Maskinorienterad programmering 2013/2014

Sammanfattning

Kursens syften är

- att vara en introduktion till konstruktion av små inbyggda system och
- att ge en förståelse för hur imperativa styrstrukturer översätts till assembler
- att ge en förståelse för de svårigheter som uppstår vid programmering av händelsestyrda system med flera indatakällor.

Vi repeterar kursens ”lärandemål”

1. Programutveckling i C och assemblerspråk

Kunna utföra programmering i C och assemblerspråk samt kunna:

- beskriva och tillämpa modularisering med hjälp av funktioner och subrutiner.
- beskriva och tillämpa parameteröverföring till och från funktioner.
- beskriva och tillämpa olika metoder för parameteröverföring till och från subrutiner.
- beskriva och använda olika kontrollstrukturer.
- beskriva och använda sammansatta datatyper (fält och poster) och enkla datatyper (naturliga tal, heltal och flyttal).

- *beskriva och tillämpa modularisering med hjälp av funktioner och subrutiner.*

EXEMPEL

```
callfunc( int aa , int ab )
{
    aa = 1;
    ab = 2;
}
```

XCC12 genererar följande kod:

```
    SEGMENT   text
    EXPORT    _callfunc [r,2]
_callfunc:
;       2 | {
;       3 |     aa = 1;
    LDD    #1
    STD    2,SP
;       4 |     ab = 2;
    LDD    #2
    STD    4,SP
;       5 | }
    RTS
```

Funktioners parametrar och returvärden.

Subrutiner för att manipulera styrregistret OUTONE och OUTZERO

```
* Subrutin OUTONE. Läser kopian av
* borrmaskinens styrord på adress
* DCCopy. Ettställer en av bitarna och
* skriver det nya styrordet till
* utporten DCTRL samt tillbaka till
* kopian DCCopy.
* Biten som nollställs ges av innehållet
* i B-registret (0-7) vid anrop.
* Om (B) > 7 utförs ingenting.
* Anrop:                 LDAB    #bitnummer
*                        JSR     OUTONE
* Utdata:                Inga
* Registerpåverkan:      Ingen
* Anropade subrutiner:   Inga
```

”bitnummer” = 0..7

7	6	5	4	3	2	1	0

Lagrings-klass och synlighet.

Kompilera följande deklarationer till assembler och studera assemblerfilen. Vilken skillnad upptäcker du?

```
int       a;
static    int    b;
```

```
;       1 | int           a;
    SEGMENT   bss
_a:     RMB    $2
    EXPORT     _a      [r,2]

;       2 | static        int    b;
_1:     RMB    $2
(symbolen _1 existerar endast under
assemblering och motsvarar då
symbolen 'b' i programmet. Symbolen
'b' exporteras inte.
```

```
**************************************
* SUBRUTIN - DELAY
* Beskrivning: Skapar en fördröjning om
* ANTAL x 500 ms.
* Anrop: LDAA  #6  Fördröj 6*500ms= 3s
*      JSR   DELAY
* Indata:Antal intervall,om 500 ms i A
*
* Utdata: Inga
* Register-påverkan: Ingen
* Anropad subrutin: Ingen.
**************************************
```

- *beskriva och tillämpa olika metoder för parameteröverföring till och från subrutiner.*

Parameteröverföring via register

Antag att vi alltid använder register `D,X,Y` (i denna ordning) för parametrar som skickas till en subrutin. Då kan funktionsanropet (subrutinanropet)

```
dummyfunc(la,lb,lc);
```

översättas till:

```
LDD     la
LDX     lb
LDY     lc
BSR     dummyfunc
```

Då vi kodar subrutinen `dummyfunc` vet vi (på grund av våra regler) att den första parametern skickas i D, den andra i X och den tredje i Y (osv).

Metoden är enkel och ger bra prestanda.
Begränsat antal parametrar kan överföras.

Parameteröverföring via stacken

Antag att listan av parametrar som skickas till en subrutin behandlas från höger till vänster. Då kan

```
dummyfunc(la,lb,lc);
```

Översättas till:

```
        LDD     lc
        PSHD
;(alternativt  STD    2,-SP)
        LDD     lb
        PSHD
        LDD     la
        PSHD
        BSR     dummyfunc
        LEAS    6,SP
```

Innehåll	Kommentar	Adressering via SP i subrutinen
lc.lsb	Parameter lc	
lc.msb		6,SP
lb.lsb	Parameter lb	
lb.msb		4,SP
la.lsb	Parameter la	
la.msb		2,SP
PC.lsb	Återhoppsadress, placeras här vid BSR	
PC.msb		0,SP

```
dummyfunc:
        . .
        LDD     2,SP
; parameter la till register D
        . .
        LDD     4,SP
; parameter lb till register D
        . .
        LDD     6,SP
;parameter lc till register D
```

Returvärden via register

Register väljs, beroende på returvärdets typ (storlek), HCS12-exempel

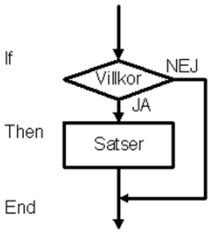
Storlek	Benämning	C-typ	Register
8 bitar	byte	char	B
16 bitar	word	short int	D
32 bitar	long	long int	Y/D

En regel (konvention) bestäms och följs därefter vid kodning av samtliga subrutiner

- *beskriva och använda olika kontrollstrukturer.*

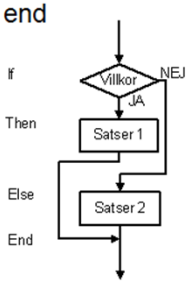
Kontrollstrukturer

If(Villkor) then ...



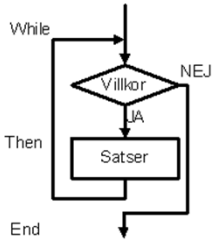

```
if(Villkor)
{
  Satser;
}
```

if(Villkor) then ... else ... end



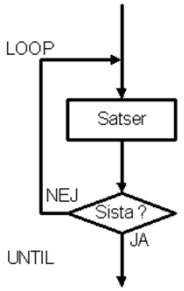

```
if(Villkor)
{
  Satser1;
}else{
  Satser2;
}
```

while(Villkor) loop



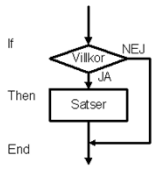

```
while(Villkor)
{
  Satser;
}
```

loop ... until(Villkor)




```
do{
  Satser;
}
while(Villkor);
```

If (...) {...}



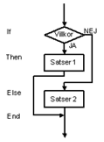

```
if (DipSwitch != 0)
   HexDisp = Dipswitch;
```

Bättre kodning...

```
DipSwitch   EQU   $600
HexDisp     EQU   $400
            ...
            TST   DipSwitch
            BEQ   end

            LDAB  DipSwitch
            STAB  HexDisp
end:
```

