur du skall testa subrutinen beskrivs i nästa uppgift).

SLUT PÅ UPPGIFT 4.21.

UPPGIFT 4.22:

Du kan testa din OUTZERO på ett enkelt sätt.

Först skriver du ett litet huvudprogram som anropar din subrutin. Du måste manipulera vissa data (här: DCCopy) innan du gör subrutinanropet.

```
#define           TEST

#ifdef            TEST
      USE         IODEFS.S68
      ORG         Start
* Startvärde till DCCopy
      MOVE.B      #$FF,DCCopy
* Läs bit som skall nollställas
      MOVE.B      DSInput,D1
      JSR         OUTZERO     .. och nollställ

* visa kopian på ljusdioderna.
      MOVE.B      DCCopy,ParOut

      JMP         Start
*
#endif
* Rutin som ska testas...

OUTZERO ...
      ...
      RTS

#ifdef            TEST
DCCopy            DS.B  1
#endif
```

Observera hur vi använder villkorlig assemblering för att tillhandahålla ett enkelt testprogram. Då vi övertygat oss om att "rutin under test" fungerar som den ska kommenterar vi bort raden

```
      #define     TEST
```

därefter kan vi inkludera denna källtext på vanligt sätt i det ordinarie huvudprogrammet.

Du kan alltså använda DIP-SWITCH’arna för att ange vilken bit du önskar nollställa med OUTZERO. Resultatet läser du av på lysdioderna.

Testa och eventuellt ändra i OUTZERO så att denna fungerar som den ska.

SLUT PÅ UPPGIFT 4.22.

Subrutinen OUTONE sköter utmatningen av styrsignaler (styrordet) till borrmaskinen, ifall en av styrsignalerna skall ettställas. En kopia av styrordet lagras på minnesadressen `DCCopy`.
OUTONE specificeras på följande sätt:

```
* Subrutin OUTONE.
* Ettställer en av bitarna  i kopian DCCopy
* och borrmaskinens styrord
*
* Vilken bit som ettställs bestäms av innehållet i
* D1-registret (0-7) vid anrop av subrutinen Ifall
* innehållet > 7 utförs ingenting.
*
* Anrop:    MOVE.B      #5,D1
*           JSR         OUTONE
* Bit nummer 5 i styrordet ettställs
*
* Indata:   D1-registret skall innehålla ett tal
* 0-7, som är numret på den bit i styrsignalordet
* som skall ettställas. Bitnumrering framgår av
* följande figur .
```

7	6	5	4	3	2	1	0

```
* Utdata:               Inga
* Registerpåverkan:     Ingen
* Anropade subrut:      Inga
```

Flödesdiagram över OUTONE

UPPGIFT 4.23:

Rita en flödesplan för subrutinen OUTONE.

Skriv sedan rutinen OUTONE i filen `SUB2.S68`.

Du testar rutinen i nästa uppgift.

SLUT PÅ UPPGIFT 4.23.

UPPGIFT 4.24:

Du kan testa din OUTONE på liknande sätt som OUTZERO. När du nu skall ettställa bitar är det lämpligt att nollställa DCCopy först.

Testa OUTONE och övertyga dig om att den fungerar enligt specifikationen.

SLUT PÅ UPPGIFT 4.24.

Flödesdiagram START

Nu är det dags att beskriva den första rutinen i COMMAND. Rutinen START startar borrmotorn så att denna roterar när tangentnummer noll har aktiverats på tangentbordet. START specificeras enligt:

```
* SUBRUTIN START.
* Subrutinen startar borrmotorn och väntar
* därefter i 500 ms före återhopp för att
* borrmaskinen skall uppnå rätt hastighet.
*
* Borrmotorn stoppas genom att subrutinen OUTZERO
* matar ut värdet "0" på bit 3 av utporten DCTRL.
* Endast bit 3 på utporten och DCCopy påverkas.
* Därefter fördröjs återhoppet
* 500 ms med subrutinen DELAY.
*
* Anrop:          JSR   START
* Indata:         Inga
* Utdata:         Inga
* Registerpåverkan:Ingen
* Anropade subrut:      OUTZERO, DELAY.
```

UPPGIFT 4.25:

Rita ett flödesdiagram för subrutinen START.
Skriv subrutinen i filen `SUB2.S68`

SLUT PÅ UPPGIFT 4.25.

Flödesdiagram över STOP

Subrutinen STOP stannar borrmaskinens motor.

```
* Subrutin STOP. Subrutinen stoppar borrmotorn.
* Borrmotorn stoppas genom att subrutinen OUTONE
* matar ut värdet "1" på bit 3 av utporten DCTRL.
* Endast bit 3 på utporten och DCCopy påverkas.
*
* Anrop:                JSR   STOP
* Indata:               Inga
* Utdata:               Inga
* Registerpåverkan:     Ingen
* Anropade subrut:      OUTONE
```

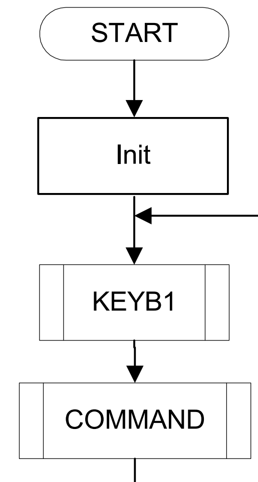
UPPGIFT 4.26:

Rita ett flödesdiagram för subrutinen STOP.
Skriv subrutinen i filen `SUB2.S68`

SLUT PÅ UPPGIFT 4.26.

För att testa rutinerna START och STOP återgår vi till vårt ursprungliga huvudprogram i filen `MAIN1.S68`.

Flödesdiagrammet för huvudprogrammet visas på nytt i marginalen. Vi repeterar också de operationer som borrmaskinen skall utföra beroende på vilka kommandon som ges.



tangent nr	Operation	subrutin
0	starta borrmotorn	START
1	stoppa borrmotorn	STOP
2	sänk borret	DOWN
3	höj borret	UP
4	rotera arbetsstycket medurs ett steg	STEP
5	borra ett hål	DRILL
6	stega arbetsstycket till referensposition	REFPO
7	borra hål längs cirkeln enligt mönster	AUTO

UPPGIFT 4.27:
Kopiera först filen `MAIN1.S68` till `MAIN2.S68`. Detta gör du enklast genom att öppna `MAIN1.S68` och spara denna med det nya filnamnet `MAIN2.S68`.

I slutet av `MAIN2.S68` lägger du nu till nya USE direktiv för att inkludera de nya subrutinerna.

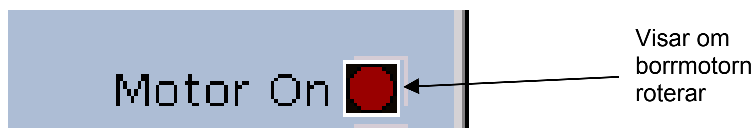
```
      USE   KeyML15.s68
      USE   DELAY.S68
      USE   SUB2.S68
```

Glöm inte att ändra i hopptabellen som utnyttjas av COMMAND. Du har nu skapat rutinerna START och STOP och då ska dessa namn anges i hopptabellen enligt:

```
*     Tabell med subrutinadresser
JUMPTAB
     FDB   START,STOP,SUB2,SUB3,SUB4,SUB5,SUB6,SUB7
```

Du kan nu kommentera bort dummy-rutinerna SUB0 och SUB1. Assemblera filen `MAIN2.S68`. Rätta eventuella fel och ladda till simulatorn för test.

Starta IO-simulatorn för borrmaskinen och för tangentbordet. Kontrollera att borrmotorn kan startas och stoppas från tangentbordet.



När du rättat eventuella fel i dina rutiner sparar du dina ändringar genom att göra save för dina filer.

SLUT PÅ UPPGIFT 4.27.

Flödesdiagram DOWN

Flödesdiagram UP

Rutinerna DOWN och UP.

```
* SUBRUTIN DOWN.
* Rutinen sänker borret genom att aktivera
* drivenheten för vridmagneten och uppdatera
* DCCopy.
*
* Anrop:                JSR DOWN
* Indata:               Inga
* Utdata:               Inga
* Registerpåverkan:     Ingen
* Anropade subrut:      OUTZERO



* SUBRUTIN UP.
* Rutinen höjer borret genom att deaktivera
* vridmagneten, uppdatera DCCopy och därefter
* vänta 500 ms före återhopp.
*
* Anrop:                JSR UP
* Indata:               Inga
* Utdata:               Inga
* Registerpåverkan:     Ingen
* Anropade subrut:      DELAY, OUTONE
```

UPPGIFT 4.28:

Skriv subrutinerna DOWN och UP enligt flödesdiagrammen du ritat i marginalen.

Spara rutinerna i filen `SUB3.S68`.

SLUT PÅ UPPGIFT 4.28.

För att testa dessa nya rutiner upprepar du förfarandet du gjorde när du testade START och STOP.

UPPGIFT 4.29:

Spara först `MAIN2.S68` om du inte gjort detta tidigare. Skapa nu en kopia `MAIN3.S68` och arbeta vidare med denna fil.

Ändra i hopptabellen så att SUB2 och SUB3 byts ut mot DOWN respektive UP.

Lägg slutligen till ett nytt USE-direktiv i Main-filen enligt

```
      USE   SUB3.S68
```

Assemblera, ladda och testa det nya programmet.

Du skall nu kunna starta, sänka borret och detektera en genomborrning av arbetsstycket, höja borret och slutligen stanna borrmotorn.

När du väl borrat ett hål i arbetsstycket så kan du välja 'New Disc' för att välja ett nytt, oborrat, arbetsstycke.

Verifiera att de implementerade funktionerna fungerar enligt specifikationerna.

SLUT PÅ UPPGIFT 4.29.

I specifikationen har vi också angett att alarmering skall kunna göras. Vi specificerar subrutinen ALARM.

Flödesdiagram ALARM

```
* SUBRUTIN ALARM.
* ger NTIMES st larmsignaler med längden 1 s och
* med 0,5 s mellanrum.
*
* Anrop:          MOVE.B      #NTIMES,D0
*                 JSR         ALARM
* Indata:         Antal larmsignaler, NTIMES,
*                 i D0-registret.
* Utdata:         Inga
* Registerpåverkan:Ingen
* Anropade subr:  DELAY, OUTZERO, OUTONE.
```

Subrutinen STEP, för att vrida arbetsstycket:

```
* SUBRUTIN STEP.
* Rutinen kontrollerar först i
* DCCopy att borret inte är sänkt och vrider sedan
* kretskortet ett steg medurs genom att mata ut
* riktningsinformation och stegpuls till steg-
* motorns drivenhet. Efter stegpulsen väntar
* rutinen 500 ms före återhopp så att
* arbetsstycket skall hinna vridas ett steg.
* Om borret är sänkt vid anrop av STEP utförs
* ingen stegning utan i stället ges tre
* larmsignaler på "summern" före återhopp.

*Anrop:           JSR STEP
*Indata:          Inga
*Utdata:          Inga
*Registerpåverkan:      Ingen
*Anropade subr:   ALARM, DELAY, OUTZERO, OUTONE
```

Kontrollen av om borret är sänkt görs genom att undersöka värdet av signalen "solenoid" (vridmagnet) i ordet DCCopy. Man kan inte använda givaren för "borr nere" vid denna kontroll eftersom en borrning kan pågå utan att borret har hunnit igenom arbetsstycket!

Flödesdiagram STEP

UPPGIFT 4.30:

Implementera rutinerna ALARM och STEP enligt specifikationerna ovan. Spara rutinerna i `SUB4.S68`.

Kopiera `MAIN3.S68` till `MAIN4.S68` och arbeta vidare med denna fil. Ändra så att anrop av ALARM och STEP sker på önskat sätt.

Testa dina nya borrmaskinfunktioner på samma sätt du gjort tidigare. Rätta eventuella fel och spara slutligen `MAIN4.S68` och `SUB4.S68`.

SLUT PÅ UPPGIFT 4.30.

För att ett hål skall kunna borras i en sammansatt operation behövs ett program som avgör om borret har arbetat sig igenom hela arbetsstycket innan borret höjs. Subrutinen DDTEST (drill down test) skall sköta detta.

Flödesdiagram DDTEST

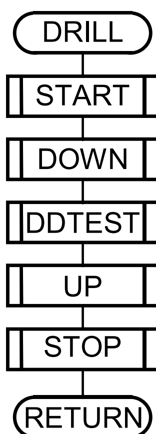
```
* SUBRUTIN DDTEST.
* Undersöker via en givare om borret nått sitt
* bottenläge. Återhopp sker ej förrän borret nått
* bottenläget. Om borret ej har nått bottenläget
* inom 3 sekunder ges två larmsignaler och
* återhopp sker.
* För att inte borrningen skall fördröjas onödigt
* länge ska sensorns värde läsas av två
* gånger per sekund.
*
* Anrop:          JSR DDTEST
* Indata:         Inga
* Utdata:         Inga
* Registerpåverkan:Ingen
* Anropade subr:  ALARM, DELAY.
```

UPPGIFT 4.31:

Implementera rutinen DDTEST i filen `SUB5.S68` enligt flödesdiagrammet du ritat i marginalen.

SLUT PÅ UPPGIFT 4.31.

Rutinen som i en sammansatt operation borrar *ett* hål i arbetsstycket betecknas DRILL och specificeras enligt flödesplanen i marginalen.



UPPGIFT 4.32:

Implementera rutinen DRILL. Placera denna i filen `SUB5.S68` tillsammans med DDTEST.

Kopiera filen `MAIN4.S68` till `MAIN5.S68` och fortsätt arbetet med denna fil.

