CHALMERS Institutionen för data och informationsteknik
LP4-2014/15, RoJ/RS/JS

Programmering av inbyggda system

Laborationer för D, Z och GU (EDA481, EDA486 och DIT152)

Laborationsserien omfattar totalt fem laborationsmoment som utförs i tur och ordning.

Det förutsätts att du inför varje laboration har genomfört ***omfattande*** laborationsförberedande hemuppgifter. Vilka dessa uppgifter är, framgår av detta PM. Du ska vara beredd att visa upp och redogöra för dina förberedda lösningar inför laborationstillfället. ***Bristfälliga förberedelser kan medföra att du avvisas från bokad laborationstid.***

Underskrifterna på detta försättsblad är ***ditt*** kvitto på att du är godkänd på respektive laborationsmoment. Spara det, för säkerhets skull, tills slutbetyg på kursen rapporterats. **Börja med att skriva ditt namn och personnummer med bläck.**

_____ _____

Personnummer Namn (textat)

Följande tabell fylls i av laborationshandledare efter godkänd laboration.

Laboration	Godkännande av laboration	
	Datum	Laborationshandledares underskrift
1		
2		
3		
4		
5		

Godkännande – hel laborationsserie:

_____ _____

Datum Laborationshandledares underskrift

Översikt av laborationsserien

Under laboration 1 bekantar du dig med laborationssystemet, konstruerar några enkla assemblerrutiner som du kommer att ha glädje av senare. Du påbörjar konstruktion av ett programpaket för övervakning och styrning av en liten borrmaskin.

Under laboration 2 färdigställer du ett komplett programpaket, i assemblerspråk, för borrmaskinen.

Under laboration 3 införs en enkel räknarkrets, programpaketet byggs ut att omfatta även avbrottshantering, du implementerar en enklare så kallad *tidsdelnings applikation* (”time sharing”).

Laboration 4 omfattar grundläggande C-programmering, du använder fält och pekare, du får också tillfälle att praktisera test, felsökning och ”avlusning”.

Under laboration 5 övas framför allt dina kunskaper i maskinnära C-programmering men även hur du kombinerar programdelar skrivna såväl i C som i assemblerspråk.

Kompletterande material

För laborationernas genomförande behöver du, utöver kurslitteraturen, program och diverse tekniska beskrivningar. Du finner detta på kursens *resurssida*.

Vid laborationerna används följande program:

ETERM 6 för MC12
XCC12 för MC12
CodeLite med GCC

Programmen bör du också installera på egna datorer för att underlätta ditt förberedelsearbete.

Följande beskrivningar finns tillgängliga på elektronisk form:

- Laborationskort ML4, multifunktionskort för MC12
- Laborationskort ML5, gränssnitt mot tangentbord/display (ML23), för MC12
- Laborationskort ML15, gränssnitt mot tangentbord/display (ML23), för MC12
- Laborationskort ML23, tangentbord/display, för MC12
- DBG12 monitor/debugger för MC12

Laboration nr 1 behandlar

MC12 med monitor/debugger DBG12
Tangentbord som inmatningsenhet
Sifferindikator som utmatningsenhet
Borrmaskin som styrobjekt

Laborationens huvudsakliga syften är att du ska:

- bekanta dig med laborationssystemet, i första hand:
  mikrodator *MC12* och dess monitor/debugger *DBG12*
  tangentbord *ML3*
  sifferindikator *ML2*
  borrmaskin
- förbereda de följande laborationerna genom att konstruera och testa grundläggande programrutiner för
  inmatning från *tangentbord*,
  utmatning till *sifferindikator*
  styrning av *borrmaskinens* funktion

För laborationen finns speciellt följande fil tillgänglig via kursens resurssida: `laddfil.s19`

Följande uppgifter ur *Arbetsbok för MC12* ska vara utförda innan laborationen påbörjas. Du ska på begäran av laborationshandledare redovisa dessa.

Uppg.	39-43	71-81
Sign.		

Utöver uppgifterna i arbetsboken ska följande hemuppgifter (ges längre fram i detta PM) vara utförda innan laborationen påbörjas:

Hem-Uppgift	1.1	1.2

Följande laborationsuppgifter skall redovisas för en handledare för godkännande under laborationen.

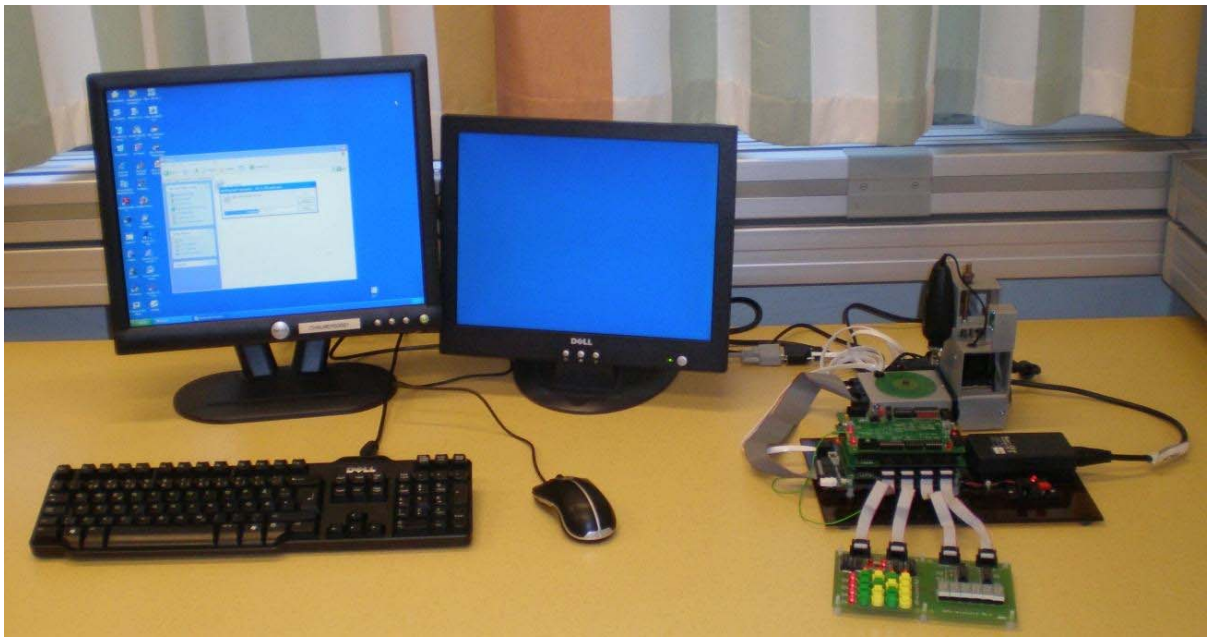
Laborations-uppgift	1.7	1.9
Sign.		

Laborationssystemet

Följande enheter ingår i laborationsmiljön:

- persondator med programvaran *ETERM för MC12* och *XCC12 för MC12*
- laborationssystemet *MC12* med monitor/debugger *DBG12*,
- borrmaskin, tangentbord och sifferindikatorer och olika typer av gränssnitt till dessa enheter.

Under laborationerna 1,2 och 3 använder du *ETERM* för att redigera, assemblera och testa dina program. Under senare laborationer använder du *XCC12* för att redigera, kompilera och testa dina program.



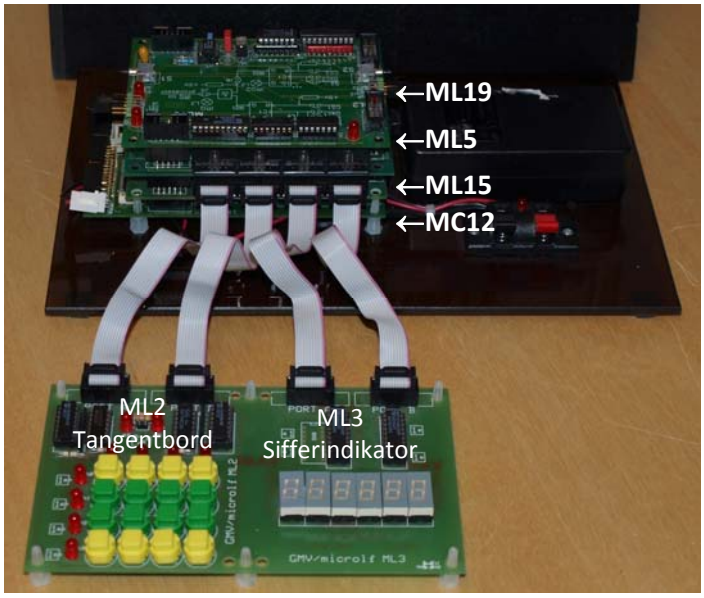
Anslutningar

Laborationsuppsättningen ska vara korrekt ansluten då du kommer till laborationsplatsen. Detta avsnitt sammanfattar anslutningarna.

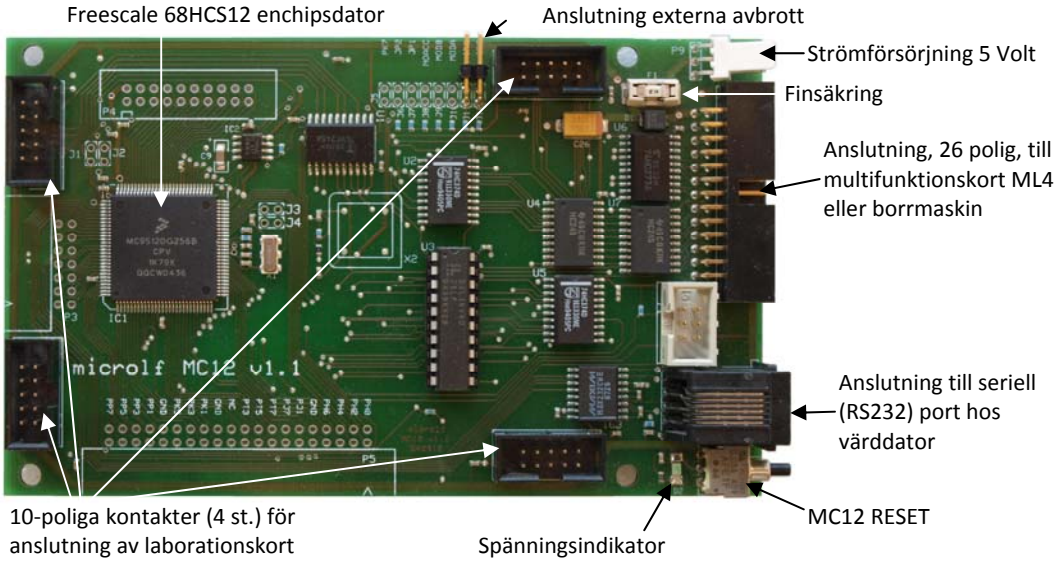
- Laborationssystemet *MC12* ansluts till terminalfunktionen i *ETERM* via persondatorns COM-port.
- Borrmaskinen ansluts till *MC12* via en 26-polig flatkabel.
- Tangentbordet (*ML2*) respektive sifferindikatorn (*ML3*) kan anslutas till *MC12* via två olika typer av gränssnitt, oberoende av varandra. Dessa gränssnitt kallas *ML15* respektive *ML5*.
- Laborationskorten *ML15* och/eller *ML5* utgör den fysiska kopplingen mellan *MC12* och *ML2*/*ML3*. De ansluts till *MC12* via en ”piggy-back” koppling som gör att flera laborationskort ansluts genom att dessa staplas på *MC12*.
- Laborationskortet *ML19* är speciellt anpassat för laborationer med *MC12’s* avbrottsmekanismer. Kortet beskrivs i ”Arbetsbok för MC12” och i detta PM.