BNE	"Hopp" om ICKE *zero*	Z=0
BEQ	"Hopp" om *zero*	Z=1

If (...) {...} else { ...}

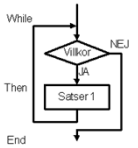


```
if (DipSwitch == 0)
      HexDisp = 1;
else
      HexDisp = 0;
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400
            ...
            LDAB  DipSwitch
            ...
            TSTB
            BEQ   not_else
            LDAB  #0
            STAB  HexDisp
            BRA   end
not_else:   LDAB  #1
            STAB  HexDisp
end:
```

BEQ	"Hopp" om *zero*	Z=1

while (...) {...}




```
Delay( unsigned int count )
{
  while (count > 0)
    count = count - 1;
}
```

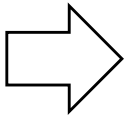
```
Delay:      LDD   "count"
Delay_loop:
            NOP
            ...
            NOP
            SUBD  #1
            BHI   Delay_loop
Delay_end:  RTS
```

Test av tal utan tecken		
BHI	Villkor: R>M	C + Z = 0

Test av tal med tecken		
BGT	Villkor: R>M	Z + (N ⊕ V) = 0

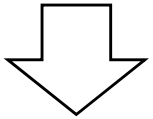
- *beskriva och använda sammansatta datatyper (fält och poster) och enkla datatyper (naturliga tal, heltal och flyttal).*

```
/*
   globals.c
   Deklaration av globala variabler
*/

short     shortint;
long      longint;
int    justint;
int    intvec[10];

struct {
   int    s1;
   char      s2;
   char*     s3;
} komplex;
```



```
;     1 | short     shortint;
   SEGMENT   bss
_shortint:   RMB    $2
   EXPORT    _shortint [r,2]

;     2 | long      longint;
_longint:    RMB    $4
   EXPORT    _longint  [r,4]

;     3 | int       justint;
_justint:    RMB    $2
   EXPORT    _justint  [r,2]

;     4 | int       intvec[10];
_intvec:     RMB    $14
   EXPORT    _intvec   [r,20]

;     5 |
;     6 | struct {
;     7 |    int    s1;
;     8 |    char      s2;
;     9 |    char*     s3;
;    10 | } komplex;
_komplex:    RMB    $5
   EXPORT    _komplex  [r,5]
```

```
main() {
   short  shortint;
   long      longint;
   int    justint;

   struct {
      int    s1;
      char      s2;
      char      *s3;
   } typen;
   justint = 0;
}
```



```
   SEGMENT   text
   EXPORT    _main  [r,2]
_main:
   LEAS   -13,SP
;     2 |    short  shortint;
;     3 |    long      longint;
;     4 |    int    justint;
;     5 |
;     6 |    struct {
;     7 |       int    s1;
;     8 |       char      s2;
;     9 |       char      *s3;
;    10 |    } typen;
;    11 | justint = 0;
   CLRA
   CLRB
   STD    5,SP
;    12 | }
   LEAS   13,SP
   RTS
```

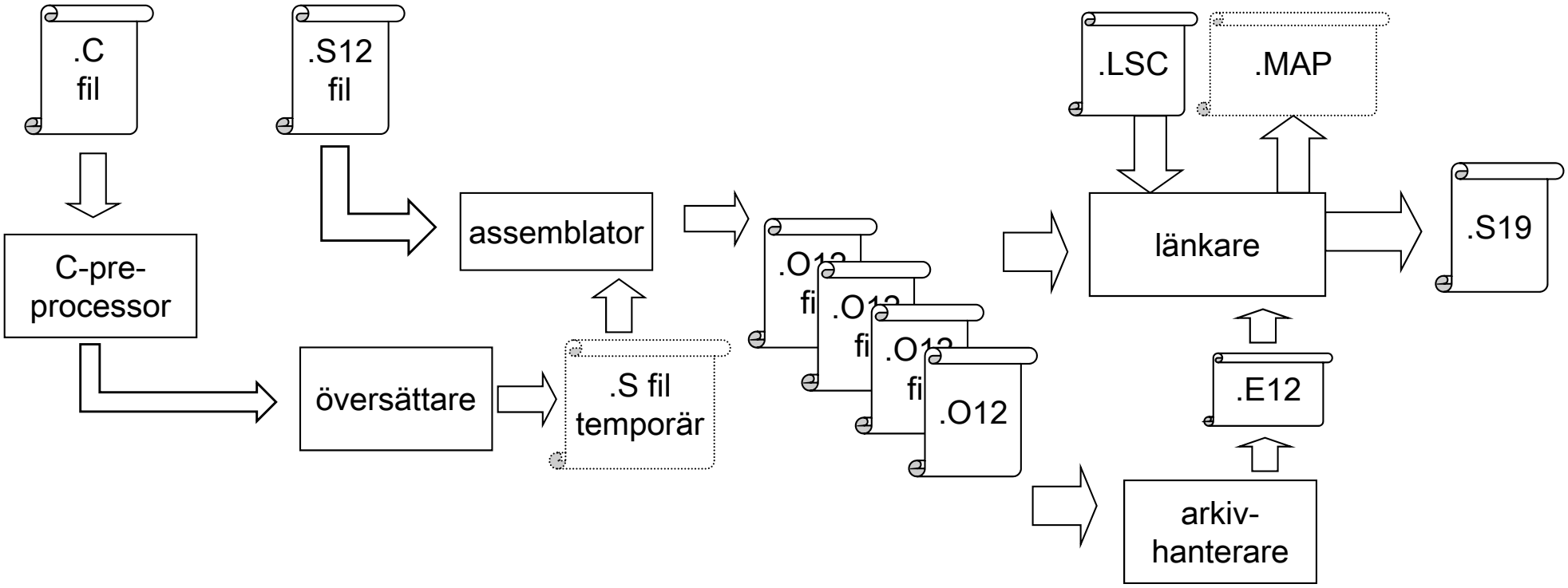
Kunna redogöra för olika lagringsklasser (GLOBAL, STATIC, LOCAL) och ”synlighet”.

2. Programutvecklingteknik

Att kunna:

- beskriva översättningsprocessen, dvs. assemblatorns arbetssätt, preprocessorns användning, separatkompilering och länkning.
- konstruera, redigera och översätta (kompilera och assemblera) program
- testa, felsöka och rätta programkod med hjälp av avsedda verktyg.