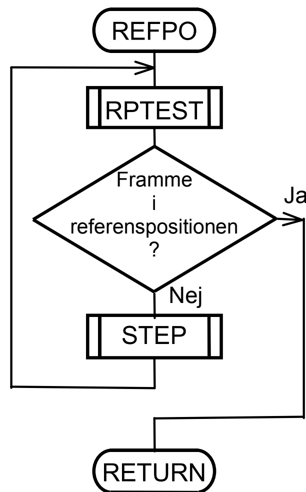
Ändra i `MAIN5.S68` så att anrop av DRILL utförs när tangent 5 aktiveras på tangentbordet.

Använd samma testförfarande som tidigare.

Rätta eventuella fel och spara slutligen `MAIN5.S68` och `SUB5.S68`.

SLUT PÅ UPPGIFT 4.32.

Arbetsstycket skall roteras till referenspositionen i subrutinen REFPO. Studera flödesdiagrammet. I REFPO ingår subrutinen RPTEST, som undersöker och lämnar information om arbetsstyckets läge. RPTEST specificeras enligt:




```
* SUBRUTIN RPTEST.
* Undersöker via läsgaffeln om arbetsstycket
* befinner sig i referenspositionen.
* Placerar det avlästa läsgaffelvärdet i
* C- flaggan.
*
* Anrop:          JSR RPTEST
* Indata:         Inga
* Utdata:         Arbetsstyckets position
* indikeras i C-flaggan med "1" för
* referenspositionen och "0" för övriga
* positioner.
* Registerpåverkan:    Ingen, speciellt sätts
* eller nollställs dock C-flaggan i SR.
* Anropade subr:  Ingen
```

Flödesdiagram över RPTEST

UPPGIFT 4.33:

Rita flödesdiagrammet över RPTEST. Implementera sedan rutinerna REFPO och RPTEST. Placera dessa i den nya filen `SUB6.S68`.

Kopiera `MAIN5.S68` till `MAIN6.S68` och ändra i denna så att anrop av REFPO utförs när tangent 6 aktiveras på tangentbordet.

Använd samma testförfarande som tidigare.

Rätta eventuella fel och spara slutligen `MAIN6.S68` och `SUB6.S68`.

SLUT PÅ UPPGIFT 4.33.

Flödesdiagram NSTEP

När arbetsstycket befinner sig i referenspositonen eller när ett hål har borrats i en position skall arbetsstycket roteras ett antal steg till positionen för nästa hål. Detta skall göras i subrutinen NSTEP som specificeras enligt

```
* SUBRUTIN NSTEP.
* Roterar arbetsstycket N steg medurs.
*
* Anrop:          JSR   NSTEP
* Indata:         Antalet steg n (0-255) finns
*                 i D0-registret.
* Utdata:               Inga
* Registerpåverkan:     Ingen
* Anropade subr:        STEP
```

UPPGIFT 4.34:

Rita först flödesdiagrammet för rutinen NSTEP. Implementera sedan rutinen i filen `SUB7.S68`. Testa rutinen på lämpligt sätt genom att exempelvis använda DIP-SWITCH'arna för att ge indata till subrutinen och studera hur arbetsstycket vrider sig.

SLUT PÅ UPPGIFT 4.34.

Vi har nu skapat praktiskt taget allt vi behöver för att uppfylla den ursprungliga specifikationen av vår "borrobot". Det är dags att ge sig i kast med den avslutande uppgiften...

Det färdiga programmet

Hålmönstret för ett arbetsstycke (ett varv) skall specificeras i en tabell, med basadressen PATTERN, såsom visas nedan. I denna tabell anges *antalet steg* till nästa hål. Avståndet från referenspositionen till första hålet är alltså 0 steg. Talet $FF används för att markera mönsteravslut.

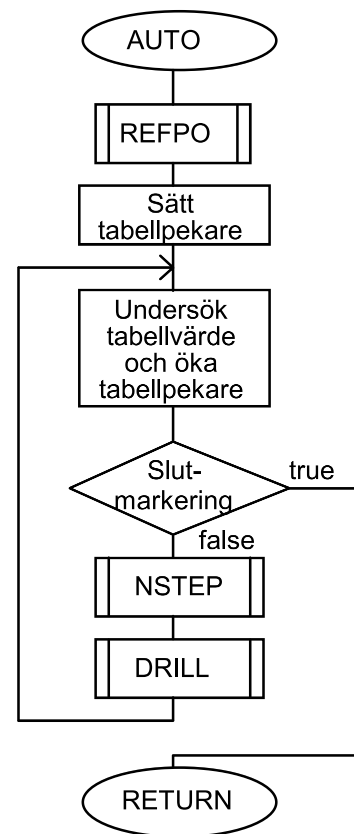
Hål nr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Antal steg	0	1	1	1	1	1	1	1	2	1	5	2	2	2	2	4	4	3	8	2	FF

En subrutin AUTO implementerar en automatisk borrmaskin. Rutinen borrar ett antal hål i arbetsstycket enligt den givna tabellen PATTERN ovan.

Se flödesdiagrammet i marginalen.

AUTO specificeras enligt:

```
* SUBRUTIN AUTO.
* Borrar ett antal hål i ett arbetstycket enligt
* en given tabell med start på adress PATTERN.
*
* Först roteras arbetsstycket till
* referenspositionen. Därefter läses ett nytt
* värde från PATTERN som anger antal steg till
* nästa hål som skall borras.
*
* Anrop:          JSR   AUTO
* Indata:         Adressen till PATTERN finns i
*                 register A0
* Utdata:         Inga
* Registerpåverkan:Ingen
* Anropade subr:  REFPO, NSTEP, DRILL
```



UPPGIFT 4.35:

Implementera rutinen AUTO och placera denna sist i filen `SUB7.S68` tillsammans med NSTEP.

Kopiera Main-filen till `MAIN7.S68` och ändra i denna så att anrop av AUTO utförs när tangent 7 aktiveras på tangentbordet.

Använd samma testförfarande som tidigare.

Rätta eventuella fel och spara slutligen `MAIN7.S68` och `SUB7.S68`.

SLUT PÅ UPPGIFT 4.35.

Avsnitt 4 Sammanfattning

I detta avsnitt har du sett hur en lite större applikation kan byggas upp och testas.

Avsnittet lägger stor tyngd på metodiken, dvs specifikationer och dokumentation. Du bör dra lärdom av detta och alltid tänka på hur du planerar din programkonstruktion så att du kan testa din kod, efter hand, i lagom portioner.

Avsnitt 5

Maskinnära programmering i C och assembler

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 MC68000-baserade mikrodatorer. Du får se exempel på hur kodgenerering bör utföras. Du får samtidigt en rad tips om vilka fällor du bör undvika då du programmerar maskinnära med en C-kompilator. Avslutningsvis erbjuds du en rad utmaningar, där du kan testa dina nyvynna kunskaper och kombinera dom för att lösa icke-triviala programmeringsproblrem

Målsättningar:

Då du arbetat i genom avsnittet ska du kunna skapa egna "stand-alone"-applikationer, i C och assembler med XCC32 (XCC68).
Kunna skapa nya programbibliotek.
Kunna underhålla och utöka befintliga programbibliotek.

Inledning

XCC32 är ett integrerat programutvecklingsverktyg för MC68340 -baserade mikrodatorer. *XCC68* är ett integrerat programutvecklingsverktyg för MC68000 - baserade mikrodatorer *XCC* har från början utvecklats som en s.k "korskompilator", dvs ett programutvecklingssystem där man använder en speciell värddator för att utveckla program till en annan maskin, den s.k måldatorn. *XCC* har också anpassats speciellt för utbildningsändamål och det är därför mycket enkelt att skapa applikationsprogram för exempelvis laborationsdatorn *MC68 (MD68k)*.

De primära syftena med *XCC* 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 MC68x00-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

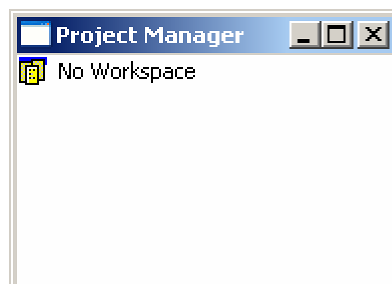
Projektets typ anges då du skapar det, typen kan därefter inte ändras. Du kan däremot lägga till nya källtextfiler efter hand, ändra en rad olika projektinställningar etc.

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:

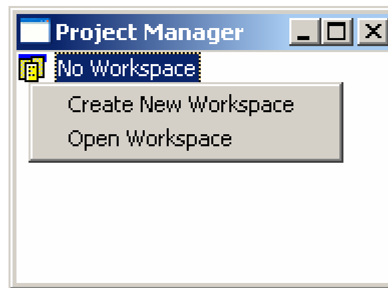
xcc\src	här skapar du alla källtextfiler.
xcc\ws	här skapas 'Workspace' och projekt

Starta XCC om du inte gjort det tidigare, det enda fönstret du ser är projekthanteraren:



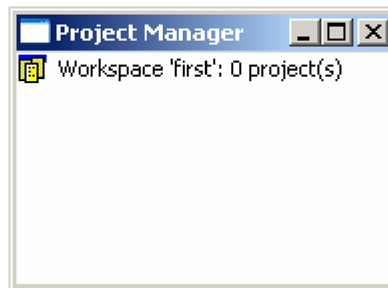
Projekthanteraren har ett fönster som alltid ligger ovanför andra fönster. Du kan gömma det genom att klicka på 'Close', visa det igen via menyvalet. 'Project | View Project Manager'

Högerklicka en gång på texten 'No Workspace', texten märks nu med blå bakgrund ...

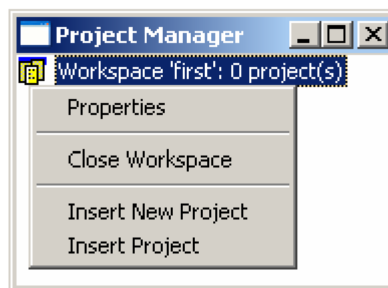


Härifrån kan du skapa ett nytt, eller öppna ett befintligt 'Workspace'.
Du kan göra samma saker via menyn:
'File | New Workspace' eller
'File | Open Workspace'

Skapa nu ett nytt workspace 'first' i biblioteket 'C:\xcc\ws'.
Observera hur projekthanteraren nu visar ett aktivt workspace:



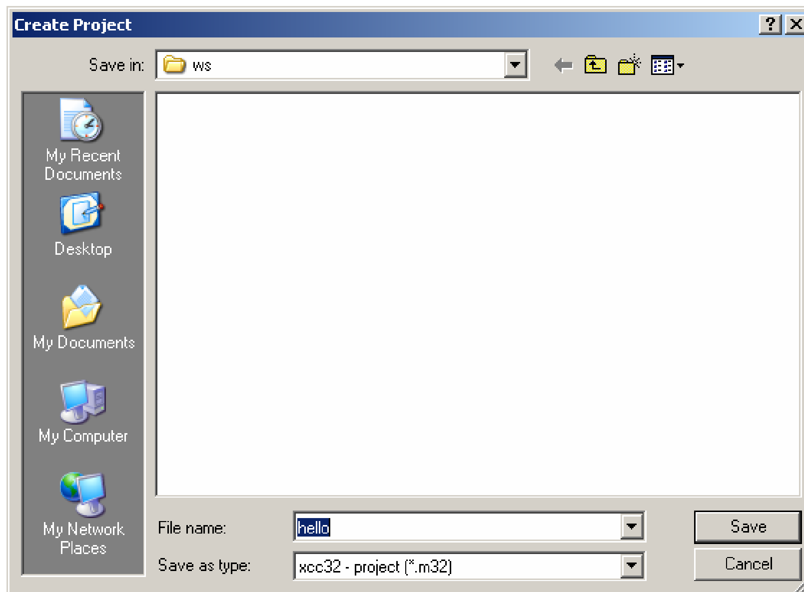
Högerklicka på texten som tidigare, följande popup mey visas:



Alternativ:

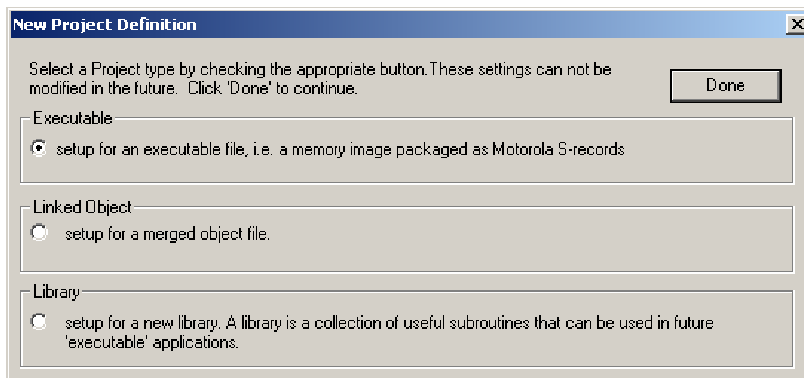
- Properties - visar information om filens egenskaper.
- Close Workspace - stänger aktivt workspace.
- Insert New Project - Skapa ett nytt projekt och inför detta i aktivt workspace.
- Insert Project - Inför projekt som skapats tidigare (eventuellt under ett annat workspace) i aktivt workspace.

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 ":



Projektfilen får automatiskt ändelsen ".m32" (*XCC32*) eller ".m68" (*XCC68*).