Laborationsuppsättning:

Laborationsdator med laborationskort och strömförsörjning



Laborationsdator *MC12*

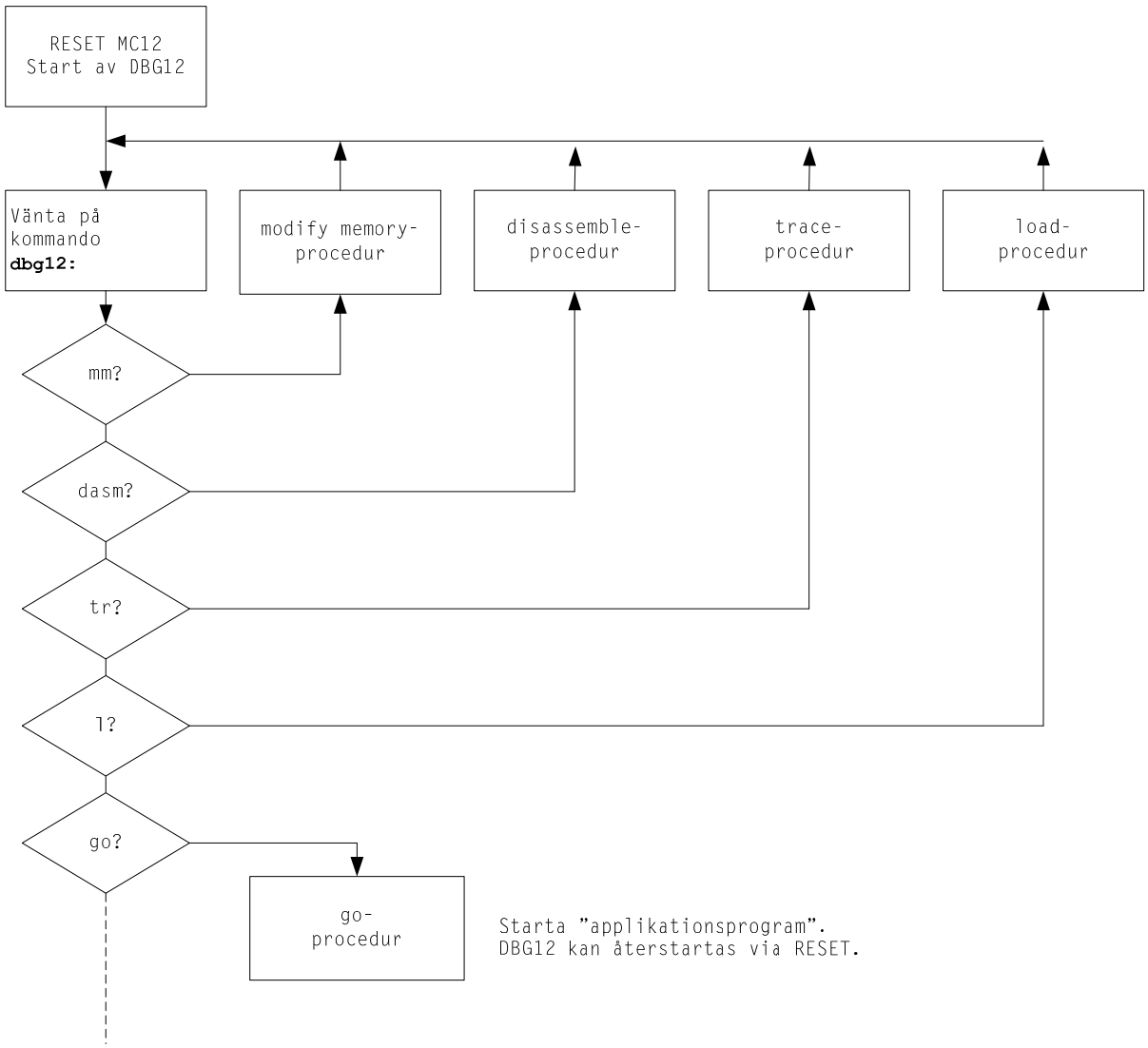


Monitor/debugger DBG12

När vi ansluter *MC12* till en COM-port på PC'n och startar ett terminalfönster fungerar PC'n som en "dum terminal" bestående av tangentbord och bildskärm. (ETERM = "Emulera terminal" = *härma terminal*).

Allt som skrivs på PC'ns tangentbord skickas direkt ner till *MC12*. I *MC12* finns ett enkelt program som kallas "monitor/debugger" (*DBG12*). Programmet är, som namnet antyder avsett för *övervakning* och *test*.

DBG12 "lyssnar" hela tiden på kommandon från tangentbordet bortsett från när *MC12* är upptaget av något applikationsprogram. För att återgå till *DBG12* i detta fall krävs RESET (omstart) av *MC12*. Huvudprogrammet i *DBG12* är utformat som en enkel kommandotolk, där en rad olika kommandon accepteras.



Läsanvisning:

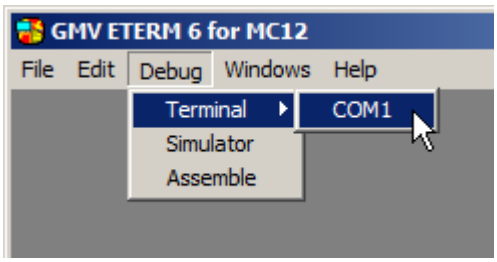
Läs om monitor/debugger'n *"DBG12 Användarbeskrivning"*
finns som länk från "resurssidan":
`"DBG12 monitor/debugger för MC12"`

Dispositionen av adressrummet hos *MC12* bestäms av *DBG12* och kan något förenklat beskrivas enligt följande:

I/O	Adress $0000-$0FFF	Med plats för I/O-kort och interna (HCS12) portar
RWM	Adress $1000-$3BFF	Med plats för dina program och data (applikationer)
RWM	Adress $3C00-$3FFF	Variabelarea för *DBG12*
ROM	Adress $4000-$FFFF	Med plats för bland annat *DBG12*

Laborationsuppgift 1.1:

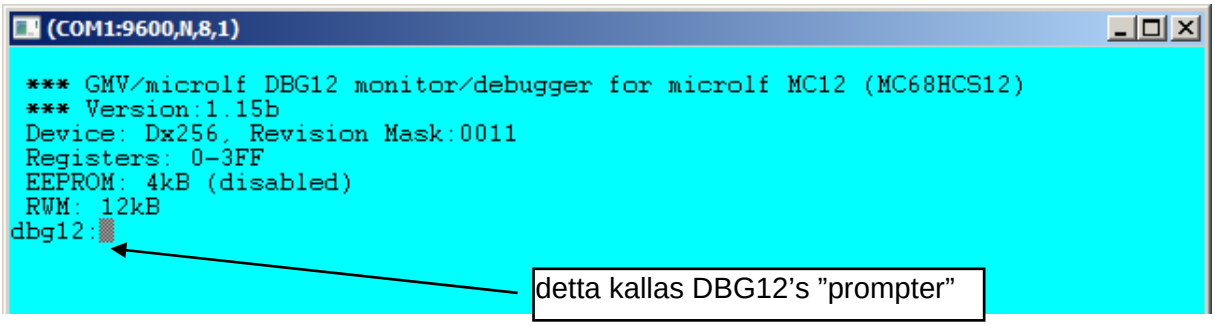
Starta *ETERM* och anslut en terminal under fliken Debug.



Tryck *RESET* på laborationssystemet *MC12* och kontrollera att *DBG12* identifierar sig genom att text skrivs till terminalfönstret.

Om ingen text skrivs till skärmen eller om texten är oläslig Kontakta en handledare.

Terminalfönstret bör se ungefär som följande figur.



Den sista raden kallas för monitorns prompter och innebär att *DBG12* är redo att ta emot kommandon från dig.

Varje gång du trycker ↵ ska *DBG12* skriva en ny prompter till terminalen. Om så inte är fallet betyder det att *DBG12* är upptagen och kanske rentav "hängt sig" eller "spårat ur". Lösningen på detta problem är att göra *RESET* på *MC12*.

Skriv: **`help`** ↵

DBG12 skriver ut en hjälptext. Se figuren till höger.

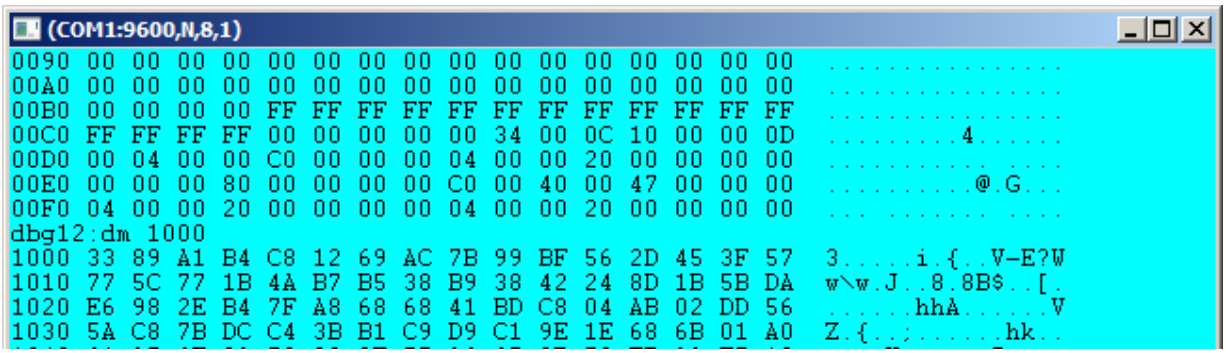
```
dbg12:
dbg12:help
 *** GMV/microlf DBG12 monitor/debugger HELP
Type help 'command' for any of:
 tr    - Trace instruction
 go    - Run program
 reg   - Display/Modify registers
 dm    - Display memory
 mm    - Modify memory
 dasm  - Disassemble
 bp    - Breakpoints
 l     - Load
 fload- Program Banked FLASH
 feras- Erase Banked FLASH
dbg12:help mm
 *** GMV/microlf DBG12 monitor/debugger HELP
 mm    - Modify memory
 Forms:
```

Undersök *olika* kommandon genom att skriva **`help`** ***`kommando`*** ↵ om du vill ha hjälp för något specifikt kommando.

Kommandot ”display memory” (dm)

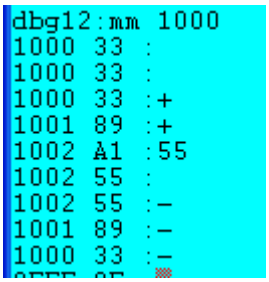
Undersök nu minnesinnehållen i de olika minnesområden vi har i systemet.