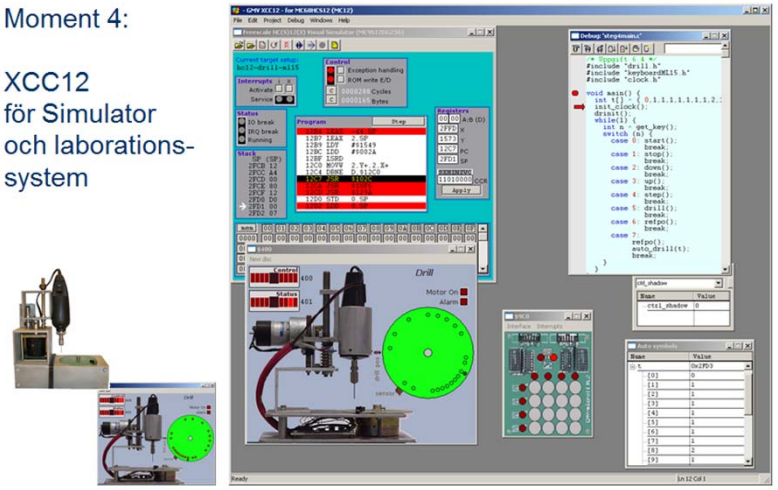
- *beskriva översättningsprocessen, dvs. assemblatorns arbetssätt, preprocessorns användning, separatkompilering och länkning.*



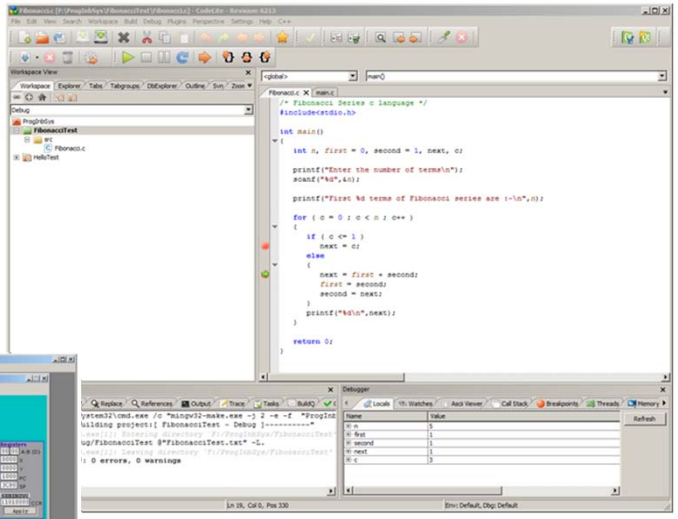
- *konstruera, redigera och översätta (kompilera och assemblera) program*
- *testa, felsöka och rätta programkod med hjälp av avsedda verktyg.*

Moment 4:

XCC12
för Simulator
och laborations-
system

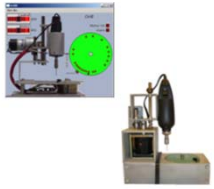


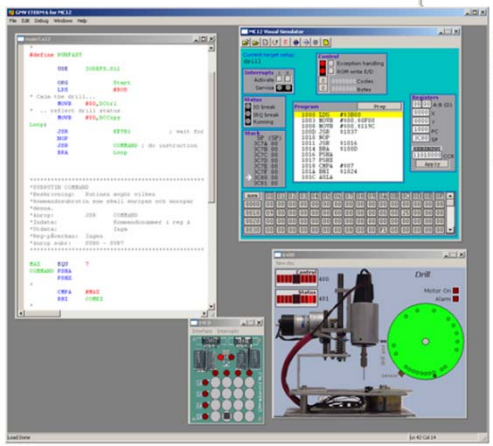
CodeLite för
moment 3
”Länkad lista”.



Moment 1 och 2:

ETERM för
Simulator och
laborationssystem





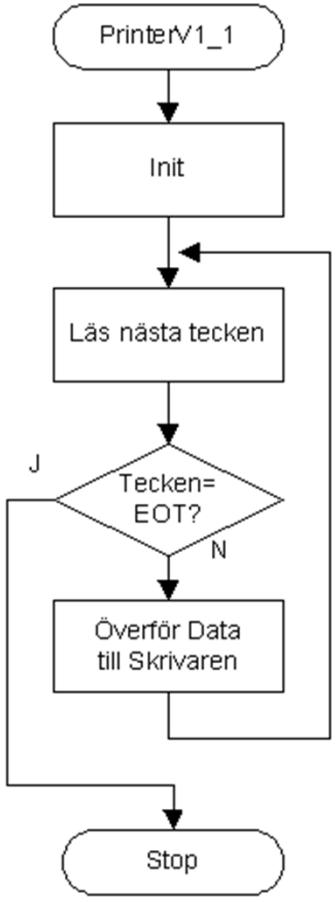
Dessa lärandemål har vi kontrollerat under laborationer.

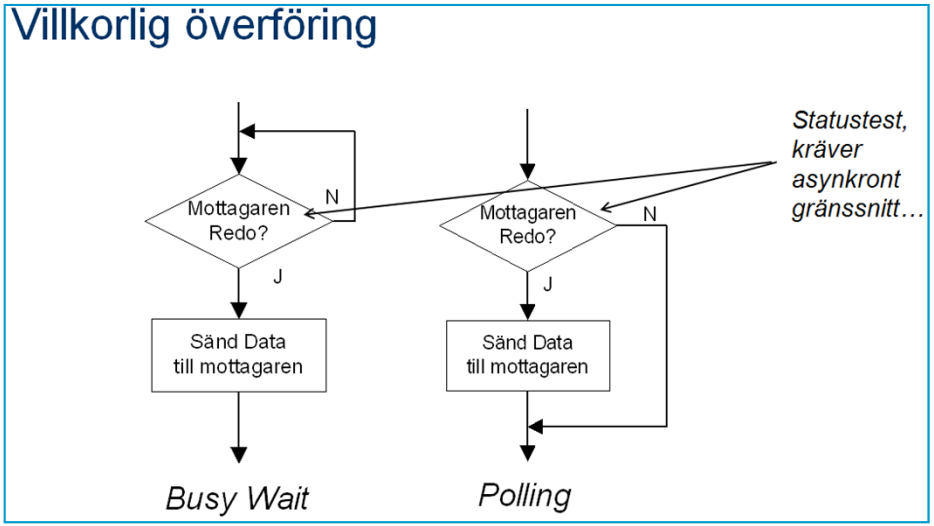
3. Systemprogrammerarens bild

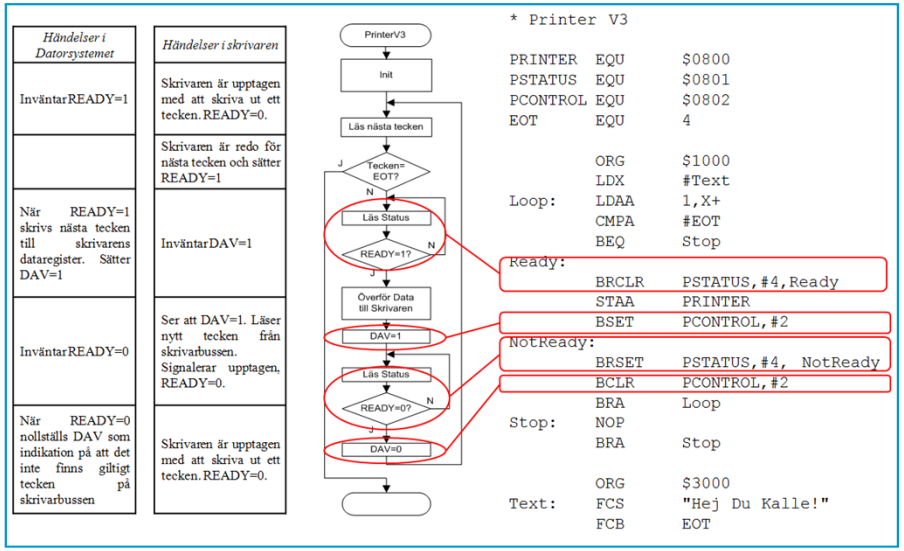
Att kunna:

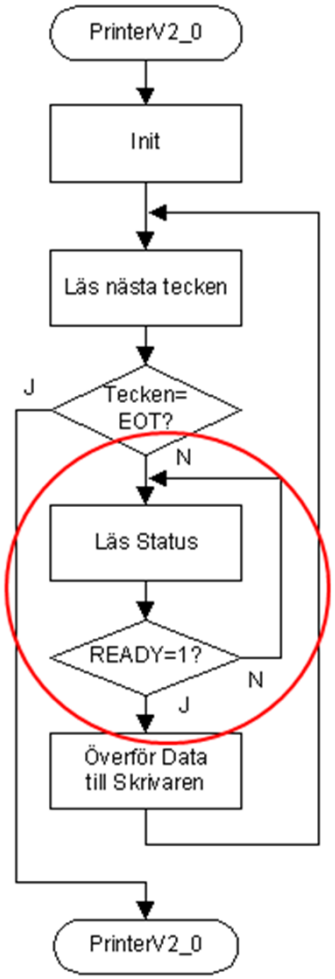
- beskriva och tillämpa olika principer för överföring mellan centralenhet och kringenheter så som: ovillkorlig eller villkorlig överföring, statustest och rundfrågning.
- konstruera program för systemstart och med stöd för avbrottshantering från olika typer av kringenheter.
- beskriva och exemplifiera olika undantagstyper: interna undantag, avbrott och återstart samt prioritetshantering vid undantag.
- kunna beskriva metoder och mekanismer som är centrala i systemprogramvara så som pseudoparallell exekvering och hantering av processer.
- beskriva och använda kretsar för tidmätning.
- beskriva och använda kretsar för parallell respektive seriell överföring.

- *beskriva och tillämpa olika principer för överföring mellan centralenhet och kringenheter så som: ovillkorlig eller villkorlig överföring, statustest och rundfrågning.*









- *konstruera program för systemstart och med stöd för avbrottshantering från olika typer av kringenheter.*

Exempel 4.43 Placering av Exceptionvektorer, assemblerkod

Följande programskelett illustrerar hur några avbrottsrutiner respektive avbrottsvektorer kan definieras i en fristående HCS12-applikation.

```
      ORG       $FFF2
      FDB       irq_service_routine
      FDB       xirq_service_routine
      FDB       software_interrupt_service_routine
      FDB       illegal_opcode_service_routine
      FDB       cop_service_routine
      FDB       clock_monitor_fail_service_routine
      FDB       Application_Start

; Symbolen "Application_Start_Address" kan vara godtycklig.
      ORG       Application_Start_Address
Application_Start:
      LDS       #TopOfStack
      ...
      ...
      ANDCC     #$FE      ; nollställ I-flagga
      JSR       _main
```

Vår slutliga "appstart" blir nu:

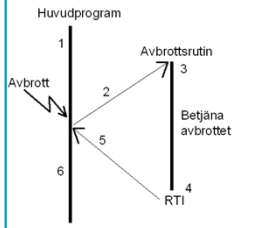
```
        segment    init
        export     _exit
        import     _main
        function   __start,__start_end
*   Här börjar exekveringen...
__start
        LDS        #$2FFF
        JSR        _main
_exit: NOP
        BRA        _exit
__start_end
```

_startup-o.c

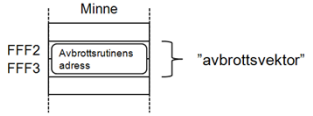
```
/* ****************************************************************************
        STANDARD STARTUP ROUTINE
**************************************************************************** */
/* Force following code to named segment "init" (see linker script) */
#pragma TEXT    init
__interrupt void        _start( void )
{
        __asm(" LDS #%a", STACK );      /* Get a valid stackpointer */
        __hwinit();                     /* Chip init */
        __crtInit( BOTTOM_OF_STACK );   /* Initialize C-library */
        main();                         /* Call user application */
}
```

- *beskriva och exemplifiera olika undantagstyper: interna undantag, avbrott och återstart.*

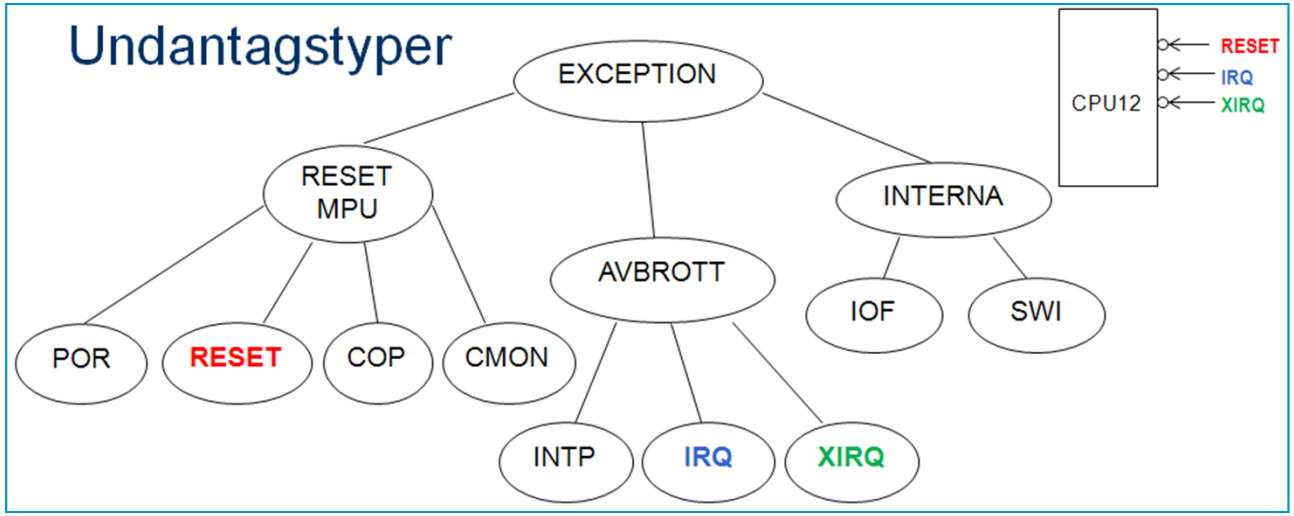
Avbrott



1) Huvudprogram exekveras när ett avbrott aktiveras
2) Hopp till avbrottsrutin
3) Avbrottsrutin startar
4) Avbrottsrutin avslutas med en speciell instruktion, *return from interrupt* (RTI)
5) Återhopp till huvudprogram
6) Huvudprogrammet fortsätter.