Klicka på "Save" , du får nu följande dialogruta:



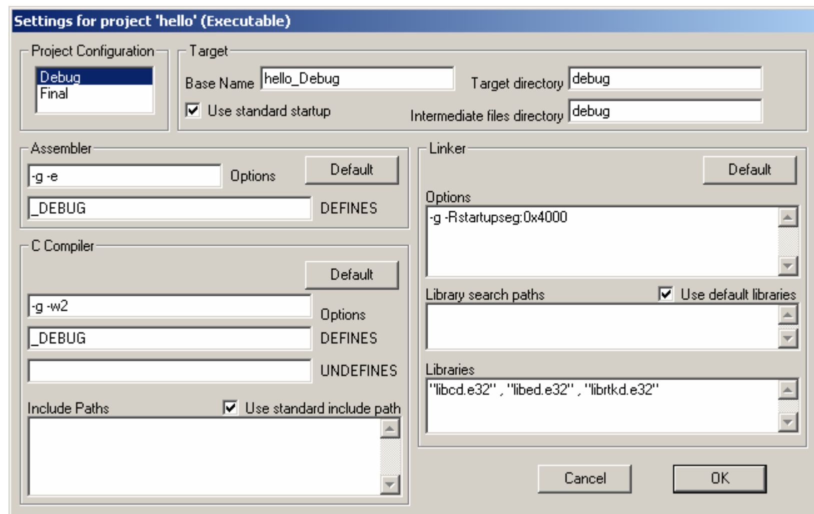
Vi vill nu skapa en exekverbar fil med debug-information enligt förslaget, klicka därför på 'Done'.

Nästa bild är 'Project Settings'-dialogen. Du kan öppna denna närsomhelst 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 vårt inledande exempel är det lämpligt att använda de föreslagna inställningarna.



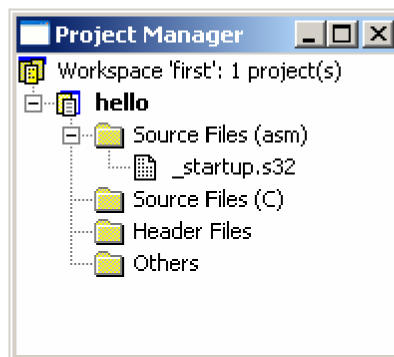
I vårt fall kommer således applikationen

'hello\debug\hello_Debug.s19'

så småningom att skapas.

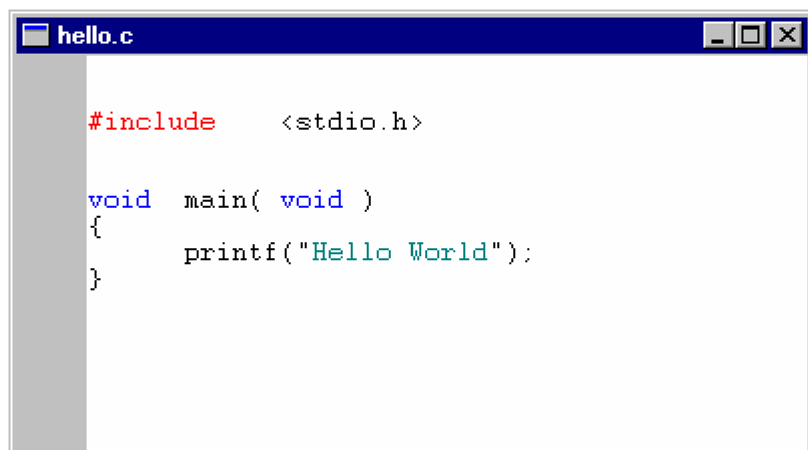
Klicka 'OK' för att stänga dialogen.

Observera hur Projekthanteraren uppdateras och aktiverar det nya projektet.



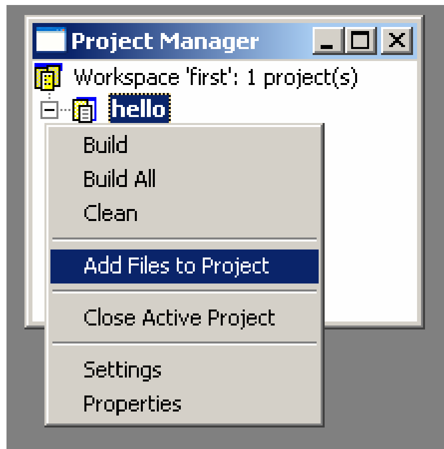
Projekthanteraren sorterar filerna med avseende på filnamnsändelser under separata flikar. Detta har ingen annan praktisk betydelse än att de ska vara mer överskådliga.

Välj nu, från menyn, File | New och skapa filen 'hello.c' i biblioteket xcc\src. En editor startas, skriv in följande program:



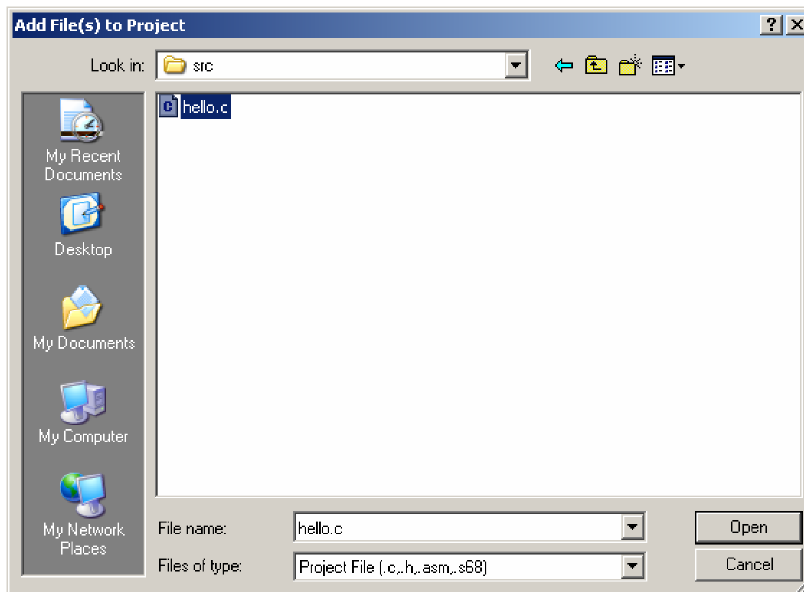
Välj File | Save för att spara den nya filen.

Märk det aktiva projektet och högerklicka, du får en ny pop-up-meny. Välj 'Add Files to Project'.



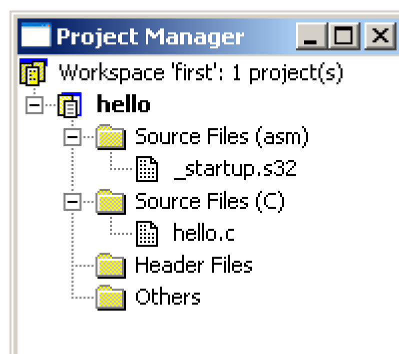
Tips:
Du kan använda 'multiple select' här om du vill lägga till fler filer från samma bibliotek samtidigt

Håll ned 'Ctrl'-tangenten samtidigt som du märker de filer du vill ska ingå i projektet.

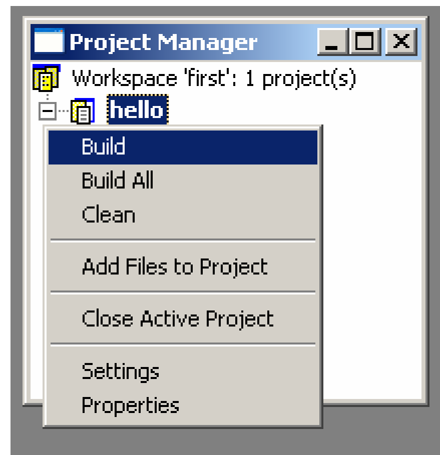


Välj den nya filen 'hello.c' och klicka på 'Open'.

Projekthanterarens fönster uppdateras....



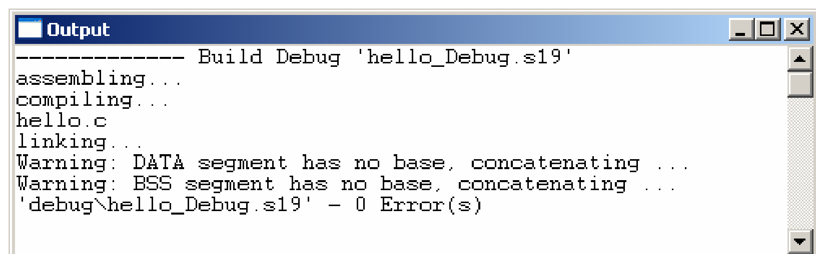
Du skapar nu den exekverbara filen 'hello_Debug.s19' genom att välja Build.



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:



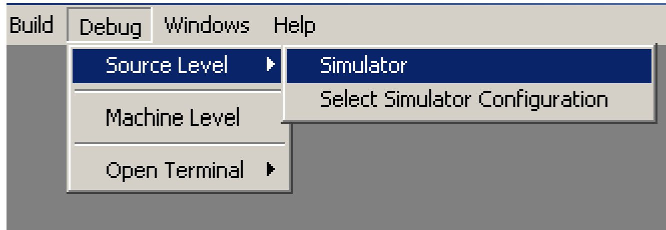
XCC kompilerar filen 'hello.c', länkar därefter samman modulen med nödvändiga biblioteksrutiner och skapar laddfilen 'hello_Debug.s19'.

Länkaren genererar här varningsutskrifter:
'DATA segment has no base....' detta betyder att segmenten tilldelats basadresser automatiskt. I vårt fall är detta inget problem så du kan bortse från varningarna.

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.

Välj 'Debug | Source Level | Simulator':

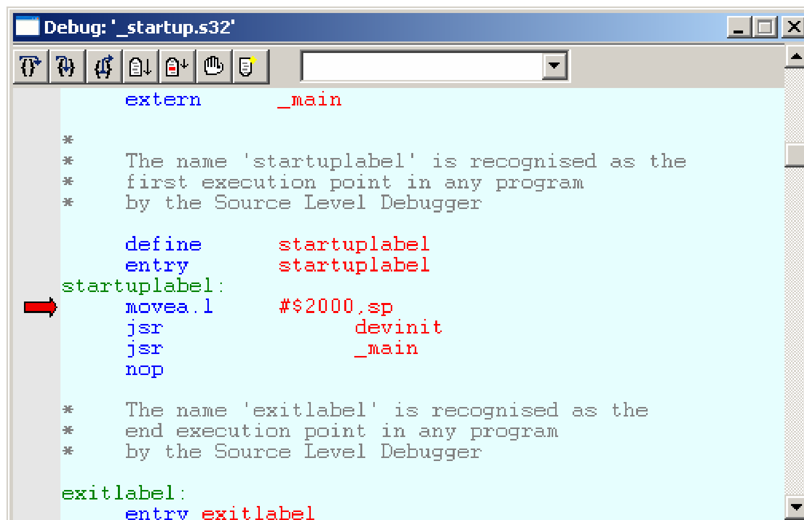


för att starta debuggern och den inbyggda simulatorn.

Välj därefter filen:
xcc\ws\hello\debug\hello_Debug.s19

Två nya fönster skapas nu, ett känner du igen sedan tidigare, det är simulator-fönstret. Simulatorn är den samma som du tidigare använt under ETERM.

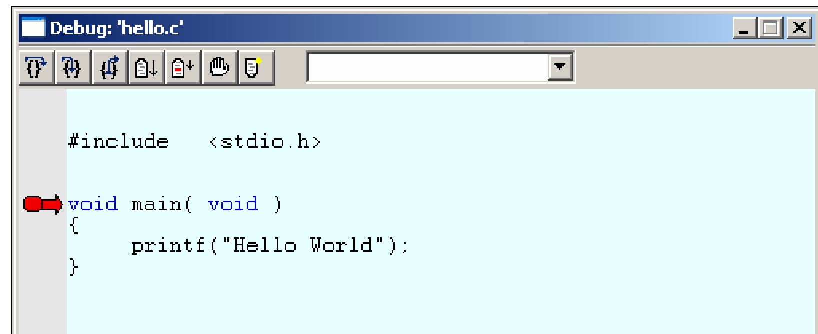
Det andra fönstret hör till själva debuggern...



Programmets första exekveringspunkt finns i filen '_startup.s32' som länkats in först av XCC. Vi ska inte nu fördjupa oss i själva uppstarten av programmet utan nöjer oss med att konstatera att debuggern dessutom satt en brytpunkt på adressen för symbolen '_main' (som är starten på vårt program...)

Välj därför 'Run' från verktygsfältet...

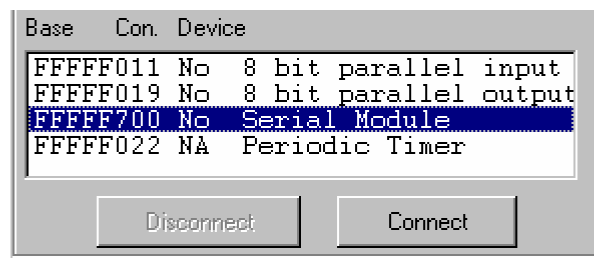
Debuggern startar programmet men det stannar omedelbart vid funktionen 'main'.



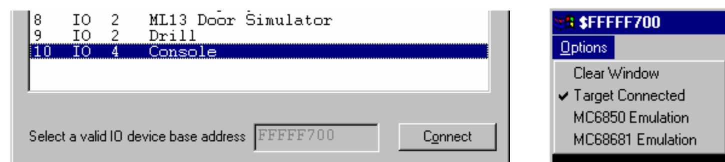
Innan vi startar programmet igen är det lämpligt att koppla ett konsoll-fönster till simulator. I detta konsollfönster förväntar vi oss sedan att utskriften 'Hello World' ska dyka upp.