- Skriv: **dm 0**↵ för att studera adressområdet där vi har in-och utenheter.
- Skriv: **dm 1000**↵ för att studera adressområdet där du kan placera dina program.
- Skriv: **dm E000**↵för att studera adressområdet där vi har PROM (FLASH).



Adressangivelsen anges till vänster, minnesinnehåll i mitten och till höger tolkas minnesinnehållet som ASCII-koder.

Kommandot ”modify memory” (mm)



Använd nu kommandot mm för att försöka ändra minnesinnehåll på några olika ställen i *MC12’s* adressrum.

Ge kommandot: **mm 1000**↵ för att visa och ha möjlighet att ändra minnesinnehållet på adress 1000.

Tryck ↵ ett par gånger. Har du samma minnesinnehåll som i figuren till höger?_____

Tryck på + (plus) för att visa minnesinnehållet på *nästa* adress. Tryck på + på nytt och ändra minnesinnehållet på adress 1002 till 55 genom att skriva 55↵. Prova även – (minus) tangenten.

Vilka två sätt kan du använda för att avsluta mm-kommandot och återgå till monitorns prompter?

__ _____

_____

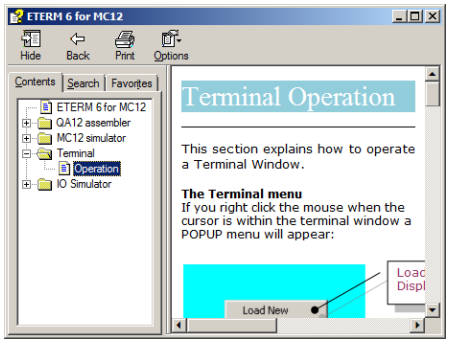
Kan du ändra minnesinnehållet på adress 4000?

_____

Prova att ändra ytterligare några minnesinnehåll inom adressintervallet 4000-FFFF. Kan du förklara vad som händer?

_____

Kommandot "load application program" (l)
Kommandot "disassemble" (dasm)
Kommandot "run application program" (go)



ETERM's terminalprogram har behändiga funktioner för att underlätta kommunikation mellan värddator och laborationsdator. Du kan läsa om sådana funktioner i *ETERM's* hjälpsystem.

På kursens resurssida, finns en `laddfil.s19` som du ska använda under nästa laborationsmoment. Filen innehåller såväl program som data. Din uppgift är att undersöka detta program.

Laborationsuppgift 1.2:

- Spara filen `laddfil.s19` i din arbetskatalog.
- Ladda `laddfil.s19` till *MC12* (högerklicka i terminalfönstret).
- *Disasssemblera*, dvs. översätt från maskinkod till assemblerkod, genom att använda *DBG12's* `dasm`-kommando. Komplettera tabellen.

Adress	Instruktion	Operand
1000	LDX	#$1020
100E	JMP	$C00F
101B	RTS	

- Utför nu programmet, starta med: **`go 1000`**↵.

Följ programmets instruktion och skriv ditt svar här:

Inmatning från tangentbord

I detta avsnitt använder vi tangentbordet *ML2* tillsammans med gränssnittet *ML15*.

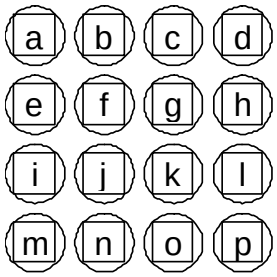
Läsanvisning:

Läs översiktligt om gränssnittet till tangentbord och sifferindikator i *Teknisk beskrivning ML15* finns som länk från ”resurssidan”:

”Laborationskort ML15, gränssnitt mot tangentbord/display (ML23), för MC12”

Laborationsuppgift 1.3:

Undersök tangentbordet *ML2*/*ML15*.
Använd `mm`-kommandot. *ML15’s* avkodning av tangentbordet, dvs. den kod som placeras i registret på adress $9C0, beroende på vilken tangent som tryckts ned, illustreras av följande figur:



Prova nu samtliga tangenter och komplettera nedanstående tabell.
Observera vad du läser från tangentbordet när ingen tangent är nedtryckt. Har föregående tangent-nedtryckning någon betydelse för avläsningen?

Nedtryckt tangent	*Avläst värde*	*Nedtryckt tangent*	*Avläst värde*
a		i	
b		j	
c		k	
d		l	
e		m	
f		n	
g		o	
h		p	

Vi övergår nu till inledande programutveckling i assemblerspråk. För att underlätta test av ett assemblerprogram introducerar vi ytterligare ett av *DBG12*’s kommandon:

`Kommandot “step (trace) application program” (tr)`

Laborationsuppgift 1.4:

Kontrollera din lösning av uppgift 41 i *Arbetsbok för MC12*, dvs. tangentbordsrutinen `CheckKbdML15`. Använd samma testprogram som i arbetsboken

Vi använder TRACE-kommandot som fungerar så att den angivna instruktionen utförs, därefter återförs kontrollen till *DBG12* och du kan inspektera/ändra register- minnesinnehåll etc. Du ska nu övertyga dig om att subrutinen `CheckKbdML15` fungerar som den ska.

- Ladda ner programmet till *MC12*.
- Ge kommandot **`tr 1000↵`** för att utföra programmets första instruktion.
- Utför programmet instruktionsvis, kontrollera programflödet, tryck ned någon tangent på *ML2*, kontrollera inläsningen av tangentkoden.
- Verifiera slutligen att `GetKbdML15` fungerar med *MC12*. (Uppgift 42 i arbetsboken).

Fortsätt nu med att praktisera test och felsökning i assemblerprogram. En viktig metod är att sätta så kallade *brytpunkter* på lämpligt valda ställen i programmet. För detta kan du använda *DBG12's* brytpunktshantering.

Kommandot "breakpoints" (bp)

Laborationsuppgift 1.5:

Testa nu `CheckKbdML15` med användning av brytpunkt.

- Bestäm adressen för NOP-instruktionen i testprogrammet och sätt en brytpunkt på denna adress.
- starta programmet med **`go 1000`**↵, tryck ned någon tangent på *ML2*, programmet ska nu stanna vid brytpunkten, vilket meddelande får du från *DBG12*?

Utmatning till sifferindikator

I detta avsnitt använder vi sifferindikatorn *ML3* tillsammans med gränssnittet *ML15*.

Läsanvisning:

Läs översiktligt om gränssnittet till tangentbord och sifferindikator i
Teknisk beskrivning ML15 finns som länk från "resurssidan":

`"Laborationskort ML15, gränssnitt mot tangentbord/display (ML23), för MC12"`

I nästa uppgift kommer vi att använda oss av:

Kommandot "display/modify registers" (reg)

Laborationsuppgift 1.6:

Du ska undersöka hårdvaran *ML3*/*ML15*. För detta använder du en enkel instruktionssekvens för initiering/visning:

```
      ORG   $1000
      LDAA  #1
      STAA  $9C2
      LDAA  #$D0
      STAA  $9C3
      LDAA  #0
      STAA  $9C2

      LDAA  #8
loop:
; Placera lämpligt värde i ackumulator B innan instruktionen utförs
      STAB  $9C3
      DECA
      BNE   loop
end:
      BRA   end
```

- Redigera, assemblera programsekvensen ovan.
- Ladda till *MC12*.

- Programmet ska utföras instruktionsvis (`tr 1000` osv.)
- Omedelbart innan instruktionen `STAB $9C3` utförs ska du placera lämpligt värde i register B med kommandot `reg B` *värde* ↵.

Sifferindikatorn har bara 6 positioner, kan alltså bara visa 6 siffror men trots detta måste en sekvens om 8 siffror ges till adress $9C3 för att indikatorerna ska tändas. Är det de 6 första, eller de 6 sista siffrorna i sekvensen som visas på sifferindikatorn?

- Kontrollera din lösning av uppgift 43 (`DisplayML15`) i *Arbetsbok för MC12.*
- Kontrollera att `DisplayML15` fungerar med *MC12*.

Laborationsuppgift 1.7:

Konstruera nu en subrutin `GetPut`, enligt följande specifikation:

```
;  Subrutin GetPut
;  Väntar på ny tangentnedtryckning
;  Visar därefter tangentkod för senast nedtryckta tangent på ML2 som
;  hexadecimala tecken på samtliga sifferindikatorer på ML3
;
```

Skriv nu följande huvudprogram, avsett för att testa utmatningsrutinen:

```
         ORG      $1000
main:    JSR      GetPut           ; Läs tangentbord, skriv till display
         BRA      main

GetPut:  … använder dina rutiner GetKbdML15 och DisplayML15
```

- Redigera, assemblera, rätta eventuella fel.
- Ladda programmet till *MC12* för att testa `GetPut`.
- Starta programmet, `go 1000`↵, testa rutinen för alla möjliga tangentnedtryckningar och verifiera korrekt funktion.

Visa upp din lösning för godkännande av en handledare.

Styrobjekt borrmaskin

Huvuddelen av de fortsatta laborationsmomenten i maskinorienterad programmering, såväl då det gäller programutveckling i assemblerspråk, som programmering i maskinnära C kommer att inbegripa styrobjektet ”borrmaskin”. Vi ska därför redan nu bekanta oss med denna laborationsutrustning så att framtida laborationsförberedelser (med hjälp av simulatorer) kan förstås mot bakgrund av den hårdvara som används under laborationerna.

Läsanvisning:

Läs om borrmaskinen på sidorna 49 och 50 i *Arbetsbok för MC12*.

Du kan också studera avsnittet om borrmaskinen i *ETERM’s* hjälpsystem.

Laborationsuppgift 1.8:

Studera borrmaskinen på laborationsplatsen framför dig. Vänd maskinen så att du ser ljusdioderna. Undersök hårdvaran med hjälp av följande programsekvens (Jämför med uppgifterna 71 och 72 i arbetsboken):

```
        ORG         $1000
Loop    NOP                     ; Ge indata till register A
        STAA        $400        ; Skriv till borrmaskinen
        LDAB        $600        ; Läs statusregistret
        BRA         Loop
```

- Stega genom programmet med `trace` när du undersöker borrmaskinen.
- Vid `NOP`-instruktionen kan du använda `reg`-kommandot för att placera lämpligt värde i register **A**.
- Testa att starta borret och sänka det. Observera lysdioderna på borrmaskinen. Gröna lysdioder är statusregistret och gula är styrregistret. Observera att bit 3-7 i statusregistret inte är definierade och kan anta vilka värden som helst.
- Vrid även på arbetsstycket för att ställa detta i referensposition och observera att lysdioden på borrmaskinen aktiveras – och – att du läser status till register **B**.
- Tryck RESET på *MC12*. Studera lysdioderna (statusregistret) på borrmaskinen och pressa ner borret för hand till bottenläge. Observera hur statusbitarna ändras. (Kanske är tryckfjädern som lyfter borret lite klen – lyft borret upp i så fall)

Det finns några skillnader mellan borrmaskinen i simulatorn och laborationssystemets borrmaskin som visserligen verkar små men har stor praktisk betydelse. Här får du några tips som kan hjälpa dig åtskilligt då du kommer till laborationsplatsen.