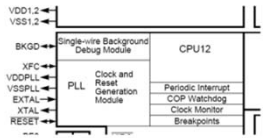


Undantagstyper



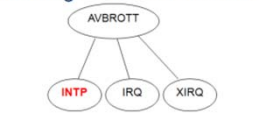
Fatala fel, kräver RESET av CPU

POR, *Power On Reset*, vid spänningstillslag
RESET, insignal till processorn aktiveras.
COP, *Computer Operating Properly*, så kallad *watchdog-funktion*.
CMON, *Clock Monitor Reset*, övervakar E-klockan, om frekvensen sjunker under 10 kHz genereras RESET.

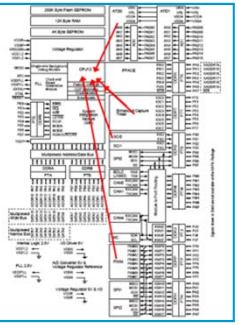


Adress (hex)	Funktion
FFFE	RESET, Startvektor
FFFC	Clock Monitor Fail
FFFA	COP Watchdog Timeout
FFF8	Illegal Op Code
FFF6	SWI
FFF4	XIRQ
FFF2	IRQ
FF00-FFF0	Enhetsspecifika vektorer, skiljer sig något beroende på olika varianter

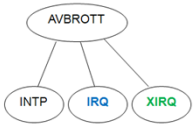
Internt genererade avbrott

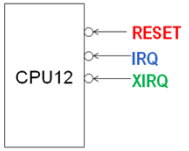


Adress (hex)	Funktion
FFF0	Real Time Interrupt
FFEE	Enhanced Capture Timer channel
FFEC	Enhanced Capture Timer channel 1
FFEA	Enhanced Capture Timer channel 2
FF8E	Port P Interrupt
FF8C	PWM Emergency Shutdown
FF8A-FF80	Reserverade



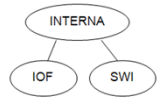
Externt genererade avbrott





Adress (hex)	Funktion
FFFE	RESET, Startvektor
FFFC	Clock Monitor Fail
FFFA	COP Watchdog Timeout
FFF8	Illegal Op Code
FFF6	SWI
FFF4	XIRQ
FFF2	IRQ
FF00-FFF0	Enhetsspecifika vektorer, skiljer sig något beroende på olika varianter

Interna undantag



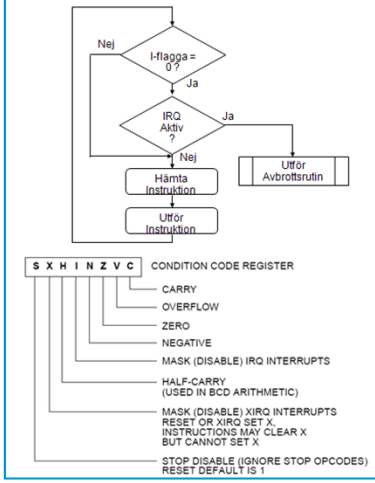
Om processorn avkodar en otillåten operationskod kallas detta *Illegal Opcode Fetch* (IOF).
Processorn avbryter då, ***sparar registerinnehåll på stacken***, läser autovektorn för IOF och utför undantagshantering.

Instruktionen *SoftWare Interrupt* (SWI) fungerar på samma sätt, men har en annan autovektor och en bestämd operationskod.

Adress (hex)	Funktion
FFFE	RESET, Startvektor
FFFC	Clock Monitor Fail
FFFA	COP Watchdog Timeout
FFF8	Illegal Op Code
FFF6	SWI
FFF4	XIRQ
FFF2	IRQ
FF00-FFF0	Enhetsspecifika vektorer, skiljer sig något beroende på olika varianter

- *beskriva och tillämpa olika metoder för prioritetshantering vid multipla avbrottskällor (mjukvarubaserad och hårdvarubaserad prioritering, avbrottsmaskering, icke-maskerbara avbrott).*

Maskering av avbrott



Maskera avbrott:

```
SEI
```

Alternativt

```
ORCC  #%00010000
```

Demaskera avbrott:

```
CLI
```

Alternativt

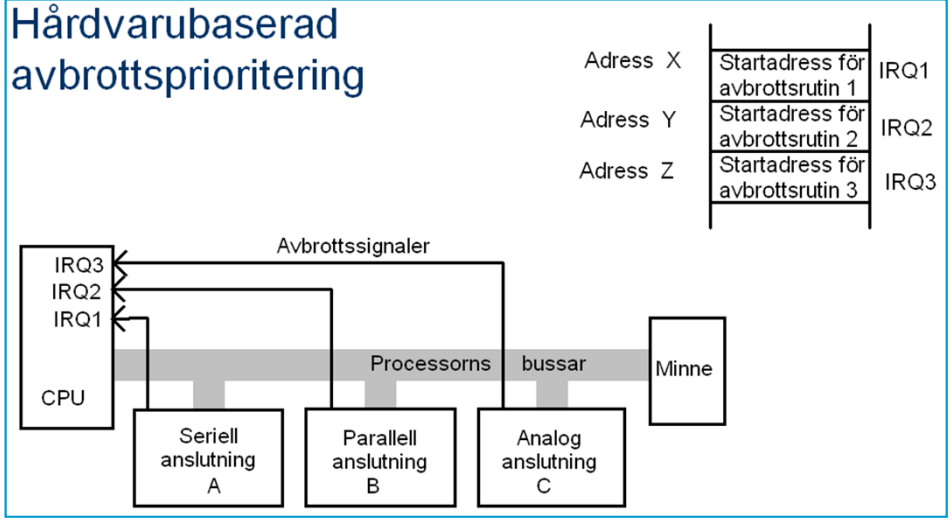
```
ANDCC   #%11101111
```

Demaskera X-avbrott:

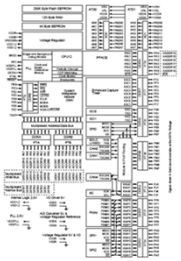
```
ANDCC   #%10111111
```

OBS: Kan INTE maskeras ("Non Maskable Interrupt")

Hårdvarubaserad avbrottsprioritering



Intern avbrottsprioritering



För avbrott från interna kretsar bestäms prioriteten av avbrottsvektorns adress.

Ju högre adress, desto högre prioritet.