Anslut ett konsollfönster
Välj 'Simulator Setup' från simulatorns verktygslist:

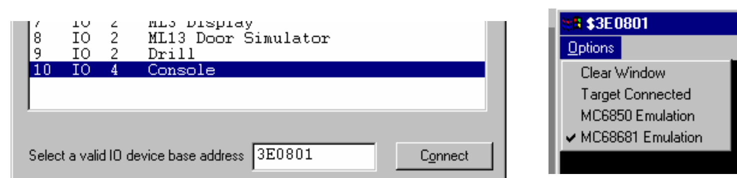
Om du använder MC68
Koppla 'Serial Module'...via knappen 'Connect



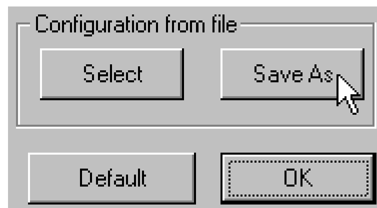
till IO-simulatorns 'Console'-enhet.
I konsollfönstret väljer du sedan Options | Target Connected.



Om du använder MD68k
Koppla IO-simulatorns 'Console'-enhet till adress 3E0801 (DUART basadress).
I konsollfönstret väljer du sedan Options | MC68681 Emulation.

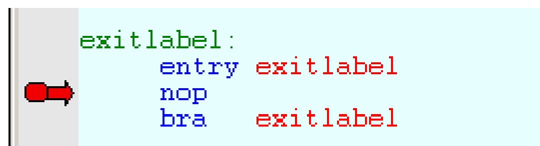


Tips: Spara inställningen under filnamnet 'console' innan du stänger 'Target Setup'.



UPPGIFT 5.1

Det är nu dags att testa exemplet. 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...

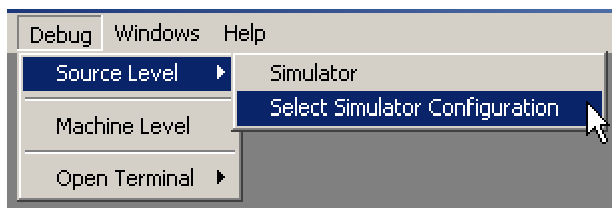
Om du inte ser texten ’Hello World’ här måste du gå tillbaka och i första hand kontrollera anslutningen av IO-simulatorn. Testa programmet och kontrollera funktionen.

SLUT PÅ UPPGIFT 5.1.

Stäng nu debugger'n. (Debug | Close).

För att spara tid kan du koppla en given simulatorkonfiguration till projektet. Du behöver då inte göra om kopplingen mellan IO-simulatorns konsoll och simulatorn varje gång du startar debuggern.

Välj från Debug-menyn...



Välj därefter filen 'console' som du sparade i föregående moment...

Nästa gång du startar debuggern för detta projekt kommer simulatorn i sin tur att göra kopplingarna automatiskt.

UPPGIFT 5.2
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", ........ );
```

Testa programmet och kontrollera funktionen.

SLUT PÅ UPPGIFT 5.2.

Debuggerns funktioner

Du har nu sett ett mycket enkelt inledande exempel på hur du kan använda XCC och färdiga programbibliotek för att snabbt skapa en komplett applikation. 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:

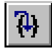
```
if( a < b )
        return 1;
```

kommer det att bli lättare att följa programflödet i debuggern.

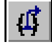
För assemblerkälltexter gäller att varje rad motsvarar en unik exekveringspunkt.

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).

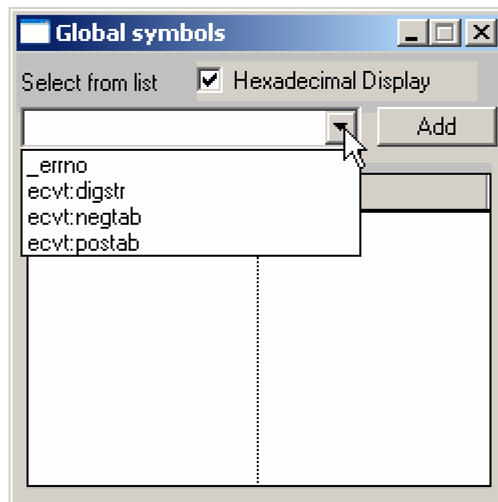
- **Breakpoints** - Öppna dialogruta för brytpunktstabellen. Används för att sätta, aktivera, deaktivera och ta bort brytpunkter.

- **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'. Alternativet 'Hexadecimal Display' används för att ställa in visningsformatet.

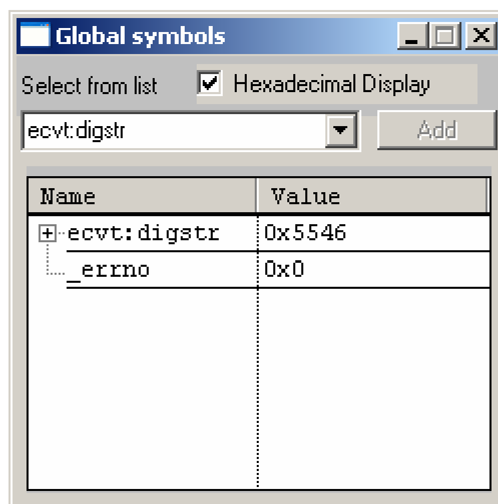
I denna bild visas variabler som deklarerats i standard C biblioteket. "_errno" är global med full synlighet, de övriga är "static"-deklarerade i modulen "ecvt" (ecvt.c).



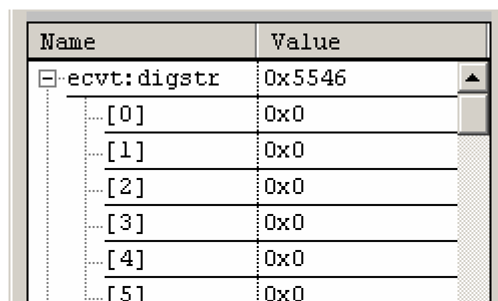
Prova funktionen genom att lägga till "_errno" respektive "ecvt:digstr"...

Variabeln "digstr" är en sträng med "characters", den startar på adress 0x5546 (du kan förstås inte ändra denna startadress..)
OBSERVERA: Denna adress kan vara annorlunda beroende på vilken version av programbiblioteket du använder.

Om du klickar på '+'-tecknet expanderas variabeln och samtliga komponenter visas.
OBS:
'Watch"-fönstren kan visa maximalt 200 element



Klicka på '+'-tecknet för att expandera en strängvariabel...



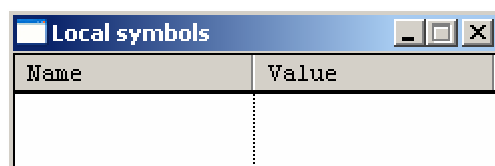
Du kan ändra variabelvärde, klicka i 'Value'-fältet för den variabel du vill påverka...

Name	Value
⊞ ecvt:digstr	0x5546
_errno	0x0

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.

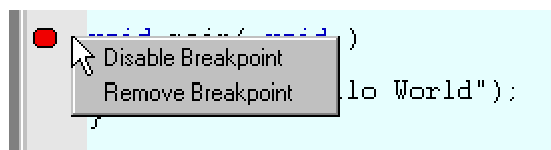
Brytpunkter

Brytpunkter hanteras dels med hjälp av den gråa listen i 'debugfönstret', dels med hjälp av en brytpunktstabell.

EXEMPEL
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.

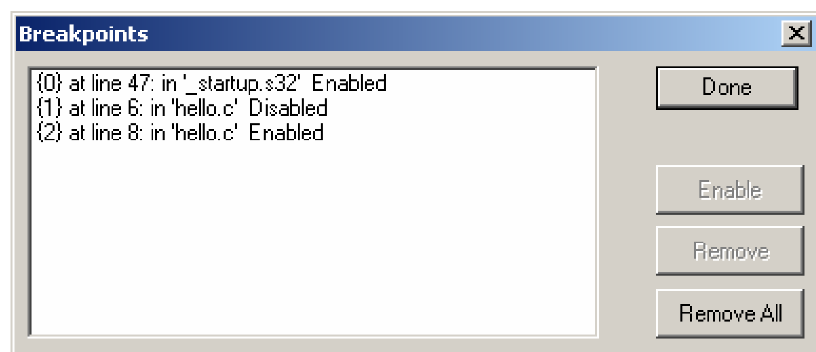
På motsvarande sätt kan du sätta ut en ny brypunkt genom att placera markören på en rad och högerklicka.

```
void main( void )
{
      printf("Hello World");
```

Insert Breakpoint

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.

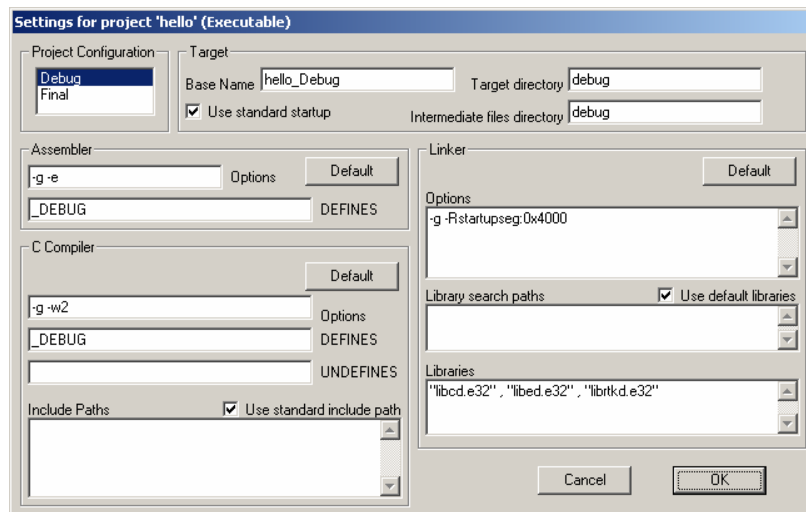
Slutligen, kan brytpunkter hanteras med hjälp av brytpunktstabellen.



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.



Raden 'Options' ska innehålla de flaggor som används av respektive verktyg (assembler, kompilator och länkare).

Du kan läsa mer om dessa flaggor i XCC's 'on-line-help'.

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, dessutom utför kompilatorn en rad olika åtgärder i syfte att förbättra ("optimera") den genererade koden.

Target

Här anges namnet på projektets resultatfil (Base Name). Till namnet läggs automatiskt rätt extension

- .S19 - för exekverbara filer
- .qld - för länkade objektfiler
- .e32 - för programbibliotek till *XCC32*.
- (.e68 - för programbibliotek till *XCC682*.

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.s32’ (’_startup.s68’)..

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 - Standardbibliotek 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/xcc32 (eller xcc68) - där INSTALL är det bibliotek där XCC32 (XCC68) installerats. Denna ruta ska vara ifylld om du vill att länkaren ska söka efter funktioner i XCC32's (XCC68’s) standardbibliotek.

Library search Paths - Här anger du bibliotek där länkaren ska söka efter Xcc's programbibliotek (.e32 eller .e68-filer) . Om en sökväg inleds med '\' innebär detta relativt biblioteket INSTALL, fullständig sökväg kan också anges.

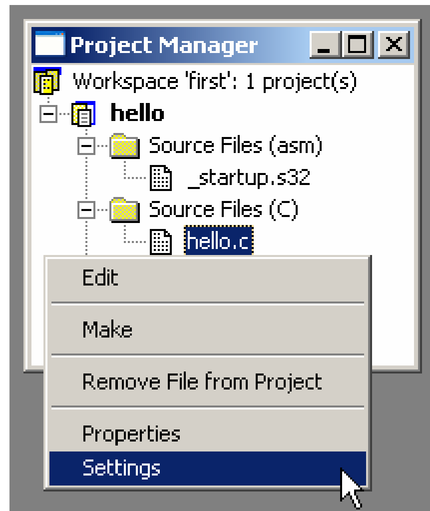
Libraries
Här anges de programbibliotek (endast namnen, ingen sökväg) du vill att länkaren ska söka i efter eventuella oupplösta symbolnamn. Detta är typiskt extra bibliotek som exempelvis standard c-biblioteket (libc), ”extended library” (libe), eller ”real time kernel library” (librtk) men det kan självfallet också 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.

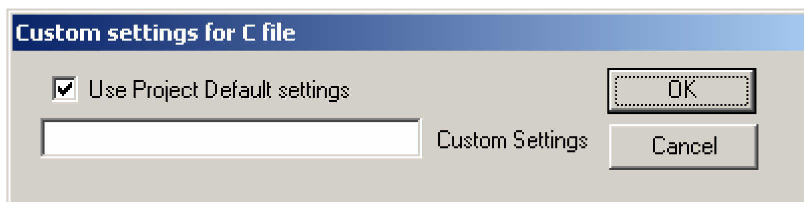
Individuella inställningar

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

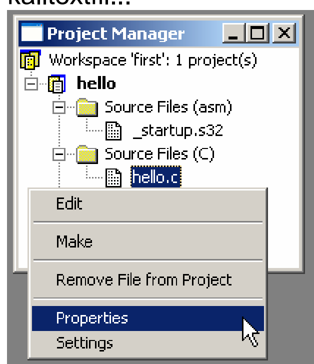
XCC består i själva verket av flera programdelar:

- Projekthanterare
- 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 .s68. För filnamn med ändelser .s68, .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 .o32/.o68) som skapas av kompilator/assembler och källtexten. Om projekthanteraren upptäcker att en fil *file.c* har sparats senare än den motsvarande *file.o32 (file.o68)* kommer filen att kompileras om automatiskt före länkningen. Även indirekta beroenden kan upptäckas av projekthanteraren. Om du exempelvis inkluderar filen *file.h* från *file.c* så tolkas detta som att *file.o32 (file.o68)* är beroende av såväl *file.c* som *file.h*. Om någon av dessa filer sparats med senare datum/tid än objektfilen kommer alltså projekthanteraren att kompilera *file.c* på nytt.

Högerklicka på en källtextfil...



.. och välj 'Properties' så visar XCC filens "egenskaper" och beroenden

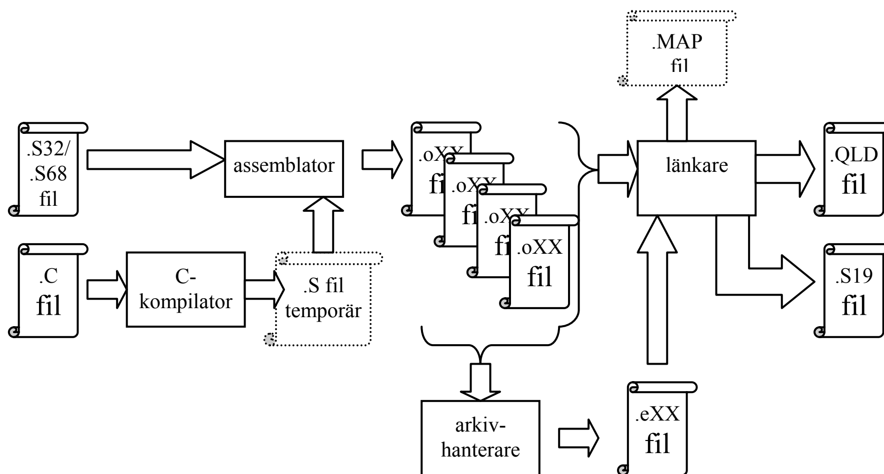
XCC kan automatiskt bestämma sådana beroenden. Du kan se dessa genom att välja 'Properties' från pop-up menyn för en märkt källtextfil.

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 .s32 (.s68) eller .asm dirigeras direkt till assemblatorn som assemblerar filen och skapar en objektmodul, .o32 (.o68). Objektmodulen innehåller relokerbar kod, dvs inga absolutadresser är ännu bestämda.

Länkaren sammanfogar flera objektmoduler till en (.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.



Då ett projekt öppnas kommer XCC att kontrollera att alla källtextfiler i projektet kan öppnas. Om så *inte* är fallet tyder detta på att källtexterna tagits bort eller flyttats på ett sådant sätt att de kan inte längre ingå i projektet. XCC meddelar detta och avlägsnar därefter filen från projektet.

Följande tabell ger en översikt av vilka filtyper och deras respektive filnamnstillägg som *XCC* arbetar med:

.w32 (.w68)	'Workspace' - innehåller information om de projekt som ingår.
.m32 (.m68)	Projektfil, innehåller beskrivning av projektet, dvs alla ingående källtexter, beroenden och växlar för att styra bygget av projektet.
.C	källtext ,C
.s32.s68, .asm	källtext, MC683xx/68xxx assembler
.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	utdata från länkning, innehåller absoluta adresser för samtliga globala symboler i laddmodulen. Du anger att denna fil ska skapas med en flagga till länkaren (-m FILNAMN.MAP)
.QLD	länkad objektfil
.e32 (.e68)	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

Använd ändelsen .s32 tillsammans med XCC32 och ändelsen .s68 tillsammans med XCC68.

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.s32
ml4.c
```