Adresser: Tänk på att simulatorns register mot borrmaskinen är konsekutiva (typiskt använder du här adresserna 400_{16} och 401_{16}). I laborationssystemet motsvaras 401_{16} i simulatorn i stället av adress

600_{16}. För att minska risken för bortkastad felsökningstid i laborationslokalen kan du införa en villkorlig assembleringssats i filen `Labdefs.s12`, exempelvis enligt:

```
#ifdef   SIMULATOR
DrillStatus    EQU   $401
#else
DrillStatus    EQU   $600
#endif
```

I ditt huvudprogram, innan filen `Labdefs.s12` inkluderas, kan du definiera

```
#define SIMULATOR
```

när du kommer till laborationen kommenterar du bort detta på följande sätt:

```
; #define SIMULATOR
```

Vi har en liknande problematik då det gäller fördröjningssekvensen i subrutinen `Delay`, som du ska göra som ett led i förberedelserna.

Hemuppgift 1.2:

Du har tidigare (i arbetsbokens första avsnitt) experimentellt bestämt fördröjningskonstanter för den simulerade miljön och ska nu bestämma fördröjningskonstanten i laborationsdatorn med liknande metod. Anledningen till att vi inte kan använda den tidigare fördröjningsrutinen är att denna använder 8-bitars fördröjningskonstant, vilket fungerar i simulatorn men blir alldeles för liten i hårdvara. Vi utformar därför fördröjningsrutinen med 16-bitars register. Använd följande programsekvens för dina tidsuppskattningar:

```
DelayConst:  EQU      ???
             ORG      $1000

Start:       CLRA
DELAY:       LDX      #DelayConst
NEXT:        LEAX     -1,X
             LDY      #100
NEXT2:       LEAY     -1,Y
             CPY      #0
             BNE      NEXT2
             CPX      #0
             BNE      NEXT
             COMA
             STAA     $400
             BRA      DELAY
```

- Gör ett antal praktiska försök, dvs. prova olika värden på `DelayConst` (`xx` respektive `yy` nedan, `zz` bestäms vid laborationstillfället) så att ljusdioderna tänds och släcks en gång per sekund (en sekund mellan varje tändning), då du tycker noggrannheten är tillräcklig har du bestämt fördröjningskonstanten för 500 ms fördröjning, dividera den därför med 2 och du har den fördröjningskonstant (250 ms) som ska användas av `Delay` i laborationssystemet.

Redigera den villkorliga assembleringssatsen i `Labdefs.s12` enligt följande:

```
#ifdef   SIMULATOR
#ifdef   RUNFAST
DelayConst  EQU   xx    ; har du bestämt under förberedelserna
#else
DelayConst  EQU   yy    ; har du bestämt under förberedelserna
#endif
#else
DelayConst  EQU   zz    ; din konstant för laborationssystemet
#endif
```

Laborationsuppgift 1.9:

Du ska nu avslutningsvis bestämma en fördröjningskonstant som fungerar även i laborationsdatorn:

- Anslut *ML4* till laborationsdatorn, kontrollera att sektionen Parallel Output är kopplad på kortet.
- Gör ett antal praktiska försök, dvs. prova olika värden på `DelayConst` så att ljusdioderna tänds och släcks en gång per sekund (en sekund mellan varje tändning), då du tycker noggrannheten är tillräcklig har du bestämt fördröjningskonstanten för 500 ms fördröjning, dividera den därför med 2 och du har den fördröjningskonstant (250 ms) som ska användas av `Delay` i laborationssystemet.

Visa upp din lösning för godkännande av en handledare.

Sammanfattning av laboration 1

Du ha undersökt och provat hårdvara:

- laborationssystemet *MC12* med *DBG12*
- I/O-enheter *ML2*, *ML3*, med gränssnitt *ML15* och styrobjekt ”borrmaskin”

Du har konstruerat och testat programdelar som ska användas i kommande laborationer:

- Inmatning från tangentbord
- Utmatning till sifferindikator
- speciellt, dimensionerat en fördröjningsrutin som är central i den fortsatta utvecklingen av programpaketet.

Laboration nr 2 behandlar

Styrning/övervakning av en borrmaskin

Följande uppgifter ur *Arbetsbok för MC12* ska vara utförda innan laborationen påbörjas. Du ska på begäran av laborationshandledare redovisa dessa.

Uppg.	82-101
Sign.	

Följande laborationsuppgift ur denna del av laborations-PM skall redovisas för en handledare för godkännande under laborationen.

Laborations-uppgift	2.2	2.3
Sign.		

Laborationsuppgift 2.1:

Du skall nu arbeta med filen `Main.s12` från de obligatoriska uppgifterna i arbetsboken.

Kontrollera att du använder rätt I/O-adresser.

Lägg till en `NOP`-instruktion direkt efter `ORG $1000` för att få en bättre utskrift på skärmen när du kör `trace`.

Assemblera och rätta eventuella fel. Ladda programmet till *MC12* genom att högerklicka i terminalfönstret och välja filnamn.

Undersökning av programmet i *MC12*.

Maskinprogrammet är nu laddat till *MC12*. Öppna listfilen `Main.lst` och ha denna tillgänglig på skärmen framför dig.

Kontrollera nu att programmet är placerat i minnet på *MC12* genom att utnyttja monitorns disassembleringskommando.

Ge kommandot **`dasm 1000`**↵ Jämför det du ser på skärmen med listfilen.

Leta upp startadressen för subrutinen `COMMAND` i listfilen och använd `dasm`-kommandot på nytt för denna adress. Troligen ser de dissassemblerade instruktionerna efter `RTS`-instruktionen konstiga ut. Kan du förklara varför?

_____

_____

Adresserna till subrutinerna som används för att styra borrmaskinen ligger i en tabell i minnet med begynnelseadressen `JUMPTAB`. Eftersom adresserna är 16 bitar breda krävs det 2 st. minnesord för att rymma varje adress. Adresserna lagras på standardformat med den mest

signifikanta byten på minnesadressen med lägst värde. Använd `mm`-kommandot för att studera `JUMPTAB` i minnet och jämför dessa startadresser med vad du erhöll med `dasm`-kommandot ovan. Diskutera med en handledare (nu eller sedan) om du är osäker.

Studera listfilen och identifiera startadresser (med start på `JUMPTAB`) för de subrutiner du implementerat. Använd därefter `dasm`-kommandot och verifiera att tabellen verkligen innehåller startadresser till dina subrutiner.

Test av huvudprogrammet ”main”

Du ska nu använda brytpunkter för att verifiera huvudprogrammet.

Se följande utdrag av huvudprogrammet i `Main.s12`. Lägg till en `NOP`–instruktion efter anropet av tangentbordsrutinen och före anrop av kommandotolken. Assemblera och rätta eventuella fel i programmet.

Studera därefter listfilen och undersök vilken adress `NOP`-instruktionen i huvudprogrammet är placerad på, sätt en brytpunkt på denna adress.

```
;  Main.s12
;  Operatörsstyrd borrautomat

;  Definitioner
   USE      Labdefs.s12
   ORG      $1000
main:
   ---   Initiera borrmaskin
; Huvudprogram, invänta vald operation
main_loop:
   JSR      CheckKbdML15
* Tangentkod nu i register B...
* Utför vald operation
   NOP
   JSR      Command
   BRA      main_loop
```

Starta nu programmet med **`go 1000`**↵.

Tryck ner tangent med kod ’7’ (ej implementerad Auto-funktion), efter det att du tryckt ner tangenten på tangentbordet ska programmet stanna vid brytpunkten och *DBG12* ger en utskrift till skärmen.

> Om dette INTE händer, tryck RESET på *MC12* och starta om programmet med instruktionsvis exekvering **`tr 1000`**↵,**`tr`**↵ osv, kontrollera tangentborsrutin och rätta fel.

Vid brytpunkten, kontrollera innehållet i register **B**, det valda kommandonumret (7).

Ge kommandot **`go`** för att nu testa tangent med kod ’0’ (starta borrmotor), fortsätt från brytpunkt med instruktionsvis exekvering (**`tr`**), försök även följa med i listfilen, kontrollera att borrmaskinens motor startar. (Kontrollera ev. strömbrytaren till borrmotorn).

Med detta har du kontrollerat att huvudprogrammet fungerar som det ska och det återstår nu att även kontrollera de implementerade funktionerna.

Laborationsuppgift 2.2:

Test av implementerade subrutiner

Du ska nu systematiskt testa de fyra första subrutinerna. Då du övertygat dig om att en subrutin fungerar korrekt fyller du i ”kontrollkolumnen” i följande tabell. Glöm inte att även testa *alarmfunktionen*, detta gör du i samband med test av subrutinen Step.

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

Tänk på att fördröjningsrutinen nu ska vara anpassad till den verkliga miljön och inte den simulerade.

Tryck RESET på *MC12*, ta bort eventuella brytpunkter och starta programmet, **`go 1000`**.

Kontrollera de fyra olika funktionerna.

Rätta eventuella fel, spara alla dina filer och visa upp resultatet för en handledare.

Laborationsuppgift 2.3:

Kontrollera funktionen hos återstående funktioner DrillHole, RefPos och Auto i *MC12*.

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

Rätta eventuella fel, spara alla dina filer och visa upp resultatet för en handledare.

Du har nu

- Färdigställt ett operatörsstyrt (interaktivt) program i assemblerspråk. Programmet realiserar en rad funktioner som krävs för att styra en borrmaskin. Under laboration 5 ska du realisera samma funktioner men då genom att använda programspråket 'C'.

Laboration nr 3 behandlar

Pseudoparallell exekvering

Följande uppgifter ur *Arbetsbok för MC12* ska vara utförda innan laborationen påbörjas. Du ska på begäran av laborationshandledare redovisa dessa.

Uppg.	65-67
Sign.	

Följande hemuppgifter ska vara utförda innan laborationen påbörjas.

Hem-Uppgift	3.1	3.2

Följande laborationsuppgift ur denna del av laborations-PM skall utföras och redovisas för en handledare för godkännande.

Laborations-uppgift	3.3
Sign.	

Övrigt:

Subrutinen DISPLAY som ska användas i laborationsuppgift 3.3 finns färdig (`Display_ML5.s12`), hämta den från ”resurssidan”.

Laborationsuppgift 3.1:

Studera komponenttrycket (vita texter och figurer) på *ML19* och jämför med bilden i I/O-simulatorn. Du har två tryckknappar , S1 och S2, på *ML19* som motsvarar Event1 och Event2 hos I/O-simulatorn. Du kan själv lista ut ljusdiodernas funktion när du fortsätter denna laborationsuppgift.

Tryck på S1 och på S2. Om lysdioderna på *ML19 var släckta* från början tänds dessa *nu*. Avbrotten var/är aktiverade och avbrottsvipporna var/är ettställda.

För att kvittera ett avbrott, gör en skrivning med mm-kommandot på adress $DC2_{16}$ och därefter en skrivning på adress $DC3_{16}$. Avbrottsvipporna nollställs oberoende av vilket värde du skriver och ljusdioderna släcks.

Gör en läsning på adress $DC0_{16}$ som är *ML19*’s statusregister. Detta skall nu vara nollställt. Studera ljusdioderna på *ML19* och tryck på S1. Gör därefter en ny läsning av statusregistret och verifiera att b_0 är ettställd.