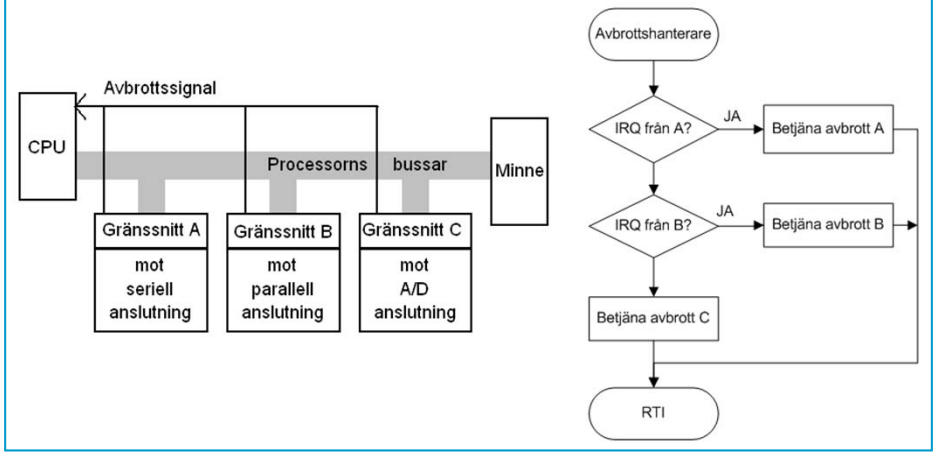
Det finns vissa möjligheter att ändra detta programmässigt.

Högre prioritet

Lägre prioritet

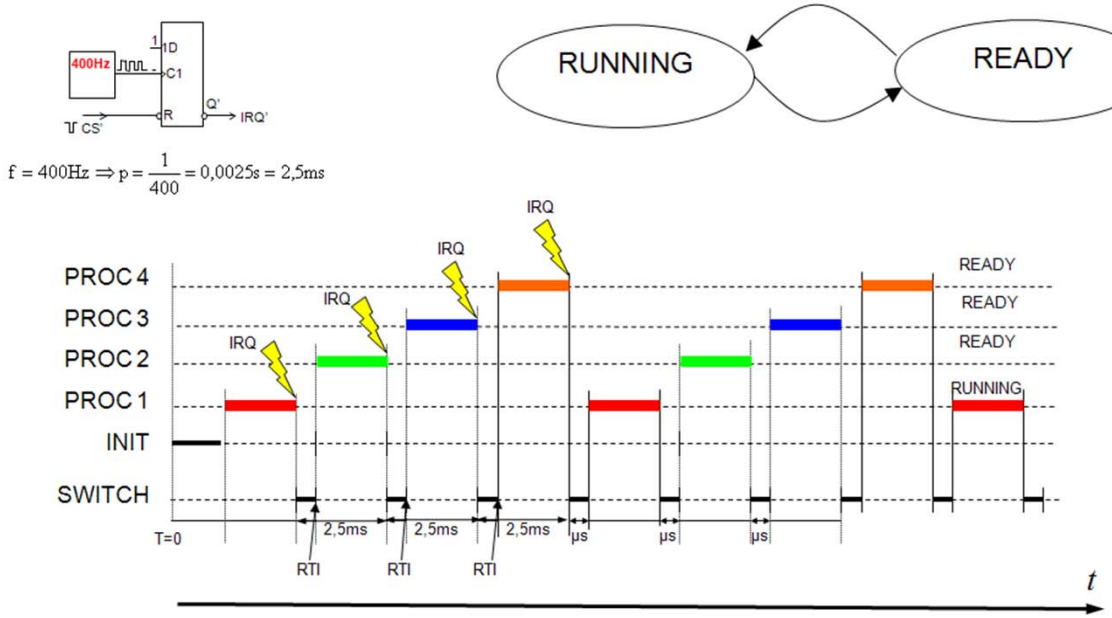
Adress (hex)	Funktion
FFF0	Real Time Interrupt
FFEE	Enhanced Capture Timer channel
FFEC	Enhanced Capture Timer channel 1
FFEA	Enhanced Capture Timer channel 2
FFE8	Enhanced Capture Timer channel 3
FFE6	Enhanced Capture Timer channel 4
FFE4	Enhanced Capture Timer channel 5
FFE2	Enhanced Capture Timer channel 6
FFE0	Enhanced Capture Timer channel 7
FFDE	Enhanced Capture Timer overflow
FFDC	Pulse accumulator A overflow
FFDA	Pulse accumulator input edge
FFD8	SPI0
FFD6	SCI0
FFD4	SCI1
FFD2	ATD0
FFD0	ATD1
FFCE	Port J
FFCC	Port H
FFCA	Modulus Down Counter underflow
FFC8	Pulse Accumulator B Overflow
FFC6	PLL lock
FFC4	CRG Self Clock Mode
FFC2	Används ej (BDLC)
FFC0	IIC Bus
FFBE	SPI1
FFBC	Reserverad
FFBA	EEPROM I-Bit
FFB8	FLASH I-Bit
FFB6	CAN0 wake-up
FFB4	CAN0 errors
FFB2	CAN0 receive
FFB0	CAN0 transmit
...	...
FF96	CAN4 wake-up
FF94	CAN4 errors
FF92	CAN4 receive
FF90	CAN4 transmit
FF8E	Port P Interrupt
FF8C	PWM Emergency Shutdown
FF8A-FF80	Reserverade

Programbaserad avbrottsprioritering



- *kunna beskriva metoder och mekanismer som är centrala i systemprogramvara så som pseudoparallell exekvering och hantering av processer.*

Processtillstånd

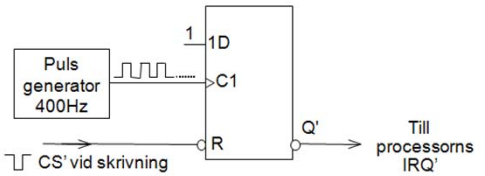


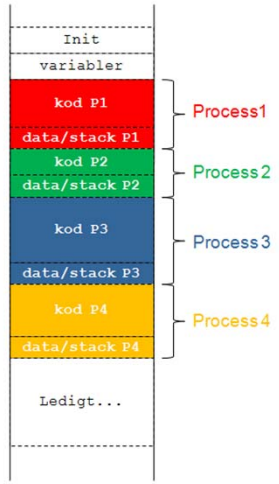
Processbyte

En processor
– flera program
– körs ”samtidigt” (pseudoparallellt)

HDW krav: En avbrottskälla som ger regelbundna avbrott (Ex Timer)

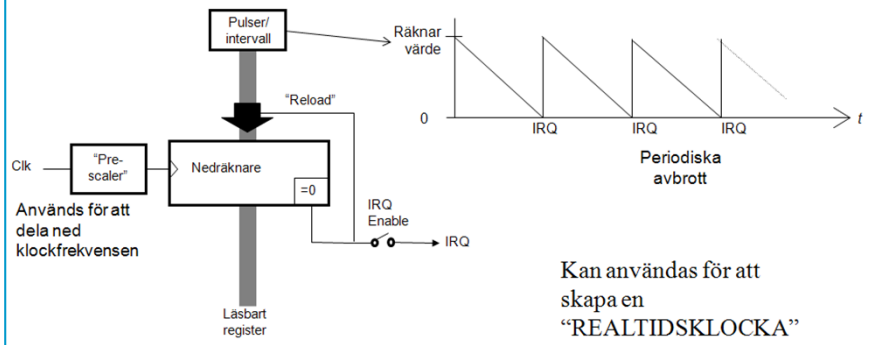
SW krav: En avbrottsrutin (SWITCH) som växlar process