”Appstart.s32” 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...
           MOVEA.L    #$2000,SP
           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...
           MOVEA.L    #$2000,SP
           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...
            MOVEA.L     #$2000,SP
            JSR         _main
_exit:      NOP
            BRA         _exit
*
```

För att avsluta ett C-program krävs nu bara funktionsanropet:

```
      exit();
```

Det är nu bara några små detaljer kvar för att vår ”appstart” ska kunna användas.

Symbolen ”_main” är definierad i en annan källtext. Vi talar om detta för assemblatorn med direktivet:

```
      extern      _main
```

På motsvarande sätt gäller att symbolen ”_exit” definierats i ”appstart.s32”. Vi talar om för assemblatorn att denna symbol ska vara global, dvs refereras från en annan källtext, med direktivet:

```
      define      _exit
```

Vi måste också definiera ett *segment* för koden. Eftersom detta är så kallad ”startup-kod” namnger vi segmentet `startupseg`.

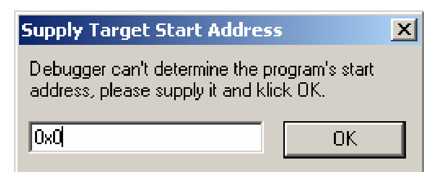
Vår slutliga ”appstart” blir...

```
            segment     startupseg

            define      _exit
            extern      _main
*     Här börjar exekveringen...
startuplabel:
            MOVEA.L     #$2000,SP
            JSR         _main
_exit:      NOP
            BRA         _exit
*
```

Observera också här hur vi namnger programmets första exekveringspunkt ’startuplabel’.

Detta beror på att XCC söker efter detta symbolnamn som programmets första startadress.

Om man *inte* definierar ’startuplabel’ kommer debuggern’n att fråga efter denna varje gång programmet startas.



UPPGIFT 5.3

Skapa en källtext "appstart.s32" enligt ovanstående anvisningar. Du kommer att få användning av den om en stund.

SLUT PÅ UPPGIFT 5.3.

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.

Portadressering (absolut adressering) i C

Portadresser i minnet kan enkelt adresseras. Man tillämpar bara 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)
```

`__XCC32` respektive `__XCC68` är *implicit* definierade macros. Du behöver alltså inte själv göra definitionen.

OBS: Det ska vara TVÅ 'underscore' (_) framför 'XCC..'

Vi kan därför definiera följande *macros:*

```
#ifdef     __XCC32
// Portadresser MC68
#define    ML4IN       0xFFFFF011
#define    ML4OUT      0xFFFFF019
#endif

#ifdef     __XCC68
// Portadresser MD68k
#define    ML4IN       0x3E0013
#define    ML4OUT      0x3E0011
#endif

#define    ML4READ     *((char *) ML4IN)
#define    ML4WRITE    *((char *) ML4OUT)
```

UPPGIFT 5.4
Skapa en "headerfil" med namnet `PORTADD.H` och med ovanstående macros/definitioner

SLUT PÅ UPPGIFT 5.4.

Följande sekvens visar hur vi nu 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:

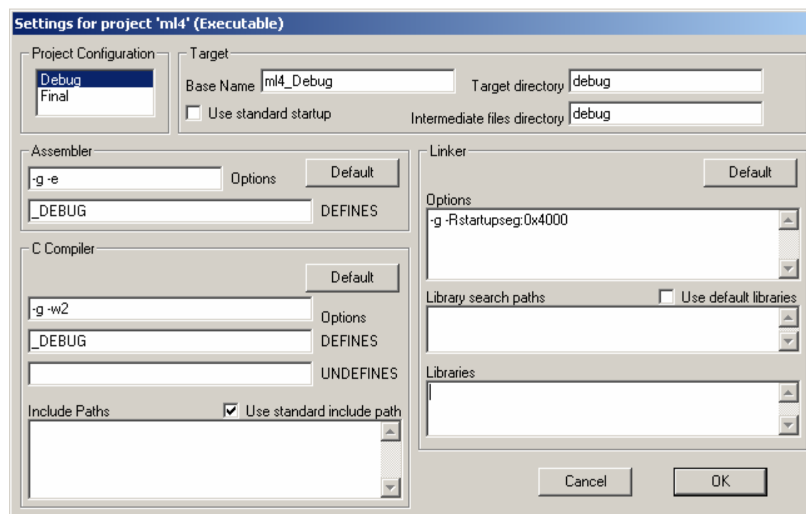
```
char  c;
c = ML4READ;
c = c >> 1;
ML4WRITE = c;
```

UPPGIFT 5.5
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 PORTADD.H för definitioner.

SLUT PÅ UPPGIFT 5.5.

UPPGIFT 5.6
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:



Observera speciellt att startadressen för programmet är angivet med ett direktiv till länkaren.

`-R startupseg:0x4000`

Vi har alltså försäkrat oss om att vårt programs första exekveringspunkt (instruktion) finns på adress 4000.
Notera också att vi avmarkerat alla standardalternativ för projektet (’Use standard startup’, ’Use default libraries’, Use standard include path’).

- Skapa den exekverbara filen, dvs "bygg" projektet med:
  Build | Build All
- Starta debugger'n
  Debug | Source Level | Simulator
- Från simulatorn, koppla ML4 Dip-Switch Input till adress adressen `ML4IN`.
- Koppla ML4 Parallel Output till adressen `ML4OUT`.
- Spara simulatorkonfigurationen under namnet 'ml4-simple'.

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 PÅ UPPGIFT 5.6.

UPPGIFT 5.7
Ändra nu testprogrammet så att ett vänsterskift utförs. Innan du startar debuggern, koppla din färdiga simulatorkonfiguration till projektet genom att välja:

Debug | Source Level | Select Simulator Configuration

Testa och kontrollera som tidigare.

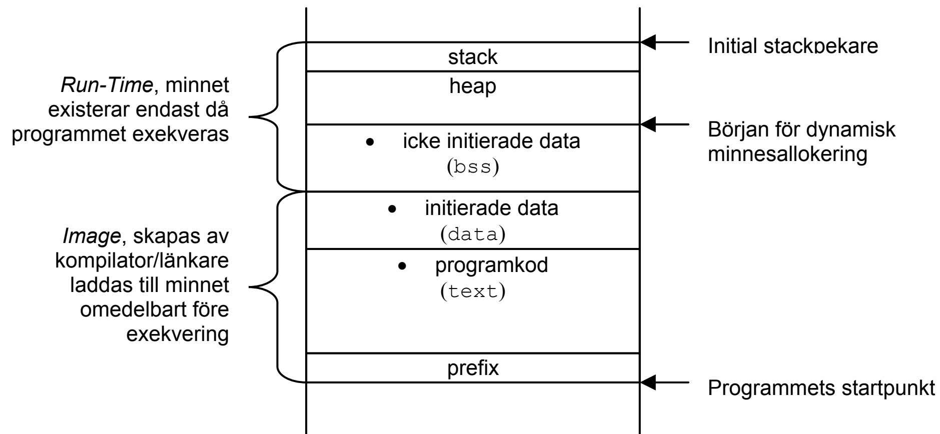
SLUT PÅ UPPGIFT 5.7.

XCC32/68 Kodgenerering

Under detta moment ska vi ge en detaljerad beskrivning av hur *XCC32/68* genererar kod för *MC68340/MC68000*.

Minnesdisposition

Programkod och data indelas i olika segment. Betrakta följande figur som beskriver hur minnesdispositionen för ett komplett program, under exekvering, kan se ut:



Figuren förstås bäst mot bakgrund av hur ett program översätts, sparas (eventuellt på en hårddisk), laddas till primärminnet och exekveras.

prefix

Prefixet, eller som det också kallas, *startup*, placeras alltid först. Detta är nödvändigt för att programmet ska ha en känd startpunkt. Den enklaste formen av prefix utgörs helt enkelt av ett subrutinanrop:

```
JSR   _main
```

Vi känner igen symbolen som namnet på det huvudprogram som måste finnas i varje C-program. Vi använder "underscore" framför symbolnamnet för att skilja C-funktionen "main" från (den översatta) assemblerfunktionen. Prefixet ingår i den så kallade *runtime-miljö* som finns tillsammans med kompilatorn. Prefixet länkas automatiskt först i varje C-program.

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 kallas detta segment för "text".

initierade data

Deklarationer som

```
      int a = 2;
      char array[] = {"Detta är en text"};
```

osv kan användas för 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.

EXEMPEL

Beroende på hur en textsträng deklareras kommer kompilatorn att placera den i olika segment:

Satsen

```
      printf("Denna text ...");
```

ger liknande 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 **text**-segmentet (just det..trots att det inte är programkod).

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 text-segmentet, i stället hamnar den i **data**-segmentet.

oinitierade data

Deklarationer som:

```
      int  a;
      char array[34];
```

osv, kan användas för 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 data placeras i ett särskilt segment. Av tradition kallas detta ofta *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 programmets dynamiska 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
- 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 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:

Om du har exekverat programmet tidigare *kan* data segmentet ha ändrats av ditt program. Då du startar om programmet har det därför inte samma begynnelsevillkor som då du exekverade det den första gången.

- Stackpekare måste initieras (eventuellt görs detta av debuggern)
- Det finns ingen dynamisk minneshantering tillgänglig.
- Programmet måste laddas från utvecklingssystem till måldator-system 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.