Studera ljusdioderna på nytt och tryck på S2. Gör därefter ännu en läsning av statusregistret och verifiera att b_1 också är ettställd.

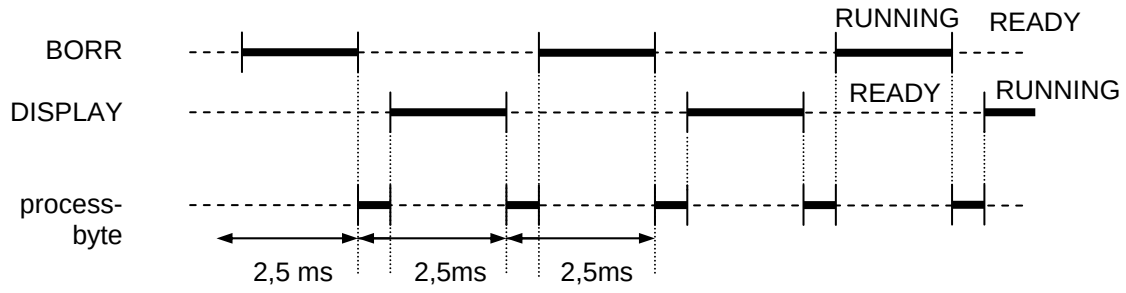
Parallell programexekvering

Den sista uppgiften under denna laboration är att utföra borrprogrammet och displayrutinen på *MC12* så att det verkar som att dessa körs "samtidigt", de körs *pseudoparallellt*.

Vi inför ett avbrottsdrivet system där processorn växelvis styr det ena och sedan det andra programmet. Växlingen går så snabbt att användaren upplever att de styrs parallellt. De båda programmen som ska utföras på detta sätt är:

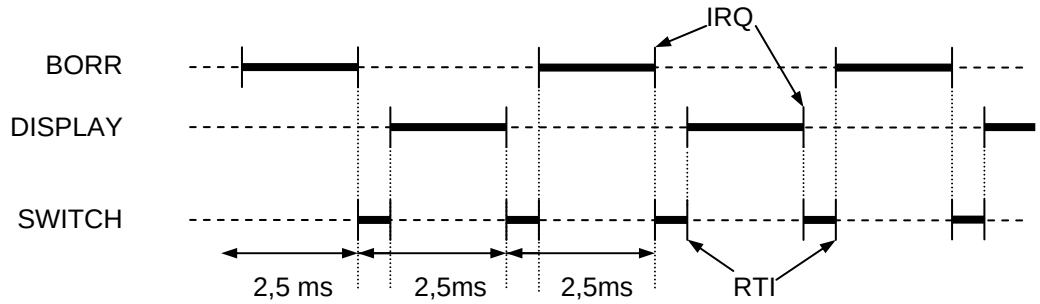
- Programmet BORR (Laborationsuppgift 2.3)
- Programmet DISPLAY (`Display_ML5.s12`) som du hämtar från ”resurssidan”

På laborationskortet *ML19* finns en klockgenerator med frekvensen 400 Hz. Denna används för att generera avbrott till processorn. Detta innebär att avbrott sker var 2,5 ms. Varje gång avbrott sker startas avbrottsrutinen som växlar program. Vi kallar detta *processbyte.*
I vår miljö definierar vi det program som för tillfället exekveras av processorn att vara i tillståndet *RUNNING*. Det program som inte exekveras och är redo att startas om på nytt, säger vi är i tillståndet *READY*. Så alltså när BORR är i tillstånd *RUNNING* så är DISPLAY i tillstånd *READY* och vice versa.



Figur 3.1

Avbrottsrutinen som åstadkommer processbytet ska utformas så att den andra processen återstartas genom att avbrottsrutinen utför `RTI`-instruktionen.



Figur 3.2

Följande beskriver då huvudsakligen vad som ska utföras i avbrottsrutinen:

BORR **avbryts:**	DISPLAY **avbryts:**
BORR’s status placeras på stacken	DISPLAY’s status placeras på stacken
BORR’s stackpekare skall sparas	DISPLAY’s stackpekare skall sparas
DISPLAY’s stackpekare till SP	BORR’s stackpekare till SP
Orsaken till avbrottet måste avlägsnas (nollställa avbrottsvippan)	
Slutligen, `RTI`	

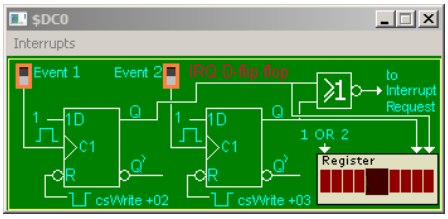
Hemuppgift 3.1:
Implementera parallell exekvering av BORR och DISPLAY.
Det är lämpligt att först isolera hela borrmaskinprogrammet (BORR) i en källtextfil `Drill.s12.`
Programmet DISPLAY hämtar du från resurssidan, källtextfilen `Display_ML5.s12.`

Din ”programkärna” som innehåller initieringssekvens, data och avbrottsrutin placerar du i filen `Kernel.s12` som lämpligen bör ha följande struktur:

```
   ORG   $1000

Kernel:
;  här placerar du initieringskoden


;  därefter följer data arean

; slutligen inkluderar du filerna med programmen:
   use   Drill.s12
   use   Display.s12
```

Assemblera `Kernel.s12` och rätta eventuella fel. Glöm inte att avlägsna eventuella ORG-direktiv i borrmaskinprogrammet. Kontrollera listfilen och försäkra dig om att programmen inte överlappar varandra i minnet.

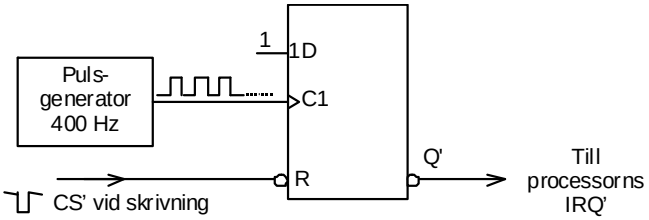
Hemuppgift 3.2: Testa ditt program i simulatorn



I den simulerade miljön är svårt att behandla snabba processbyten (400 processbyten per sekund) och dessutom omöjligt att följa händelseförloppet. Därför är det lämpligt att under simuleringsarbetet utnyttja en av de avbrottsvippor som finns tillgängliga i IO-simulatorn.

Genom att ersätta pulsgeneratorn med knapptryckningar (klicka på Event-knappen i I/O-simulatorn) bestämmer du när processbytet ska ske. Du kan därför lugnt studera händelseförloppet när du själv klickar i fönstret för avbrottsvippan. BORR kommer att stanna, processbytet utförs och DISPLAY startas.

I laborationssystemet har vi en pulsgenerator som är ansluten till processorns avbrottsingång.



Att generera 400 avbrott per sekund har vi tyvärr inte möjlighet till i vår simulerade miljö så vi får nöja oss med den verifiering vi genomfört ovan och testa våra program BORR och DISPLAY på ett verkligt system i laborationsmiljö i stället.

Laborationsuppgift 3.2:

Modifiera eventuellt konstanterna i din programvara så att det fungerar i hårdvaran.

Kontrollera att du använder avbrottsvektorn $3FF2

Ladda och testa din `Kernel.s12` på hårdvaran.

Testa programmen i hårdvaran på samma sätt som du gjorde tidigare i simulatorn. Sätt brytpunkter i BORR och DISPLAY. Verifiera att programmet stoppas vid dessa brytpunkter när du trycker på S1. Se till att hela programpaketet fungerar.

Ett bra visuellt sätt att se programväxlingen är att starta upp REFPO för borrmaskinen när du trycker på S1. Det ser då ut som borrmaskinen ”dör” och DISPLAY startas upp när du trycker på S1. Då du gör ännu ett tryck på S1 verkar det som DISPLAY ”dör” och borrmaskinen fortsätter. Ge ett antal tryck på S1 och studera förloppet.

När du känner dig nöjd, kontakta då en handledare som hjälper dig att växla till pulsgeneratorn så att du får ca 400 avbrott i sekunden.

Testa rutinen REFPO, går denna långsammare än tidigare?
Studera även DISPLAY. Verkar det som om denna blinkar på ett annat sätt nu än vad den gjorde i förra labbet?

Diskutera resultatet med din labbkompis, försök förklara vad som sker. Visa upp resultatet och redogör för era slutsatser för en handledare.

Du har nu

- Implementerat ett tidsdelningssystem i miniatyr, där du kört två program "samtidigt" (två processer) med endast en CPU. Metoden kan enkelt utvidgas till attt omfatta flera processer. Den här typen av tidsdelning är grundläggande för alla operativsystem bestyckade med enkelprocessorer.

Laboration nr 4 behandlar

C-programmering
Prioritetskö

Följande hemuppgift ska vara utförd innan laborationen påbörjas.

Hem-Uppgift	4.1

Följande laborationsuppgift ur denna del av laborations-PM skall utföras och redovisas för en handledare för godkännande.

Laborations-uppgift	4.1
Sign.	

Programmeringsmiljö

Använd utvecklingsmiljön ***CodeLite*** som bland annat finns på resurssidan. Där hittar du också en kortfattad ”tutorial” om hur du kommer i gång med att använda *CodeLite*.

För laborationen finns speciellt följande fil tillgänglig via kursens resurssida:
`Lab4_linkedlist.zip`

Hemuppgift 4.1:

- Om du inte tidigare använd CodeLite, arbeta igenom den ”tutorial” som finns på resurssidan.
- Skapa ”arbetsutrymme” och ”projekt” för denna laboration
- Läs noga igenom laborationsuppgifterna 4.1 och 4.2 så att du förstår vad du skall göra och hur problemen skall lösas.
- Gör en skiss av hur programmen skall se ut och försök att skriva programkoden i förväg, innan laborationstillfället.

I programspråket C finns det varken klasser eller generiska enheter, men trots detta kan man, om man programmerar på ett disciplinerat sätt, konstruera hyggligt återanvändbara programmoduler. I denna laboration får implementera en prioritetskö. Prioritetsköer används t ex internt i realtidsoperativsystem för att hålla reda på de olika processer som står i tur att exekveras och möjliggör en mer avancerad schemaläggning än den round-robin schemaläggning som används i laboration 2. Denna labb använder pekare, dynamiskt allokerade objekt och länkade datastrukturer vilket är nödvändiga förkunskaper för att kunna implementera styrningen av borrmaskinen i C (sista labben).

Programmeringsmiljö

Även för denna laboration är det lämpligt att använda *CodeLite*. Du ska också ladda ner och packa upp `Lab3_linkedList.zip`, som innehåller testprogram, skelett och h-fil för laborationen. Börja sedan med att skapa ett *CodeLite* -”projekt” bestående av filerna `qtest.c`, `queue.h` och `queue.c`.

Godkännande

Din kömodul skall provköras med programmet i filen `qtest.c`. När programmet fungerar skall det visas upp för en handledare för godkännande. För att laborationen skall bli godkänd räcker det inte med att programmet fungerar. Dina funktioner måste också vara skrivna på ett snyggt och begripligt sätt. Programraderna skall t.ex. indenteras (dras in) på det sätt som lärs ut i kursen.