- *beskriva och använda kretsar för tidmätning.*

Räknarkrets (”timer”), principiell funktion



Kan användas för att skapa en “REALTIDSKLOCKA”

.. Program för initiering..

```
; Adressdefinitioner
CRGINT   EQU      $38
RTICTL   EQU      $3B

timer_init:
; Initiera RTC avbrottsfrekvens
; Skriv tidbas för avbrottsintervall till RTICTL
         MOVB     #$49,RTICTL
; Aktivera avbrott från CRG-modul
         MOVB     #$80,CRGINT
         RTS
```

Anmärkning: Det är olämpligt att använda detta värde då programmet testas i simulator, använd då i stället det kortast tänkbara avbrottsintervallet enligt;

```
; Skriv tidbas för avbrottsintervall till RTICTL
         MOVB     #$10,RTICTL       ; För simulator
```

Realtidsklocka i HCS12

Address Offset	Use	Access
$_00	CRG Synthesizer Register (SYNR)	R/W
$_01	CRG Reference Divider Register (REFDV)	R/W
$_02	CRG Test Flags Register (CTFLG)[1]	R/W
$_03	CRG Flags Register (CRGFLG)	R/W
$_04	CRG Interrupt Enable Register (CRGINT)	R/W
$_05	CRG Clock Select Register (CLKSEL)	R/W
$_06	CRG PLL Control Register (PLLCTL)	R/W
$_07	CRG RTI Control Register (RTICTL)	R/W
$_08	CRG COP Control Register (COPCTL)	R/W
$_09	CRG Force and Bypass Test Register (FORBYP)[2]	R/W
$_0A	CRG Test Control Register (CTCTL)[3]	R/W
$_0B	CRG COP Arm/Timer Reset (ARMCOP)	R/W

Tre olika register används för realtidsklockan

NOTES:
1. CTFLG is intended for factory test purposes only.
2. FORBYP is intended for factory test purposes only.
3. CTCTL is intended for factory test purposes only.

Realtidsklocka i HCS12, avbrottshantering

```
; Adressdefinition
CRGFLG  EQU     $37

timer_interrupt:
; Kvittera avbrott från RTC
        BSET    CRGFLG,#$80
        RTI

; Avbrottsvektor på plats…
        ORG     $FFF0
        FDB     timer_interrupt
```

Adress (hex)	Funktion
FFF0	Real Time Interrupt
FFEE	Enhanced Capture Timer channel
FFEC	Enhanced Capture Timer channel 1
FFEA	Enhanced Capture Timer channel 2
....	....
FF8E	PortP Interrupt
FF8C	PWM Emergency Shutdown
FF8A-FF80	Reserverade

- *beskriva och använda kretsar för parallell respektive seriell överföring.*

Multiplexed External Bus Interface (MEBI)										
Offset		7	6	5	4	3	2	1	0	Mnemonic
$00	R W	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	PORTA
$01	R W	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	PORTB
$02	R W	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	DDRA
$03	R W	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	1=OUT 0=IN	DDRB
$04	R W	.....								

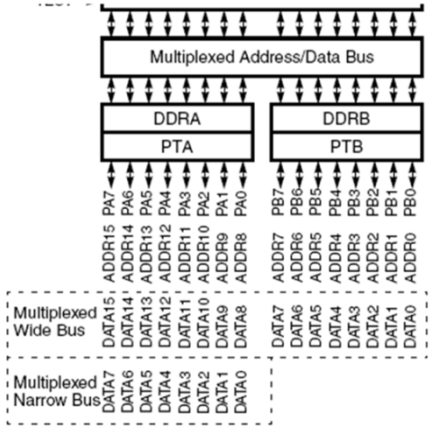
Figur 4.5: Register för Port A/B som generell IO

Exempel 4.50

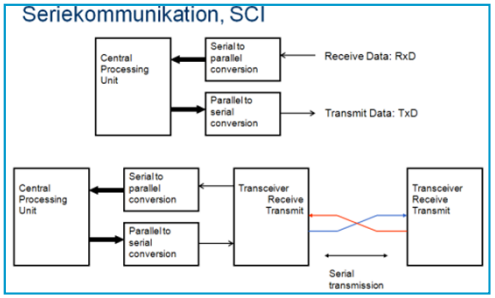
Ange i såväl assemblerspråk som C, programkonstruktioner som initierar port A för användning som inport samt port B för användning som utport.

Lösning:

```
PORTA       EQU   0
PORTB       EQU   1
DDRA        EQU   2
DDRB        EQU   3
      ...
            CLR   DDRA
            MOVB  #$FF,DDRB
      ...

typedef struct sMEBI{
      volatile unsigned char porta;
      volatile unsigned char portb;
      volatile unsigned char ddra;
      volatile unsigned char ddrb;
}MEBI, *PMEBI;

#define     MEBI_BASE   0

( ( ( PMEBI )( MEBI_BASE ))-> ddra ) = 0;
( ( ( PMEBI )( MEBI_BASE ))-> ddrb ) = 0xFF;
```



Seriekommunikation, SCI



Bestämma Baudrate-värde

$$BR = \frac{PLLCLK}{16 \times baudrate} \qquad baudrate = \frac{PLLCLK}{16 \times BR}$$

9 600	$\frac{48\times10^6}{16\times9600} = 312{,}5$	$\frac{48\times10^6}{16\times312} \approx 9615$, $\frac{48\times10^6}{16\times313} \approx 9585$
57 600	$\frac{48\times10^6}{16\times57600} \approx 52{,}08333$	$\frac{48\times10^6}{16\times52} \approx 57692$
256 000	$\frac{48\times10^6}{16\times256000} = 11{,}71875$	$\frac{48\times10^6}{16\times12} \approx 250000$

```
Eclock:         EQU     8000000         ; 8 MHz
; BaudRate register värden, baserad på PLL-klocka
Baud9600:       EQU     (Eclock/(16*9600))
```

Initiering, ”busy-wait”

Basadress = $C8

Algoritm:
1. Initiera BAUDRATE

2. Aktivera Transmitter Receiver

Serial Communication Interface (SCI)

Offset		7	6	5	4	3	2	1	0	Mnemonic	Namn
$00	R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8	SCIBDH	Baud Rate Register High
	W										
$01	R	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	SCIBDL	Baud Rate Register Low
	W										
$02	R	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCICR1	Control Register 1
	W										
$03	R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2	Control Register 2
	W										
$04	R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1	Status Register 1
	W										
$05	R	0	0	0	0	0	BRK13	TXDIR	RAF	SCISR2	Status Register 2
	W										
$06	R	R8	T8	0	0	0	0	0	0	SCIDRH	Data Register High
	W										
$07	R	R7	R6	R5	R4	R3	R2	R1	R0	SCIDRL	Data Register Low
	W	T7	T6	T5	T4	T3	T2	T1	T0		

```
SCI0BD:         EQU     $C8     ; SCI 0 baudrate-register (16 bit).
SCI0CR2:        EQU     $CB     ; SCI 0 styr-register 2.
; Bitdefinitioner, styrregister
TE:             EQU     $08     ; Transmitter enable.
RE:             EQU     $04     ; Receiver enable.
```