XCC, minnesallokering

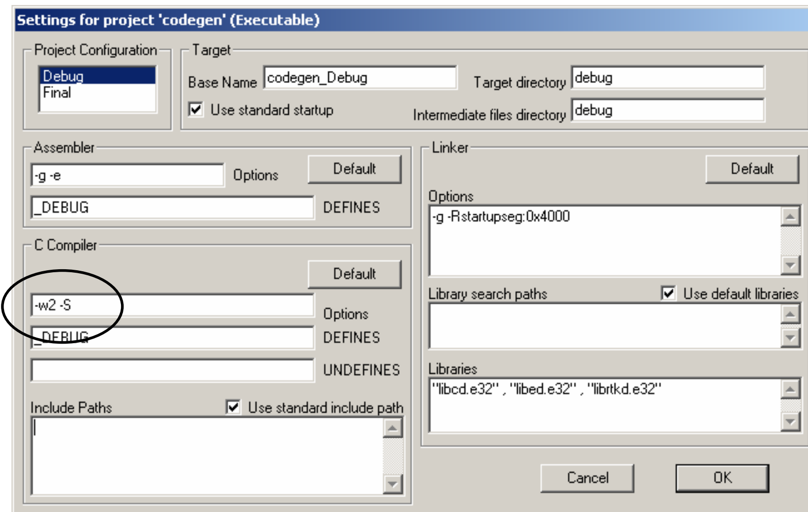
Vi ska nu se hur *XCC* genererar kod för några enkla variabeldeklarationer i **C**. Du kan själv enkelt upprepa detta genom att ange flaggan "-S" för källtexten i projektet. *XCC* kommer då att lämna de genererade assemblerfilerna som du kan granska med hjälp av texteditorn. Filnamnskonventionerna är enkla, om den kompilerade källtexten heter:

```
minfil.c
```

så kommer den genererade assemblerfilen att heta

```
minfil.S
```

UPPGIFT 5.8
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 inställningarna under *C-Compiler, Options,*

vi använder inte '-g'
men har lagt till '-S'

för övrigt kan du använda de inställningar som föreslås.

Du kan läsa om kompilatorns 'flaggor' ('växlar') i XCC's hjälpsystem.

SLUT PÅ UPPGIFT 5.8.

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 5.9
Skapa en ny källtext "globals.c" enligt ovanstående. Lägg filen till projektet. Välj
Make
för att kompilera filen.
Nu skapas bland annat filen "globals.S" i underbiblioteket "debug". Öppna denna fil...

SLUT PÅ UPPGIFT 5.9.

```
* CC32 - ...
* Code generation: MC683xx
      genfile     "C:\xcc\src\globals.c"
      bss
      align
      define      _shortint
_shortint:
      ds.b  2
      align
      define      _longint
_longint:
      ds.b  4
      align
      define      _justint
_justint:
      ds.b  4
      align
      define      _intvec
_intvec:
      ds.b  40
      align
      define      _komplex
_komplex:
      ds.b  10
```

Först genereras några inledande rader med text där *XCC* skriver ut aktuell version av kompilatorn (CC32 eller CC68, visas endast delvis).

"genfile"-direktivet anger den absoluta sökvägen till källtextfilen. Informationen används exempelvis av länkaren vid varnings- eller fel- meddelanden. Observera att den absoluta sökvägen kan se annorlunda ut i ditt eget exempel.

Direktivet

```
      bss     (samma sak som..)
      segment    bss
```

anger att påföljande direktiv kommer att reservera plats för icke initierade data.

Eftersom variablerna är deklarerade globala 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:

```
      define     _shortint
```

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."*

innebär att variabelnamnet får ett *globalt scope.* Observera också den inledande understrykningen. Alla globala namn, såväl funktioner som variabler förses med detta av kompilatorn. På så sätt undviks namnkonflikter mellan exempelvis rutiner som skrivs i assembler och placeras i programbibliotek, och funktioner som definierats i ett C-program. `shortint` är en variabel av typen `short`.

Efter label:n `_shortint` har *XCC* placerat direktivet

```
      ds.b 2
```

vilket alltså reserverar 2 bytes för variabeln.

På motsvarande sätt ser vi att variabeln `_longint` av typen `long` tilldelas 4 bytes i segmentet och att variabeln `_justint` av typen `int` tilldelas 4 bytes. För *XCC32 (XCC68)* gäller alltså att typen `long` och typen `int` är likvärdiga. Observera dock att detta inte gäller generellt för **C**. Datatypen `int` kan implementeras som 2-bytes i vissa system. Detta är exempelvis fallet för processorer som t.ex MC68HC11 eller MC68HC12. Konsekvensen av detta är att man bör tänka sig för vid deklaration av `int`-typer, dvs om man har anledning att tro att variabeln i fråga kan tänkas tilldelas större värden än vad som ryms i en `short` så ska den deklareras som `long`. Denna tumregel måste följas som man vill skapa *portabla* program, dvs **C**-källkod som kan kompileras om i olika miljöer, av olika maskiner, och ändå fungera som avsett.

ANSI definitionen av datatypen `long`

"Typen är synonym för: `long int, signed long` *och* `signed long int`*. Det är ett heltal med tecken som kan representeras med 32 bitar."*

Grundtanken med datatypen `int`, som ju egentligen kan betraktas som överflödig, är att den ska representera heltal som *enkelt* kan hanteras av den använda målprocessorns instruktionsrepertoir. Eftersom *MC68x00* direkt kan utföra instruktioner på 32-bitars tal är det lämpligt att jämställa `int` med `long`.

De ”grundläggande data-typerna” i C är:

```
char
short (int)
int
long (int)
pekare
float
double
long double
```

de tre sistnämnda avsedda för "flyttal" med olika precision.

Flyttalsoperationer utförs med hjälp av biblioteksrutiner. Du finner mer information om dessa i XCC's hjälpsystem. Vi kommer inte här att behandla flyttal.

Variabeln `intvec` är en vektor bestående av 10 komponenter, var och en av typen `int`. Följaktligen tilldelas variabeln `_intvec` 10*4 = 40 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` och en variabel `s3` av typen: *pekare till* `char`.

Vi ser att minnesbehovet bestämts till 10 bytes och vi vill nu kontrollera detta: En datatyp `int` kräver 4 bytes, en datatyp `char` kräver 1 byte, en pekartyp (ovidkommande vad den pekar på) kräver 4 bytes. Det totala behovet är alltså 9 bytes, vilket dock *inte* överensstämmer med direktivet (`ds.b 10`). Förklaringen ligger i att kompilatorn ser till att variabeln `s3` (pekare) hamnar på jämn adress (krävs av *MC68000*). Detta innebär, eftersom `s2` deklarerats som char (1 byte), att en utfyllnads-byte måste läggas in efter `s2`.

UPPGIFT 5.10

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 detta till en assemblerfil.

SLUT PÅ UPPGIFT 5.10.

UPPGIFT 5.11

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 detta till en assemblerfil.

SLUT PÅ UPPGIFT 5.11.

UPPGIFT 5.12

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;
```

Kontrollera din lösning genom att skapa en källtextfil i C och kompilera detta till en assemblerfil.

SLUT PÅ UPPGIFT 5.12.

Synlighet (Scope)

En global variabel är ”synlig” dvs kan refereras från alla delar av programmet oavsett vilken källtextfil den programdelen definierats.

För att kompilatorn, vid kompileringstillfället ska veta att en refererad symbol är deklarerad i en annan modul måste man ange detta med en extern-deklaration.

EXEMPEL

```
extern     int   foo;
main()
{
      ...
      foo =1;
      ...
}
```

Extern-deklarationer i C ger normalt *inte* upphov till speciella deklarationer i assemblerfilen. I stället assembleras filen under antagandet att alla icke-definierade symboler är externa. Det är först vid länkningen man upptäcker om något variabelnamn (symbol) inte är deklarerat.

Om man vill ange att en variabel ska vara osynlig utanför den källtext 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 med internt genererade symbolnamn som är unika inom källtexten

UPPGIFT 5.13

Kompilera följande deklarationer till assembler och studera assemblerfilen. Vilken skillnad upptäcker du?

```
int              a;
static     int   b;
```

SLUT PÅ UPPGIFT 5.13.

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 5.14
Betrakta 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 till 'unsigned int och kompilera på nytt till assembler, vilken villkorlig branch-instruktion används denna gång.

SLUT PÅ UPPGIFT 5.14.

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.

UPPGIFT 5.15
Kompilera följande program till assembler och studera assemblerfilen.

```
int  la;
short int  sa,sb;

void main()
{
     sa = la;
     sa = (short int) la;
}
```

Implicit typkonvertering. Denna bestäms av att typen på *resultatet* (sa) är short int

Explicit typkonvertering, dvs angiven i källtexten

Upptäcker du någon skillnad mellan tilldelningssatserna?

Ändra nu kompilatorns varningsnivå genom att ta bort flaggan "-w2" (Project | Settings, *C-Compiler, Options*) – kompilera på nytt. Vilka varningsmeddelanden får du?

SLUT PÅ UPPGIFT 5.15.

UPPGIFT 5.16

Kompilera följande program till assembler och studera assemblerfilen.

```
unsigned int la;
unsigned short int sb,sc;
void   main()
{
  la = (sb<<2)-sc;
  sb = la & sc;
}
```

Studera assemblerfilen, fyll i följande tabell med instruktionsföljder för respektive satser

la = (sb<<2)-sc;
sb = la & sc;

SLUT PÅ UPPGIFT 5.16.

UPPGIFT 5.17

Kompilera följande program till assembler och studera assemblerfilen. Upptäcker du någon skillnad i de respektive tilldelningarna.?

```
int     li;
unsigned short int      ui;
signed short int        si;
void   main()
{
  li = ui;
  li = si;
}
```

SLUT PÅ UPPGIFT 5.17.

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. Programmet kompileras och kod genereras (se marginalen).

```
main()
{
        short  shortint;
        long          longint;
        int           justint;

        struct {
              int           s1;
              char          s2;
              char          *s3;
        } typen;
}
```

```
_main:
      link  a6,#-20

* Källtext, utelämnad
      unlk  a6
      rts
```

Vi har här klippt bort introduktion och inledande assemblerdirektiv. Därefter inleds det vi känner igen som assemblerkod för MC68000, nämligen ett lägesnamn, i det här fallet `_main` som vi deklarerade som funktion. Direkt efter lägesnamnet följer instruktionen

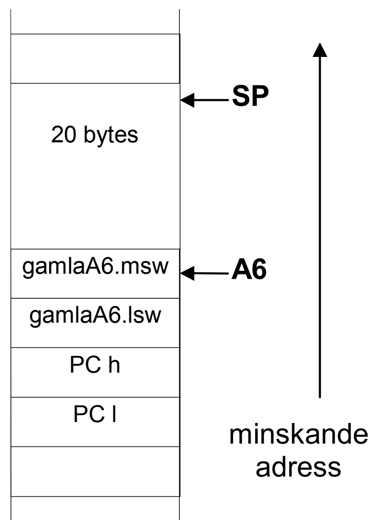
```
        link        a6,#-20
```

dvs:

- **A6** placeras på stacken
- stackpekaren kopieras till **A6**
- stackpekaren minskas med 20 bytes.

Följaktligen har vi:

- sparat gamla innehållet i **A6**.
- placerat en ny pekare i **A6**
- reserverat 20 bytes på stacken



Det totala minnesbehovet för de deklarerade variablerna är alltså 20 bytes vilket vi enkelt kan kontrollera genom att jämföra med föregående exempel.

Variablerna refereras därefter genom att ange en offset relativt till adressregistret **A6**. Denna offset är alltså *alltid negativ* eftersom stacken "växer nedåt" i minnet (se figur och tänk på att minskande adress visas uppåt) och stackpekaren efter instruktionen således har ett mindre värde än innehållet i **A6**.

Vid utträde ur funktionen återställs stacken med instruktionen:

```
unlk        a6
```

dvs:

- **A6** kopieras till stackpekaren, 20 bytes deallokeras
- **A6** återställs från stacktopppen

Såväl **A6** som **SP** (stacken) är därefter återställda till de värden de hade före `link`-instruktionen. Överst på stacken ligger nu returadressen för anropet till rutinen.

Observera att det sätt *XCC* använder register **A6** på innebär att detta register inte får modifieras av någon annan kod, exempelvis assemblerkod som länkas med programmet. En vanlig benämning på ett register som används på detta sätt är *frame-pointer*. Den del av stacken som pekas ut av **A6** kallas *aktiveringspost*.

Kompilatorn måste självfallet hålla ordning på var någonstans i aktiveringsposten respektive lokal variabel är placerad. Detta har ingenting att göra med `link` instruktionen som bara skapar aktiveringsposten.

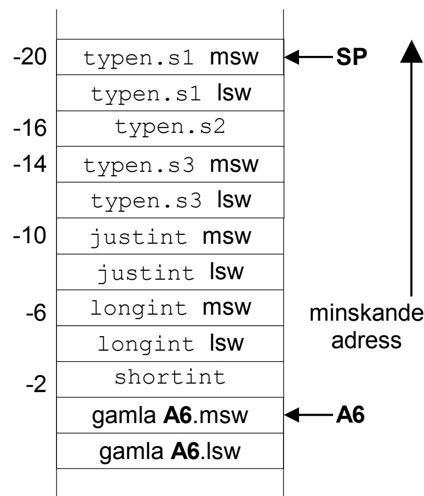
Lägg nu till följande tilldelningar i filen "locals.c" och observera hur de lokala variablerna refereras:

```
main()
{
      ....
      shortint = 1;
      longint = 2;
      justint = 3;
      typen.s1 = 4;
      typen.s2 = 5;
      typen.s3 = (char *) 6;
}
```

Följande assemblerkod genereras av kompilatorn: Anm: I assemblerkälltexten finner du också de källtextrader som assemblerkoden genereras från. Dessa inleds med kommentar och radnummer. Radnumren kan skilja i ditt eget exempel.