Laborationsuppgift 4.1

Uppgiften är att konstruera en programmodul som kan användas för att skapa prioritetsköer. En modul i C skall som bekant alltid byggas upp med hjälp av två filer, en `.h`-fil som innehåller deklarationer av funktioner och typer och en `.c`-fil som innehåller funktionsdefinitionerna, dvs. implementeringen av funktionerna. I denna uppgift skall modulen bestå av de två filerna `queue.h` och `queue.c`. Filen `queue.h` är redan färdigskriven och finns på kursens webbsida. På kursens webbsida finns också ett färdigskrivet testprogram i filen `qtest.c`. Detta skall du använda för att provköra din kömodul.

- Din uppgift är att skriva filen `queue.c`.
- I denna fil skall du implementera alla de funktioner som deklareras i filen `queue.h`.

Obs! Du får inte ändra något i filen `queue.h`. Du måste också i filen `queue.c` använda dig av de typdefinitioner som ges i avsnittet *Implementering* nedan. De skall användas precis som de är.

Gränssnittet

I filen `queue.h` specificeras kömodulens gränssnitt mot andra programdelar:

```
// Filen queue.h
// Datatyp definierar typen för datan som skall läggas i kön.
#ifndef QUEUE_H
#define QUEUE_H

#define MAX_PRIO 100
typedef const char *DataPtr;

struct QueueElement {                    // typen för ett köelement
    struct QueueElement *next;        // pekare till nästa köelement
    int prio;                            // prioritet (ger köns ordning)
    DataPtr data;                        // pekare till dataelement
};

typedef struct QueueElement *QueuePtr;

QueuePtr new_queue();                        // Skapar en ny (tom) kö
void delete_queue(QueuePtr q);               // tar bort kön helt och hållet
void clear(QueuePtr q);                      // tar bort köelementen men behåller
kön
int  size(QueuePtr q);                       // räknar köns aktuella längd
void add(QueuePtr q, int prio, DataPtr d);   // lägger in d på rätt plats
DataPtr get_first(QueuePtr q);               // avläser första dataelementet
void remove_first(QueuePtr q);               // tar bort första köelementet

#endif
```

Typen `QueuePtr` definieras som pekare till typen `struct QueueElement`. Det gör det lite enklare att läsa argumenten till funktionerna, men är annars ekvivalent med att skriva ut `struct QueueElement*`.

För att skapa en ny kö anropar man funktionen `new_queue()`. Man kan sedan lägga in element i kön med hjälp av funktionen `add`. När man anropar funktionen `add()` styr prioriteten var det nya elementet läggs in. Ett element med hög prioritet placeras före ett med lägre prioritet och om flera element har samma prioritet hamnar dessa i s.k. FIFO-ordning (first in first out). Funktionen `add()` har tre parametrar: kön, prioriteten och en pekare till data (för element som skall läggas in i kön). (Det är egentligen inte datan som hamnar i kön, utan pekare till dem.) Den sista parametern har typen `DataPtr` och är en egendefinierad typ (via `typedef`), vilket gör kön flexibel om man vill återanvända den för olika ändamål. Om man t.ex. vill skapa prioritesköer av poster av typen `struct Person`, så skall man istället definiera `DataPtr` till på följande sätt:

```
typedef struct Person *DataPtr;
```

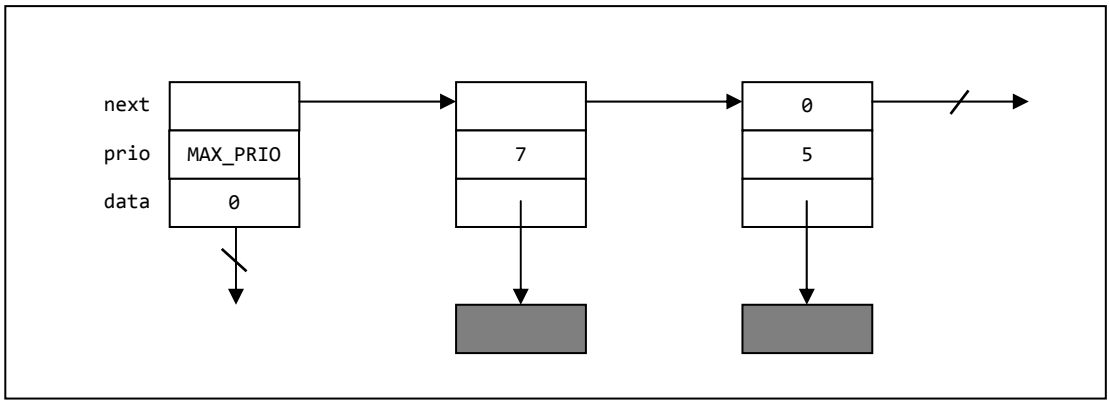
Funktionen `get_first()` avläser det första elementet i kön, utan att ta bort det, och funktionen `remove_first()` tar bort det första elementet. Funktionen `size()` ger köns längd. Funktionen

`clear()` tar bort alla element ur kön, dvs. egentligen alla pekarna till elementen. Kön blir då tom, men kan användas igen. Funktionen `delete_queue()` tar bort kön helt och hållet.

Implementering

Du skall implementera prioritetskön med hjälp av en *enkellänkad lista*. En sådan består av ett antal sammanlänkade poster, s.k. *köelement*. Varje köelement innehåller en pekare sompekar på nästa köelement. I denna laboration skall varje köelement dessutom innehålla ett heltal som anger köelementets prioritet samt en pekare till ett dataelement. Köelementen beskrivs av typen `struct QueueElement`, vilken redan är definierad i `queue.h`.

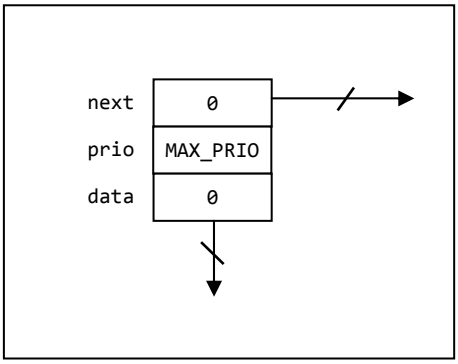
Figuren visar hur en prioritetskö, som för ögonblicket innehåller två dataelement, byggs upp. De två dataelementen (vilka kan ha vilken typ som helst) har markerats med skuggade rektanglar.



När man arbetar med länkade listor visar det sig att fallen att en lista är tom eller att man skall sätta in eller ta ut ett element först eller sist ofta måste specialbehandlas. Detta gör att funktionerna som hanterar listor kan blir ganska komplicerade. För att slippa ifrån dessa problem är det praktiskt att låta varje enkellänkad lista ha ett speciellt startelement som sitter först i listan. Då blir funktionerna mycket enklare.Vi skall utnyttja denna teknik i denna laboration. Det är därför det finns tre köelement i figuren ovan (av typen `struct QueueElement`) , trots att bara två dataelement har lagts in i kön. Observera att startelementet inte pekar till något dataelement (den pekar på `NULL`). Sista elementet i listan pekar inte heller ut något nästa element (den pekar också på `NULL`).

När man lägger in ett nytt dataelement i en kö skall man skapa ett nytt köelement (allokera det dynamiskt) och låta det peka på det nya dataelementet. Därefter skall man länka in det nya köelementet på rätt ställe i den enkellänkade listan. Prioriteten avgör placeringen. Högst prioritet först i listan och för enkelthetens skull så har startelementet den hösta möjliga prioriteten `MAX_PRIO` som definieras i `queue.h`.

En tom lista har endast ett startelement och ser ut som Figuren till höger.



När man tar bort ett dataelement från kön skall man länka ur motsvarande köelement ur listan och därefter frisläppa det allokerade minnesutrymmet.

Test programmet (i `qtest.c`) består av tre test. Man anger hur många test man vill testa genom att definiera `TESTS_TO_TRY` till ett tal mellan noll och tre. De tre testen är:

1. Skapa kö, lägga till element och beräkna storlek.
2. Ta bort först elementet och ta bort alla element.
3. Ta bort kön och kontroll av minnesläckor.

För att bli godkänd måste alla tre avklaras, men man kan testa sin kö implementation i steg.

För test 1 måste man implementera `new_queue()`, `add()`, `size()`, och `get_first().`

För test 2 måste man implementera `remove_first()`, och `clear().`

För test 3 måste man implementera: `delete_queue()` och använda det externa verktyget DrMemory som inkluderats i zip-filen. För att testa ditt program så drar du ditt färdigkomilerade program (`.exe`) och släpper det på `drmemory.exe` som finns i mappen `drMemory/bin/`. Ditt program kommer då att köras som vanligt, men alla anrop till `malloc()` och `free()` kommer att registreras av DrMemory. När ditt program terminerar öppnas en textfil med statistik över felaktig minnesanvändning med referenser till vilka rader som orsakar dessa. Se till att fixa eventuella läckor och felaktiga minnes accesser.

Laboration nr 5 behandlar

Användning av XCC12 för korskompilering till MC12

Styrning av borrmaskin

Följande hemuppgift ska vara utförd innan laborationen påbörjas.

Hem-Uppgift	5.1

Följande laborationsuppgift ur denna del av laborations-PM skall utföras och redovisas för en handledare för godkännande.

Laborations-uppgift	5.4
Sign.	

Läsanvisningar:

Laborationen förutsätter att du arbetat igenom avsnitt 5, sidorna 63-74, i *Arbetsbok för MC12*. Det är lämpligt, dock ej obligatoriskt, att utföra uppgifterna 102-107.

I denna laboration får du lära dig hur man med hjälp av en korskompilator kan utveckla C-program för en dator som direkt styr hårdvara.

Programmeringsmiljö

Korskompilatorn *XCC12* skall användas. Tänk på att denna skiljer sig från *GCC* genom att den är *C89*-kompatibel (”ANSI”) snarare än *C99*. Den stora fördelen med *XCC12* är att utvecklingsmiljön, vid sidan av källtextdebugger, också innehåller simulatorer, på samma sätt som i *ETERM*, vilket gör det enklare för dig att förbereda laborationen med borrmaskinen.

Godkännande

När programmet fungerar skall det visas upp för en handledare för godkännande. För att laborationen skall bli godkänd räcker det inte med att programmet fungerar. Det måste också vara skrivet på ett snyggt och begripligt sätt. Programraderna skall t.ex. indenteras (dras in) på det sätt som lärs ut i kursen. Du måste också ha delat in det i moduler med användning av include-filer så som beskrivits i detta lab-pm.

Hemuppgift 5.1:

Läs noga igenom laborationsuppgifter 5.1-5.4 så att du förstår vad du skall göra och hur problemen skall lösas. Du måste ha arbetat igenom avsnitt 5 i *”Arbetsbok för MC12”*. För att hinna göra laborationen är det nödvändigt att du före laborationstillfället har skrivit C-programmen och testat dem in källkodsdebuggern i *XCC12*.
Läs också igenom avsnittet om CRG-kretsen, i häftet ”Maskinnära programmering med HC12”.