Programmet...

```
; enkelt testprogram
        ORG     $1000
        JSR     serial_init
Loop:   JSR     in      ; "eka" tecken
        JSR     out
        BRA     loop
```

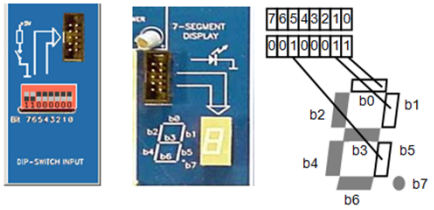
```
;       OUT tecken rutin
;       Skriv tecken till SCI0
;       Inparameter, register B: tecken.
out:    BRCLR   SCI0SR1,#TDRE,out       ; vänta till TDRF=1
        STAB    SCI0DRL                 ; skicka tecken ...
        RTS
```

```
;       IN tecken rutin
;       Läs tecken från SCI0
;       Returnera i register B
in:     BRCLR   SCI0SR1,#RDRF,in        ; vänta till RDRF=1
        LDAB    SCI0DRL                 ; läs tecken
        RTS
```

Av speciell vikt:
”maskinorienterad programmering...”

2.24 En strömbrytare och en sju-sifferindikator (se figur) är anslutna till adresser 0x400 respektive 0x600 i ett MC12 mikrodatorsystem.

Konstruera en funktion
`void DisplayNBCD( void )`
som hela tiden läser från strömbrytarna och skriver värden till sju-sifferindikatorn).



När bit 7 på inporten är ettställd skall sifferindikatorn släckas helt. När bit 7 på inporten är nollställd skall sifferindikatorn tändas enligt följande beskrivning:

- Bit 3-0 på inporten anger vad som skall visas på sifferindikatorn.
  - Om indata är i intervallet [0,9] skall motsvarande decimala siffra visas på sifferindikatorn.
  - Om indata är i intervallet [A,F] skall ett ’E’ (Error) visas på sifferindikatorn.
- Bitarna 6-4 på inporten kan anta vilka värden som helst.

Du har tillgång till en tabell i minnet med segmentkoder för de hexadecimala siffrorna [0..F] (mönster för sifferindikatorn) enligt

```
unsigned char SegCodes[]={ 0x77,0x22,0x5B,0x6B,0x2E,0x6D,0x7D,0x23,
                           0x7F,0x6F,0x3F,0x7C,0x55,0x7A,0x5D,0x18 };
```

Segmentkoden för bokstaven ’E’ ges av:

```
#define    ERROR_CODE    0x5D
```

Läsa/skriva på fasta adresser (portar)

Datatyper, storlek (8,16 eller 32 bitar...)

Heltalstyper, med eller utan tecken, vad innebär typkonverteringarna?

Bitoperationer &, |, ^ (AND, OR, XOR)

Skiftoperationer <<, >> (vänster, höger)

Kodningskonventioner

Program som kräver källtexter
både i 'C' och assemblerspråk...

Kompilatorkonvention XCC12:

- Parametrar överförs till en funktion via stacken.
- Då parametrarna placeras på stacken bearbetas parameterlistan från höger till vänster.
- Utrymme för lokala variabler allokeras på stacken. Variablerna behandlas i den ordning de påträffas i koden.
- **Prolog** kallas den kod som reserverar utrymme för lokala variabler.
- **Epilog** kallas den kod som återställer (återlämnar) utrymme för lokala variabler.
- Den del av stacken som används för parametrar *och* lokala variabler kallas *aktiveringspost*.

2.31 Inledningen (parameterlistan och lokala variabler) för en funktion ser ut på följande sätt:

```
void function( char *b, char a )
{
   char *c, *d;
.....
```

a) Visa hur utrymme för lokala variabler reserveras i funktionen (*prolog*).

b) Visa funktionens aktiveringspost, ange speciellt offseter för parametrar och lokala variabler.

```
/*
  Men även tilldelningar och
  enklare typkonverteringar!
*/
.....
{
  char a;
  int b,c;
  a = b;
  c = a;
...
}
```

2.31: a)

```
LEAS    -4,SP
```

b)

Parameter/ variabel	adressering
a	8,SP
b	6,SP
c	2,SP
d	0,SP

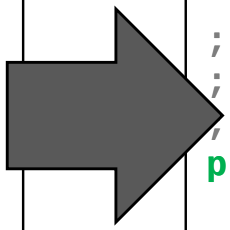
Pekare och deras användning

```c
/*
  strpbrk.c
  C-library function "strpbrk"
*/
#include  <string.h>
char *strpbrk(char *s,char *breakat)
{
  char *sscan, *bscan;

  for (sscan = s; *sscan != '\0'; sscan++) {
    for (bscan = breakat; *bscan != '\0';)
      if (*sscan == *bscan++)
        return sscan;
  }
  return((char *) 0 );
}
```

```c
/*
  memcpy.c
  C-library function "memcpy"
*/
#include  <string.h>
void *memcpy(void *dst, void *src, size_t size)
{
  char *d, char *s, size_t n;
  if (size <= 0)
    return(dst);
  s = (char *) src;
  d = (char *) dst;
  if (s <= d && s + (size - 1) >= d) {
  /* Overlap, must copy right-to-left */
    s += size - 1;
    d += size - 1;
    for (n = size; n > 0; n--)
      *d-- = *s--;
  }else
    for (n = size; n > 0; n--)
      *d++ = *s++;
  return(dst);
}
```

Assemblerprogrammering...

```
# define DATA       *( char *) 0x700
# define STATUS     *( char *) 0x701
void printerprint( char *s )
{
  while( *s )
  {
    while( STATUS & 1 )
    {}
    DATA = *s;
    s++;
  }
}
```



```
; void printerprint( char *s )
_printerprint:
; {
;   while( *s )
    LDX   2,SP
printerprint1:
    TST   ,X
    BEQ   printerprint2
;   {
;     while( !( STATUS & 1) )
;     {}
printerprint3:
    LDAB  $0701
    ANDB  #$01
    BEQ   printerprint3
;     DATA = *s;
    LDAB  1,X+      (även 's++' nedan)
    STAB  $0700
;     s++;
    BRA   printerprint1
printerprint2:
;   }
; }
    RTS
```

Maskinorienterad programmering 2013/2014

Sammanfattad...

Tisdag 3 juni

Tentamen, 14.00-18.00, Maskin

Tillåtna hjälpmedel:

Instruktionslista för CPU12

En av böckerna:

”Vägen till C” *eller*

”The C programming language”