```
      text
      align
      define      _main
_main:
      link  a6,#-60
* 0020 |    shortint = 1;
      move.w      #1,(-2,a6)
* 0021 |    longint = 2;
      move.l      #2,(-6,a6)
* 0022 |    justint = 3;
      move.l      #3,(-10,a6)
* 0023 |    typen.s1 = 4;
      move.l      #4,(-20,a6)
* 0024 |    typen.s2 = 5;
      move.b      #5,(-16,a6)
* 0025 |    typen.s3 = (char *) 6;
      move.l      #6,(-14,a6)
_1:
      unlk  a6
      rts
```

*Figuren visar hur aktiveringsposten skapats på stacken av link, dvs var **A6** respektive **SP** pekar. Av figuren framgår också hur de lokala variablerna förhåller sig till **A6**.*



UPPGIFT 5.18

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.

SLUT PÅ UPPGIFT 5.18.

UPPGIFT 5.19
Antag att en funktion deklarerats på följande sätt:

```
void main( void )
{
     char *cp1,*cp2;

     cp1 = cp2;
     cp2 = &cp1[1];
}
```

Beskriv hur tilldelningarna översätts till assemblerkod av *XCC*. Kontrollera ditt svar genom att kompilera till assemblerkod.

SLUT PÅ UPPGIFT 5.19.

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:

```
int     a,b;
main()
{
        callfunc( a,b );
}
```

XCC genererar följande kod:

```
        text
        bss
        align
        define          _a
_a:
        ds.b            4
        align
        define          _b
_b:
        ds.b            4
        text
        align
        define          _main
_main:
        link            a6,#0
        move.l          (_b).l,-(a7)
        move.l          (_a).l,-(a7)
        jsr             (_callfunc).l
        addq.l          #8,a7
_1:
        unlk            a6
        rts
```

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 8 bytes till stackpekaren, dvs återställer denna

Vid `jsr` läggs återhoppsadressen (4 bytes) på stacken och i subrutinen `_callfunc` kan vi enkelt bestämma den offset från stackpekaren som gäller för de överförda parametrarna. Denna offset kombineras med den offset som genereras av eventuella lokala variabler, det vill säga: referens till parametrar kan sker via **A6** som *positiv* offset, eftersom de placerats på stacken före subrutinanrop och allokering för lokala variabler.

Vi exemplifierar detta genom att låta kompilatorn generera kod för "dummy"-funktionen `callfunc`.

```
callfunc( x , y )
{
      x = 1;
      y = 2;
}
```

XCC genererar följande kod:

```
      text
      align
      define        _callfunc
_callfunc:
      link          a6,#0
      move.l        #1,(8,a6)
      move.l        #2,(12,a6)
      unlk          a6
      rts
```

Observera hur parametrarna refereras via **A6**.

Om vi hade skrivit rutinen i assemblerspråk hade vi antagligen refererat parametrarna direkt via stackpekaren. Eftersom vi inte har några lokala variabler är ju egentligen `link/unlink` konstruktionen onödig och koden skulle då (kortare) ha skrivits enligt föjande:

```
callfunc:
      move.l        #1,(4,sp)
      move.l        #2,(8,sp)
      rts
```

För funktioner som returnerar något värde gäller att detta värde alltid finns i register D0 efter funktionsanropet. Detta gäller oberoende av datatyp.

UPPGIFT 5.20

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 PÅ UPPGIFT 5.20.

UPPGIFT 5.21
Översätt följande funktionsanrop till assemblerkod. Register D0 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 PÅ UPPGIFT 5.21.

UPPGIFT 5.22
Översätt följande funktionsanrop till assemblerkod. Ange också hur parametrar refereras i den anropade funktionen. Register D0 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.

parameter...	*refereras med offset...*
`low`	
`high`	

SLUT PÅ UPPGIFT 5.22.

UPPGIFT 5.23

Översätt följande funktionsanrop till assemblerkod. Ange också hur parametrar refereras i den anropade funktionen. Register D0 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). Detta callas också *call by reference* till skillnad från när parametern utgör ett *värde* vilket kallas *call by value.*

```
size = scalar( pvec , PVECSIZE );
```

där följande deklaration har gjorts:

```
#define    PVECSIZE   512
int        pvec[PVECSIZE];
int        size;
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

parameter...	*refereras med offset...*
PVECSIZE	
pvec	

SLUT PÅ UPPGIFT 5.23.

UPPGIFT 5.24

Översätt följande funktion till assemblerkod. Register D0 används för returvärde.

```
int   max( int x , int y )
{
            if( x >= y ) return x;
            return y;
}
```

Kontrollera ditt svar genom att kompilera till assemblerkod.

SLUT PÅ UPPGIFT 5.24.

Seriekommunikation via DUART

Vi ska nu ge ett mera fullständigt exempel på hur vi kombinerar källtexter skrivna i assembler respektive C. För exemplet använder vi seriekommunikationsrutiner för DUART-kretsarna hos *MC68* respektive *MD68k*.

Dina uppgifter blir därefter att redigera dessa källtexter, kompilera och testa funktionen. Du bör utföra detta för både *MC68* och *MD68k*.

Vi börjar med att beskriva C-källtexten som utformats så att alla rutiner (i assemblerkälltexten) används, helt enkelt ett testprogram...

```
/*
  serialtest.c
  För att testa enkla in- ut-matningsrutiner
  via serieterminalinterface ('konsoll')
*/

/* Prototyper */
void    ser_init( void );
void    outchar( int );
int     tstchar( void );

void    main( void )
{
  int   c;
  ser_init();

  while( 1 )
  {
        c = tstchar();
        if( c )
              outchar( c );
        if( c == 'e')
              break;
  }
}
```

Testprogrammet får väl anses vara tämligen självdokumenterande, efter initiering av DUART (`ser_init()`) faller exekveringen in i en slinga som kontinuerligt testar om det kommit in något tecken till DUART-en (`tstchar()`) och i så fall skriver ut tecknet (kallas i bland "eko) via `outchar( tecken)`. Om tecknet är 'e' terminerar programmet.

Vi måste alltså skriva dessa tre rutiner i assemblerkod. För *MC68* får vi exempelvis följande lösning:

```
*
*    serial32.s68
*    rutiner för DUART-seriekommunikation (MC68)
*
DUART EQU   $FFFFF700

            TEXT
            DEFINE     _ser_init
_ser_init:
*    normalt sett krävs ingen initiering av MC68 DUART
*    vid användning under DB68 eller simulator
*    rutinen finns dock med av kompatibilitetsskäl
*
*    C-interface:
*    void ser_init( void );
*
            RTS

*
*    OUTCHAR    rutin
*
*    C-interface:
*    void _outchar( int );
*
     DEFINE     _outchar
_outchar:
            MOVE.L     (4,SP),D0       parameter från stack
out:
            BTST.B     #2,(DUART+$11).L vänta tills TR
ledigt
            BEQ.S      out
            MOVE.B     D0,(DUART+$13).L skicka tecken
            RTS

*    TSTCHAR    rutin
*       om tecken finns, returnera det
*       returnera 0 annars
*
*    C-interface:
*    int  tstchar( void );
*
            DEFINE     _tstchar
_tstchar:
            BTST.B     #0,(DUART+$11).L  finns tecken i
DUART?
            BEQ.S      exit_testchar
            MOVE.B     (DUART+$13).L,D0  returnera tecken
            ANDI.L     #$7f,D0           endast ASCII
            RTS
exit_testchar:
            CLR.L      D0                    returnera 0
            RTS
```

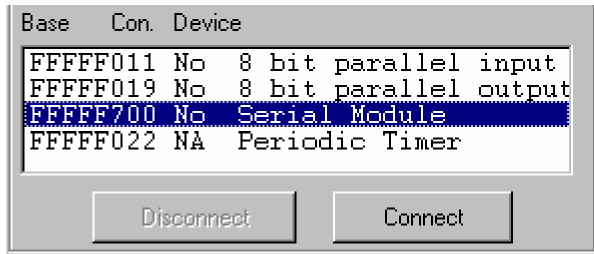
På motsvarande sätt måste vi tillhandahålla samma rutiner för *MD68k*.

```
*
* serial68.s68
* rutiner för DUART-seriekommunikation MD68k
*
DUART       EQU   $3E0801

       TEXT
       DEFINE     _ser_init
_ser_init:
* initiera MD68k duart
*
* C-interface:
* void ser_init( void );
*
       MOVE.B     #$13,(DUART).L
       MOVE.B     #$7,(DUART).L
       MOVE.B     #$BB,(DUART+2).L
       MOVE.B     #$5,(DUART+4).L
       RTS

*
* OUTCHAR    routine
*
* C-interface:
* void _outchar( int );
*
       DEFINE     _outchar
_outchar:
       MOVE.L     (4,sp),D0       parameter från stack
out:
       BTST.B     #2,(DUART+$2).L vänta tills TR ledigt
       BEQ.S      out
       MOVE.B     D0,(DUART+$6).L skicka tecken
       RTS

* TSTCHAR    rutin
*      om tecken finns, returnera det
*      returnera 0 annars
*
* C-interface:
* int  tstchar( void );
*
       DEFINE     _tstchar
_tstchar:
       BTST.B     #0,(DUART+$2).L
       BEQ.S      exit_testchar
       MOVE.B     (DUART+$6).l,D0
       ANDI.L     #$7f,D0
       RTS
exit_testchar:
       CLR.L      D0
       RTS
```

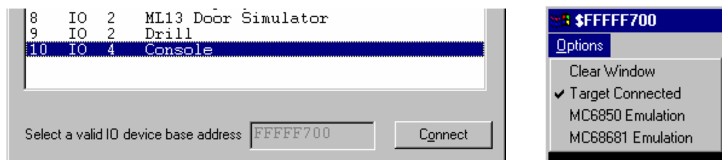
Du kan så småningom testa programmen med IO-simulatorns konsoll-fönster.

Då du testar för MC68

Koppla 'Serial Module'...via knappen 'Connect

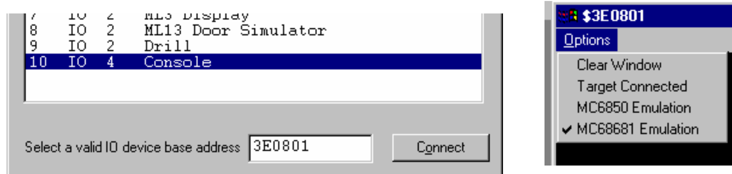


till IO-simulatorns 'Console'-enhet.
I konsollfönstret väljer du sedan Options | Target Connected.



Då du testar för MD68k

Koppla IO-simulatorns 'Console'-enhet till adress 3E0801 (DUART basadress).
I konsollfönstret väljer du sedan Options | MC68681 Emulation.



UPPGIFT 5.25

Redigera nu tre nya filer, `SERIALTEST.C`, `SERIAL32.S68` och `SERIAL68.S68` enligt de givna exemplen. Skapa två nya projekt SERTEST32 respektive SERTEST68.

Kompilera och testa programmet.

SLUT PÅ UPPGIFT 5.25.

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 programutecklingsmiljöer 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 utvecklingsmiljöns assemblator. Man måste dessuton 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. I följande exempel illustreras hur vi kan använda *dbg32*’s (*db68*’s) inbyggda rutin för utmatning av tecken genom att ”bädda in” assemblerinstruktioner i C-koden:

```
void  outchar ( char c )
{
      asm(" MOVE.B      (11,A6),D0");
      asm(" TRAP        #14");
      asm(" DC.W        1");
}
```

Den slutgiltiga assemblerkoden ser ut på följande sätt:

```
      text
      align
      define       outchar
 outchar:
      link  a6,#0

      MOVE.B      (11,A6),D0

      TRAP  #14

      DC.W  1
      unlk  a6
      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).

Maskinnära programmering med XCC

Vi har nu klarat av det mesta av konstigheterna som innefattas av en modern utvecklingsmiljö och det är alltså dags för mer konkreta uppgifter med vars hjälp du kan kontrollera dina kunskaper. Vi kommer att göra detta genom att återkoppla till alla tidigare avsnitt i denna arbetsbok.

Vi kommer i detta moment succesivt att bygga upp ett programbibliotek med drivrutiner för några *ML*-kort. Vi visar, med exempel hur drivrutiner för *ML4* kan utformas. Därefter specificeras drivrutiner för tangentbord (*ML2*) och displaykort (*ML3*) . Din uppgift blir att konstruera och testa dessa drivrutiner. Då detta är klart kommer vi att sätta samman alltihop i ett nytt programbibliotek ”`mllib.e32 (.e68)`”. Vi visar hur detta nya programbibliotek kan underhållas, dvs hur man tar bort, lägger till eller uppdaterar moduler i biblioteket. Du ska därefter modifiera dina testprogram så att de använder biblioteket.

Drivrutiner för ML4

Vi måste börja med att specificera drivrutinerna på ett entydigt sätt, dvs tala om exakt vad dom utför, eventuella parametrar och returvärden. Vi har här tre rutiner att specificera:

```
ML4_DipSwitch
ML4_Output
ML4_7Seg
```

Vi börjar med ML4_DipSwitch. Sedan tidigare vet vi att detta är en enhet som vi kan läsa en byte ifrån. En lämplig deklaration blir därför:

```
unsigned char   ML4DipSwitch( void );
```

dvs en funktion som returnerar en byte. Genom att dessutom ange denna `unsigned` försäkrar vi oss om att inga eventuella typkonverteringar kommer att ändra på ett värde från funktionen.

Vi använder definitionsfilen `PORTADD.H` för adressdefinitionerna.

Med detta blir nu de båda första funktionerna triviala. ”Skalet” för våra drivrutiner får följande utseende:

```
/*
      ml4drvr.c
      Drivrutiner för ML4
*/

#include   "portadd.h"


unsigned char   ML4_DipSwitch( void )
{
      //
}


void ML4_Output( unsigned char c )
{
      //
}

void ML4_7Seg( unsigned char c )
{
      //
}
```

Det kan nu vara lämpligt att komplettera `PORTADD.H` med prototyp-deklarationer av dessa rutiner. På så sätt underlättar vi användningen av dessa i andra applikationer.