Mer om portadressering (absolut adressering) i C

Portadresser i minnet kan enkelt adresseras. För att göra det på ett snyggt sätt i C är det lämpligt att börja med att definiera en typ som beskriver portar. Om man, som i *MC12*, har 8-bitars portar kan man göra deklarationen:

```
typedef unsigned char * port8ptr;      // pekare till 8-bitars port
```

För 16-bitars portar ska man i stället använda

```
typedef unsigned short * port16ptr;    // pekare till 16-bitars port
```

och hade man haft 32-bitars portar hade man skrivit

```
typedef unsigned long * port32ptr;     // pekare till 32-bitars port
```

då den aktuella kompilatorn, så som XCC12, använder 32 bitar för typen `long`.

Typen `port8ptr` kan alltså användas som en pekare till en 8-bitars port. En utport på *ML4* har adressen 400_{16}. För att inte behöva lägga denna sifferkonstant inne i programmet definierar man lämpligen en macro:

```
#define ML4OUT_ADDRESS 0x400
```

För att enkelt kunna skriva till den 8-bitars porten kan man definiera ytterligare en macro:

```
#define ML4OUT *((port8ptr) ML4OUT_ADDRESS)
```

Uttrycket

```
(port8ptr) ML4OUT_ADDRESS
```

är en explicit typomvandling från `int` (konstanten 0x400) till typen `portptr`. Den inledande asterisken innebär att man tar det som denna pekare pekar på, dvs. utregistret på *ML4*.
Vill man kunna läsa från inporten på *ML4* som har adressen 600_{16} kan man på motsvarande sätt definiera följande två macron:

```
#define ML4IN_ADDRESS  0x600
#define ML4IN  *((port8ptr) ML4IN_ADDRESS)
```

Nu kan man använda sig av dessa macron för att komma åt portarna. Följande sekvens visar t.ex. hur man deklarerar en variabel, tilldelar denna värde från *ML4*'s inport, skiftar bitarna i variabeln ett steg åt höger och slutligen skriver variabelns värde till *ML4*'s utport:

```
port8 r;
r = ML4IN ;
r = r >> 1;
ML4OUT = r;
```

Macrona `ML4IN` och `ML4OUT` beskriver då egentligen att man avläser det som finns på adressen 600_{16} och skriver till adressen 400_{16}. Naturligtvis går det att göra detta utan att införa typerna `port8` och `port8ptr` och utan att definiera några macron, men om man gör på det sätt som beskrivits här blir programmen mycket tydligare och därför lättare att få felfria.
Man bör lägga alla makron av detta slag i en inkluderingsfil som t.ex. heter `ports.h`. I denna bör man också lägga typdefinitionerna. Fördelen med detta är att det är lätt att hittta en portadress och ändra den om konfigurationen ändras.

Som exempel på en port vars adress varierar med den använda miljön har vi statusregistret hos "borrmaskinen" som du laborerat med tidigare och som du återkommer till i denna laboration. I simulatorn har detta register adress 401_{16} medan i laborationssystemet finns det på adress 600_{16}. Med villkorlig kompilering skapar du definitioner så att du sedan, på ett enkelt sätt, kan skapa versioner för den simulerade miljön, såväl som den fysiska laborationsmiljön:

```
#ifdef SIMULATOR
   #define DRILLSTATUS_ADDRESS 0x401
#else
   #define DRILLSTATUS_ADDRESS 0x600
#endif
```

Man kan naturligtvis använda *C*:s alla olika operatorer för att manipulera de enskilda bitarna i en port när man skriver till den. För att kunna göra detta på ett bekvämt sätt kan man definiera följande macron:

```
#define set(x, mask)    (x) = (x) | (mask)
#define clear(x, mask)  (x) = (x) & ~(mask)
```

alternativt (samma betydelse)

```
#define set(x, mask)    (x) |= (mask)
#define clear(x, mask)  (x) &= ~(mask)
```

För att t.ex. sätta bitarna b_0 och b_1 hos *ML4*:s utport kan man ledas att tro att man ska skriva

```
set(ML4OUT, 0x3);
```

vilket på många sätt är riktigt tänkt. Dock är det fel i detta fallet eftersom det inte går att avläsa värden som man tidigare skrivit till en utport. Ibland händer det också att man vill göra flera ändringar i en port när man skriver till den, men att alla ändringarna *måste ske samtidigt* i porten.

I sådana här fall är det lämpligt att använda ett s.k. *skuggregister*. Detta är en vanlig variabel som man hela tiden låter innehålla en kopia av porten. Om man t.ex. vill ha ett skuggregister för ML4:s utport kan man göra deklarationen

```
unsigned char ML4shadow = 0;  // deklaration av kopia
```

Vill man göra ändringar i utporten utför man sedan *först dessa på skuggregistret* och *sedan kopierar man skuggregistrets värde till porten i en enda tilldelningsoperation.* Antag t.ex. att man som ovan vill sätta b_0 och b_1 i *ML4*:s utport men att man även vill nollställa b_7. Då kan man skriva

```
set(ML4shadow, 0x3);
clear(ML4shadow, 0x80);
ML4OUT = ML4shadow;
```

Avbrottshantering i C

När debuggern *DBG12* körs i *MC12* fångar denna avbrotten på de ”riktiga” avbrottsvektorerna. Men avbrotten skickas vidare till avbrottsvektorer som har samma adress som de riktiga, fast med adresser som börjar på 3 istället för F. Avbrott med vektor på adress $FFF2_{16}$ skickas t.ex. vidare till adressen $3FF2_{16}$.

En avbrottsvektor innehåller adressen (en pekare) till en avbrottsrutin som skall anropas när avbrott av ett visst slag inträffar. En avbrottsrutin är en parameterlös funktion som inte ger något returvärde. Man kan alltså i C göra följande typdefinition som beskriver typen för en avbrottsvektor:

```
typedef void (*vec) (void);      // avbrottsvektor
```

För att kunna ändra en avbrottsvektor behöver man en pekare till den. En sådan pekare har typen

```
typedef vec *vecptr;             // pekare till avbrottsvektor
```

Som exempel visas hur man kan lägga in en pekare till en avbrottsrutin i avbrottsvektorn på adressen $3FF2_{16}$ i *MC12*. Man börjar med att göra definitionerna

```
#define IRQ_VEC_ADR  0x3FF2
#define IRQ_VEC      *((vecptr) IRQ_VEC_ADR)
```

Adressen till avbrottsrutinen kan nu läggas in genom att man gör en enkel tilldelning till `IRQ_VEC`. Antag t.ex. att avbrottsrutinen heter `inthandler` och är deklarerad på följande sätt i en .h-fil som man har inkluderat i sitt program

```
void inthandler (void);
```

För att styra avbott på avbrottsvektorn $3FF2_{16}$ till denna rutin kan man göra tilldelningen

```
IRQ_VEC = inthandler;
```

När en avbrottsrutin anropas går inte anropet till på samma sätt som när man gör ett vanligt funktionsanrop. I MC68HC12 sparas t.ex. alla processorns register på stacken innan anropet. Detta betyder att man inte kan återvända från en avbrottsrutin på det sätt som man gör från vanliga funktioner. En speciell instruktion (`RTI`) måste användas. Av denna anledning måste funktionen `inthandler` vara skriven i assembler. En standard C-kompilator kan nämligen bara generera kod för vanliga funktioner. Men det finns inget som hindrar att avbrottsrutinen bara innehåller att anrop av en vanlig C-funktion, vilken får göra själva jobbet.

För att MC68HC12skall acceptera avbrott måste man i programmets början se till att nollställa I-flaggan. Detta kan inte heller göras i standard-C. Man måste antingen anropa en assemblerfunktion som utför denna operation eller så kan man använda (icke standardiserad) inbäddad assemblerkod.

Anmärkning:

Längre fram ska du använda avbrott från CRG-kretsen, tänk då speciellt på att denna krets använder avbrottsvektor $3FF0_{16}$.

I *XCC* finns visserligen ett icke standardiserat nyckelord `__interrupt` som man kan använda för att få en funktion att avslutas som en avbrottsrutin och man kan då skriva även undantagshantering helt och hållet i C. Under denna laboration måste du dock följa ANSI-C89, varför du *inte* får använda detta nyckelord i din lösning.

Laborationsuppgift 5.1

Starta programmet XCC och välj alternativet *File* | *New Workspace*. Kalla din nya workspace *lab6.w12*. I denna workspace skall du sedan skapa ett nytt projekt för varje uppgift i denna laboration. Börja därför nu med att lägga in ett nytt projekt. Kalla det nya projektet *uppgift1.m12*.

Kortet ML4 skall nu anslutas istället för borrmaskinen till mikrodatorsystemet. På kortet ML4 finns bl.a. en s.k. *DIP-switch*. Det är en enhet med åtta små switchar. Man kan från ett program avläsa switcharnas lägen via inporten ML4IN som har adressen 0x600. Porten har åtta bitar och varje bit motsvarar läget för en switch. Kortet ML4 har också en s.k. *Parallell output*. Det är en enhet med åtta lysdioder. Man kan tända och släcka lysdioderna i denna enhet genom att skriva till utporten ML4OUT vilken har adressen 0x400.

Uppgiften är att skriva ett program som avläser switcharnas lägen på ML4-kortet och som visar de avlästa lägena genom att skriva till *Parallell output*-enheten.

Programmet skall utformas som en “Stand alone” applikation. Detta betyder att du inte får använda *Standard startup* när du anger egenskaperna för ditt program i dialogrutan *Settings*. Du skall istället skriva en egen startsekvens i assembler. Denna sekvens skall anropa din `main`-funktion. Läsningen och skrivningen av ML4-portarna skall ske i `main`. Du skall skapa en speciell fil med namnet `ports.h`. Denna skall innehålla en typdefinition som anger portarnas typer samt definitioner av makron som beskriver portadresserna. (Gör på det sätt som visas i ”Arbetsbok för MC12”.) Inga explicita portadresser får användas i funktionen `main`. Ditt projekt skall alltså bestå av tre filer, filen `port.h`, en assemblerfil med startsekvensen samt en C-fil med funktionen `main`. Skapa dessa filer och lägg dem till projektet.

Innan du kompilerar ditt program skall du kontrollera att konfigurationen är satt till *Debug*. (Man bestämmer konfiguration med menyalternativet *Build* | *Configuration*. Om *Debug* är gråmarkerat är konfigurationen redan satt till *Debug*.) Testa din lösning i källkodssimulatorn i XCC. För att kunna göra detta måste du ansluta en ML4 *Dip-switch* och en ML4 *Parallell output*-enhet med hjälp av knappen *Simulator setup* på simulatorns verktygslist. Du skall inte ansluta något *Console*-fönster.

När du är i laborationslokalen och har tillgång till hårdvaran skall du kontrollera att kortet ML4 är anslutet till MC12-kortet och att strömtillförseln för ML4 är inkopplad. Ändra sedan konfiguration för ditt projekt till *Final* och välj *Build* | *Build all* på menyn. Välj därefter menyalternativet *Debug* | *Open Terminal*. Du får då ett terminalfönster i vilket du kan kommunicera med monitorn DBG12 som exekverar i MC12. Ladda ner ditt program till MC12. (Högerklicka i terminalfönstret.) Starta ditt program med *go*-kommandot.

När du fått denna uppgift att fungera har du en fungerande startsekvens som du kan återanvända i alla de följande uppgifterna i denna laboration. Du har också lärt dig hur man skapar projekt och hur man kör och konfigurerar simulatorn. Detta kommer du att ha nytta av i fortsättningen. Om du sitter i laborationslokalen har du dessutom fått kommunikationen med MC12 att fungera och du vet hur man laddar ner och startar sitt program.

Laborationsuppgift 5.2

Skapa nu ett nytt projekt i samma workspace som tidigare. Kalla det nya projektet *uppgift2.m12* och gör det till det aktiva projektet. Du kan börja med att lägga din assemblerfil med startsekvensen till det nya projektet. Resten av programkoden i projektet skall skrivas i C.

Uppgiften är att läsa från tangentbordet ML2 och visa det avlästa värdet på displayen ML3. Tangentbordet har 16 knappar numrerade från 0 till 15. När programmet körs skall det varje gång användaren trycker på någon av knapparna visa knappens nummer på displayen. Numret skall visas på decimal form. Programmet skall alltså inte avslutas när en knapp tryckts ner, utan det skall tillåta ett godtyckligt antal knapptryckningar.

Läsning från tangentbordet skall ske med hjälp av kortet ML15. På detta finns inporten *Key Decode Register* (adress 0x9C0). Beskrivning av bitarna på denna port finns i hjälptexten till XCC. Ditt program skall innehålla en inläsningsmodul bestående av filerna `keyboardML15.c` och `keyboardML15.h`. Modulen skall bara innehålla funktionen `get_key`. När denna anropas skall den vänta tills någon tangent tryckts ner. Därefter skall den som resultat ge numret på den nedtryckta tangenten.

Utskrift till displayen skall göras med hjälp av en modul bestående av filerna `displayML15.c` och `displayML15.h`. Denna modul skall innehålla två funktioner. Den första av dessa är `display_digits`. Denna får som parameter ett fält med sex element där varje element är en byte långt. (Använd typen `unsigned char`.) Varje element i fältet skall innehålla ett heltal i intervallet 0 till 9. Funktionen `display_digits` skall visa de sex talen i fältet i displayen ML3. Detta skall göras med hjälp av kortet ML15. ML15 har två utportar *Display Mode Register* och *Display Data Register*. Dessa ligger på adresserna 0x9C2 resp. 0x9C3. Du hittar beskrivning av ML3 och ML15 i hjälptexterna till XCC, men en liten förklaring av hur utskriften till *Display Data Register* går till behövs kanske ändå. Talen som skall visas på displayen skall skrivas till *Display Data Register* ett i taget. (De skall alltså skrivas till *samma* port i tur och ordning.) Därefter måste man skriva ytterligare två bytes till porten. Dessa skall båda innehålla värdet noll och de kommer inte att synas på displayen.

Den andra funktionen i displaymodulen skall heta `display_dec`. Den skall få en `unsigned int` som parameter. Dess uppgift är att visa parameterns värde på displayen i decimal form. Detta gör den naturligtvis genom att plocka ut de sista sex siffrorna i parametern en och en, placera dem i ett fält med bytes och därefter anropa funktionen `display_digits`.

Definitioner av makron för portadresserna skall läggas i filen `ports.h` från uppgift 5.1.

Testa ditt program i källkodssimulatorn. När du ansluter enheterna ML2 och ML3 till simulatorn skall du vara noga med att för båda välja *Interface* | *ML15*. Innan du kör ditt program på MC12 i laborationslokalen bör du kontrollera att kortet med tangentbordet och displayen är anslutet till kortet ML15.

När du klarat av detta steg har du två moduler med vilkas hjälp du kan läsa indata respektive visa utdata. Du kommer att behöva båda modulerna i de övriga stegen i laborationen.

Laborationsuppgift 5.3

Skapa ytterligare ett nytt projekt i samma workspace som tidigare. Kalla det nya projektet *uppgift3.m12* och gör det till det aktiva projektet. Även denna gång kan du börja med att lägga din assemblerfil med startsekvensen till det nya projektet.

I kurslitteraturen beskrivs en CRG-krets som kan generera periodiska avbrott. I detta steg skall du skapa en klockmodul som använder sig av dessa avbrott. Din klockmodul skall bestå av filerna `clock.h` och `clock.c`. Modulen skall internt innehålla en räknare (en klocka) som räknar antalet avbrott som skett. Eftersom antalet avbrott kan bli stort duger inte typen `int`. Definiera istället en egen typ, `time_type`, som är lika med `unsigned long int` och låt räknaren ha denna typ. Räknaren skall deklareras på sådant sätt att den inte kan påverkas från någon funktion som ligger utanför klockmodulen. Den bör också markeras som flyktig, eftersom den ändras varje gång ett avbrott inträffar och detta inte syns i den "vanliga" koden. I klockmodulen skall det finnas fyra funktioner:

- `init_clock`. Nollställer klockan och initierar CRG-kretsen så att den genererar ett avbrott ungefär var 10:e ms.
- `clock_inter`. Anropas av avbrottsrutinen varje gång ett avbrott inträffar. Tickar upp klockan.
- `get_time`. Ger som resultat det ungefärliga antalet ms som gått sedan klockan initierades. Observera att resultatet *inte* skall vara antalet avbrott utan antalet ms. Resultattypen skall vara `time_type`.
- `hold`. Innehåller en repetitionssats som fördröjer exekveringen ett visst antal ms. Får som parameter ett heltal av typen `time_type` som anger hur många ms fördröjningen skall vara. Funktionen skall använda sig av avbrottsräknaren i klockmodulen för att avgöra hur länge fördröjningen skall vara.

För att lösa denna uppgift måste du skriva en avbrottrutin i assembler vilken anropar funktionen `clock_inter`. Du måste också initiera avbrottsvektorn för CRG-kretsen så att din avbrottsrutin anropas. Tänk också på att man i MC12 måste nollställa I-flaggan för att avbrott skall tillåtas. Detta måste göras i en assemblerrutin som anropas från `init_clock`.

Observera att du i denna uppgift inte får använda några icke-standardiserade specialegenskaper för kompilatorn XCC, såsom inbäddad assemblerkod eller speciella markörer för funktioner. Endast sådant som kan skrivas i standard-C är tillåtet. Det som inte går att göra i standard-C skall göras i separata assemblerrutiner.

Alla macron för att definiera portar och avbrottsvektorer skall förstås läggas i filen `ports.h` från de tidigare stegen.

När klockmodulen är klar skall du testa den genom att skriva ett program som visar en sekundräknare på display-enheten ML3. (Använd displaymodulen från uppgift 5.2.) Värdet på displayen skall alltså ökas med ett varje sekund (ungefär).

När du skall testa din lösning i källkodssimulatorn finns ett litet problem. Avbrotten genereras där mycket långsammare än när man kör programmet "på riktigt". För att det skall bli möjligt att testa programmet i simulatorn måste du därför sätta CRG-kretsens avbrottsintervall till det kortast möjliga värdet. Ett bra tips är också att i funktionen `hold` bara vänta tills nästa avbrott kommer. För att du inte skall behöva ändra i programkoden när du växlar mellan simulatorversionen och den slutliga versionen av programmet kan du använda dig av s.k. villkorlig kompilering (se kursboken). Du kan t.ex. i dialogrutan D*ebug* | *Settings* för simulatorversionen i rutan *DEFINES* för C-kompilatorn lägga till macron SIMULERING. Denna kan du sedan testa på i programkoden. När du nu fått klockmodulen att fungera har du alla verktyg du behöver för att på ett enkelt sätt klara av det sista steget i laborationen.

Laborationsuppgift 5.4

Skapa ett nytt projekt i samma workspace som tidigare. Kalla det nya projektet *uppgift4.m12* och gör det till det aktiva projektet.

I denna uppgift skall du skriva ett program som hanterar borrmaskinen du stiftade bekantskap med i laborationen i assembler. Det program du skall skriva skall göra samma sak som det assemblerprogram som diskuterades där. Skillnaden är att programmet nu skall skrivas i C. De enda undantagen från detta är de tre assemblerrutiner du redan konstruerat i de tidigare uppgifterna i denna laboration (startsekvensen, avbrottsrutinen och funktionen som nollställer I-flaggan).

Funktionerna som styr borrmaskinen skall samlas i en modul bestående av de två filerna `drill.c` och `drill.h`. Modulen skall bland annat innehålla följande funktioner:

- `void MotorStart( void )`
- `void MotorStop( void )`
- `void DrillDown( void )`
- `void DrillUp( void )`
- `int Nstep( int )`
- `int DrillDownTest( void )`
- `void Alarm( int )`
- `DrillHole`
- `int RefPos( void )`
- `void DoAuto( void )`

Dessa funktioner skall utföra exakt samma ting som motsvarande assemblerfunktioner, men de skall vara skrivna i C. De ska implementeras med parameterlistor och returtyper enligt ovan. Observera att funktionsnamnet `Auto` är olämpligt att använda i C-programmet (varför då?).

Följande funktioner ska också konstrueras och testas för att därefter *konsekvent användas* då styrord ges till borrmaskinen:

- `void Outzero( int bit );`
- `void Outone( int bit );`

Specifikationen för dessa är den samma som i laboration 1.

När du konstruerar funktionerna för borrmaskinen behöver du ibland lägga in fördröjningar. Då skall du använda dig av klockmodulen från uppgift 5.3. Glöm inte att initiera denna i `main`. Nu skall borrmaskinen kopplas in igen, på samma sätt som i assemblerlaborationen. Den skall styras via utporten som har adressen 0x400. Naturligtvis skall alla makrodefinitioner läggas i filen `ports.h`. Använd ett skuggregister vid skrivning till porten.

Ditt program skall läsa från tangentbordet och om någon av tangenterna 0-7 trycktes ner skall en av funktionerna `start`, `stop`, `down`, `up`, `step`, `drill`, `refpo` resp. `auto_drill` anropas. Om någon annan tangent tryckts ner skall inget utföras. (Här är det lämpligt att använda en `switch`-sats.) För att läsa från tangentbordet använder du förstås inläsningsmodulen från uppgift 5.2.

Appendix: MC12 IO-adresser för laborationskort

Laborationskort Noter	Register/Port Symboliska namn	Adress (hexadecimal form)	Simulator (lämpligt val)
ML4			
"Borrmaskin" är också ansluten till dessa adresser. OBS: Skillnad mellan IO-simulator och fysisk hårdvara.	Out	0400	0400
	In	0600	0401
ML5			
De här angivna adresserna för ML5 gäller PAL-revision 2.	Out	0C00	0C00
	In	0C01	0C01
	Out	0C02	0C02
	Out	0C03	0C03
ML13			
	Ctrl/Status	0B00	0B00
	IRQ Ctrl/Status	0B01	0B01
ML15			
	Kbd Data	09C0	09C0
	Kbd Status	09C1	09C1
	Led Mode	09C2	09C2
	Led Ctrl/Data	09C3	09C3
ML19			
	Status	0DC0	0DC0
	Kvittera händelse 1	0DC2	0DC2
	Kvittera händelse 2	0DC3	0DC3