```
/*
portadd.h

Definitioner och adresser för ML4
Maskintyp väljs med hjälp av XCC's implicita
macrodefinitioner
*/
#ifndef     I_PORTADD_H
// Denna test gör att filen inte inkluderas flera
// gånger
#define     I_PORTADD_H

#ifdef      __XCC32
// Portadresser MC68
#define     ML4IN       0xFFFFF011
#define     ML4OUT      0xFFFFF019
#endif

#ifdef      __XCC68
// Portadresser MD68k
#define     ML4IN       0x3E0013
#define     ML4OUT      0x3E0011
#endif

#define     ML4READ     *((char *) ML4IN)
#define     ML4WRITE    *((char *) ML4OUT)

/* Prototypdeklarationer ML4 drivrutiner */

unsigned char     ML4_DipSwitch( void );
void              ML4_Output( unsigned char c );
void              ML4_7Seg( unsigned char c );

#endif      /* I_PORTADD_H */
```

Vi visar nu de fullständiga rutinerna.

```
#include   "portadd.h"

unsigned char   ML4_DipSwitch( void )
{
     return( ML4READ );
}

void ML4_Output( unsigned char c )
{
     ML4WRITE = c;
}

/* definiera vektor med segmentskoder...*/
/* se även 'Avsnitt 1' där vi använde dessa
     första gången */

static     char seg_codes[]={
0x77,0x22,0x5B,0x6B,0x2E,0x6D,0x7D,0x23
0x7F,0x6F,0x3F,0x7C,0x55,0x7A,0x5D,0x1D
};


void ML4_7Seg( unsigned char c )
{
     if( c > 0xF )
     /* Ignorera otillåtna tal */
          return;

     /* Översätt segmentkod och skriv ut */
     ML4WRITE = seg_codes[ c ];
}
```

UPPGIFT 5.26
Redigera två källtexter `PORTADD.H` och `ML4DRVR.C` enligt det givna exemplet.

Redigera också en källtext med följande testprogram:

```
# include  "PORTADD.H"

void main( void )
{
     unsigned char c;
     while( 1 )
     {
          c = ML4_DipSwitch();
          ML4_Output( c );
          ML4_7Seg( c );
     }
}
```

OBS: Du måste testa en utmatningsrutin i taget eftersom de använder samma port. Kommentera bort en utav dom under test.
Kompilera drivrutinerna (Build), koppla IO-simulatorn till de olika enheterna och testa drivrutinerna.

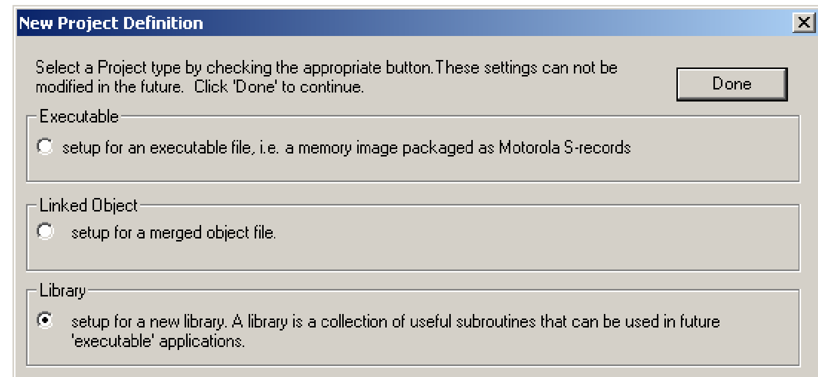
SLUT PÅ UPPGIFT 5.26.

Då drivrutinerna för *ML4* fungerar som de ska är det dags att placera dessa i ett nytt programbibliotek.

Att skapa programbibliotek

Skapa ett nytt projekt, namnge det nya projektet 'mllib'. Du ska sedan ändra '(Target) Base Name' till "MLLIB". Biblioteket kommer då att få namnet ”`MLLIB.E32`” ("`MLLIB.E68`" om du använder *MD68k*).

Som projektets typ anger du “Library”



Lägg till filen `ML4DRVR.C` till projektet.

Välj sedan ”Build All” från menyn för att skapa biblioteket.

Anm:
Vi har gjort detta på enklaste sätt. För en fullständig installation bör vi skapa såväl en 'Debug'-variant som en 'Final'-version.

Att installera programbibliotek

För att det ska vara enkelt att använda det nya programbiblioteket krävs att filerna `MLLIB.Exx` respektive `PORTADD.H` placeras i bibliotek där de enkelt kan hittas. Konventionerna hos *XCC* är att själva programfilen (`.Exx`) placeras i ett LIB-bibliotek medan header-filer placeras i ett INCLUDE-bibliotek. Dessa standard-kataloger finns under den katalog där *XCC* är installerad enligt:

```
{Installationskatalog}\include
{Installationskatalog}\lib\xccXX
```

För att installera det nya programbiblioteket måste du alltså

- Kopiera `PORTADD.H` till {Installationskatalog}\ include
- Kopiera programbiblioteket (.E68) till `{Installationskatalog}\lib\xcc32 (xcc68)`

UPPGIFT 5.27
Installera ditt nya programbibliotek.

SLUT PÅ UPPGIFT 5.27.

Att använda programbibliotek

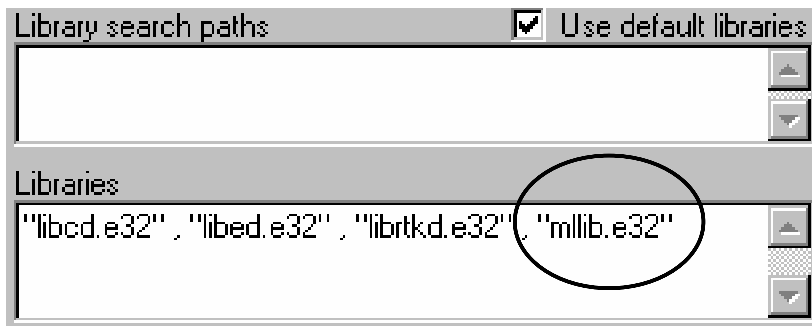
Vi ska nu se hur enkelt det blir att använda det nya programbiblioteket. Enklast demonstrerar vi detta med en uppgift.

UPPGIFT 5.28

Skapa ett nytt projekt, `MLLIBTEST`. Redigera en källtext med följande testprogram (jfr tidigare testprogram) och lägg filen till projektet. Observera att "include" direktivet ändrats en aning, i stället för `"PORTADD.H"` använder vi `<PORTADD.H>`, detta instruerar *XCC* att söka efter filen i "standard-include" katalogen.

```
# include  <PORTADD.H>

void main( void )
{
      unsigned char c;
      while( 1 )
      {
            c = ML4_DipSwitch();
            ML4_7Seg( c );
      }
}
```

I projektinställningarna måste du utöka listan med bibliotek.



Observera att ändelsen är **.e68** om du använder *XCC68*

Skapa nu testapplikationen med biblioteksrutiner. Testa och kontrollera att den fungerar som den ska.

SLUT PÅ UPPGIFT 5.28.

Du ska nu, på egen hand bygga ut det påbörjade programbiblioteket.

UPPGIFT 5.29

Aktivera projektet `MLLIB`.

Konstruera drivrutiner för *ML15* enligt följande specifikationer. Redigera dessa i en ny fil `ML15DRVR.C.` Lägg filen till projektet Placera även nya adress- och prototypdeklarationer i filen `PORTADD.H`.

```
char ML15_Keyboard( void );
```

Undersök tangentbordet. Om ingen tangent är nedtryckt skall 0xFF returneras, annars tangentkoden. Rutinen väntar tills nedtryckt tangent släpps upp igen.

respektive:

```
void ML15_Display( char *);
```

Rutinen matar ut en sträng om 6 st hexadecimala siffror till Display-modulen och tänder denna. Parametern är en pekare till en sträng som innehåller de hexadecimala siffrorna (mest signifikanta siffra först). Strängen förutsätts vara riktig, dvs inte innehålla värden större än 0xF.

Det är lämpligt att du först testar i ett lokalt bibliotek. Då du ser att dina drivrutiner fungerar som de ska installerar du programbibliotek och header-fil i standardkataloger.

SLUT PÅ UPPGIFT 5.29.

Alternativ till programbibliotek

Om du exempelvis vill samla ett antal vanliga rutiner och alltid sammanfoga dessa med en applikation kan du skapa en sammanlänkad objektfil (`.QLD`). Till skillnad mot ett programbibliotek så genomsöks inte en sådan objektfil efter saknade symboler och länkningen går därför betydligt snabbare.

Avsnitt 5
Sammanfattning

I avsnittet behandlas en typisk utvecklingsmiljö för programutveckling i C och assembler.

Avsnittet ger en introduktion till hur en C-kompilator kan generera assemblerkod.

Tyngdpunkten i avsnittet ligger vid att lära ut förmågan att självständigt planera och konstruera olika typer av program.

Appendix

Appendix A: IO-adresser för laborationskort

		Laborationssystem, alla adresser anges på hexadecimal form							
		MC11		**MC12**		**MC68**		**MD68k**	
		Adress	Not	Adress	Not	Adress	Not	Adress	Not
Lab kort	Register/Port								
ML4									
	Out	1400		0400		FFFFF019		3E0011	c,d
	In	1600		0600		FFFFF011		3E0013	c,d
ML5			e		e		e		e
	Out	0C00	a	0C00		8C000		418001	d
	In	0C01	a	0C01		8C001		418003	d
	Out	0C02	a	0C02		8C002		418005	d
	Out	0C03	a	0C03		8C003		418007	d
ML12				ej möjlig				ej möjlig	
	CAN	0E00	b	-		88000		-	
	In/Out	0F00	b	-		88100		-	
ML13									
	Ctrl/Status	0B00	a	0B00		8B000		416001	d
	IRQ Ctrl/Status	0B01	a	0B01		8B001		416003	d
ML15									
	Kbd Data	09C0	a	09C0		89C00		413801	d
	Kbd Status	09C1	a	09C1		89C01		413803	d
	Led Mode	09C2	a	09C2		89C02		413805	d
	Led Ctrl/Data	09C3	a	09C3		89C03		413807	d
ML16									
	Ctrl/Status	0A00	a	0A00		8A000		414001	d
	Data	0A01	a	0A01		8A001		414003	d
ML19									
	Status	09C0	a	09C0		89C00		413801	d
	Kvittera ev 1	09C2	a	09C2		89C02		413805	d
	Kvittera ev 2	09C3	a	09C3		89C03		413807	d

Noter

a) Laborationskortet måste anslutas via ovansidans kontakter på MC11.

b) Laborationskortet måste anslutas via undersidans kontakter på MC11. ML12 måste byglas för anslutning till MC11 (se även ML12 handbok).

c) Förutsätter att PIT har initierats av monitor/debugger.

d) Via expansionskort *ML18*

e) De här angivna adresserna för ML5 gäller PAL-revision 2.

Appendix B: 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 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 gäller detta MC68000/MC68010 men även MC68008, och varianter av denna som dock har ett mindre adressrum.

S3 Typen innehåller kod/data som kan laddas av system med 32-bitars adressrum. Detta gäller exempelvis MC68020/MC68030/MC68040 samt flera mikrocontrollers i 683X0-serien.

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 C: ASCII representation

American Standard Code for Interchange of Information.

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 D: Exceptionvektorer

Vektor nr	Adress(* (hex)	Funktion
0	000	Initial stackpekare
1	004	Initial programräknare
2	008	Bus Error (ex: referens till adress där minne/periferi ej finns)
3	00C	Adress Error (ex: referens till udda adress med *word* operand)
4	010	Illegal instruktion (icke-definierad operationskod)
5	014	Division med 0
6	018	Trap vektor för instruktionen CHK
7	01C	Trap-vektor för instruktionen TRAPV
8	020	Privilege Violation, försök att utföra *supervisor*-instruktion i *user mode*
9	024	Trace, en-instruktions exekvering
10	028	Line 1010, reserverad operationskod
11	02C	Line 1111, reserverad operationskod
12	030	Reserverad för framtida bruk
13	034	Reserverad för framtida bruk
14	038	Reserverad för framtida bruk
15	03C	Avbrott från enhet som ej tillhandahållit avbrottsnummer
16-23	040-05F	Reserverade vektorer
24	060	Icke-identifierat avbrott
25	064	Autovektor avbrottsnivå 1
26	068	Autovektor avbrottsnivå 2
27	06C	Autovektor avbrottsnivå 3
28	070	Autovektor avbrottsnivå 4
29	074	Autovektor avbrottsnivå 5
30	078	Autovektor avbrottsnivå 6
31	07C	Autovektor avbrottsnivå 7
32-47	080-0BF	Trap vektor för instruktionen TRAP #<vektor_nummer>
48-63	0C0-0FF	Reserverade vektorer
64-255	100-3FF	Användardefinierade vektorer

Anmärkning:

Med 'Adress' avses här 'offset', dvs hos *MC68*, som har en CPU32-processor läggs detta värde till innehållet i VBR (Vector Base Register). VBR ges värdet 0 vid RESET.

För *MD68k*, som har en MC68000 processor, finns inte detta register och 'Adress' anger därför alltid den absoluta adressen.

Appendix D: 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 *e.s68*, skapas en objektfil *e.o68*, modulnamnet blir då *e.o68*. Objekttypen blir *O68*.

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 BSS,DATA,TEXT,DPB och ABS men andra namn är tillåtna, de får innehålla max 16 tecken (a-z, 0-9).

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

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`. Beroende på om referensen sker från TEXT- eller DATA- segmentet kan ett relokeringskommando 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.