

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.*

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

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

*Lagrings-
klass och
synlighet.*

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

```
; 2 | static int b;
1: RMB $2
```

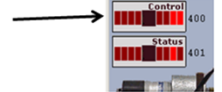
(symbolen `_1` existerar endast under assemblering och motsvarar då symbolen `'b'` i programmet. Symbolen `'b'` exporteras inte.

Subrutiner för att manipulera styrregistret OUTONE och OUTZERO

```
* Subrutin OUTONE. Läser kopian av
* borrar maskinens 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
---	---	---	---	---	---	---	---



```
*****
* SUBROUTIN - 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);
```

översätts 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);
```

Översätts 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	6, SP
lc.msb		
lb.lsb	Parameter lb	4, SP
lb.msb		
la.lsb	Parameter la	2, SP
la.msb		
PC.lsb	Återhoppadress, placeras här vid BSR	0, SP
PC.msb		

`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
```

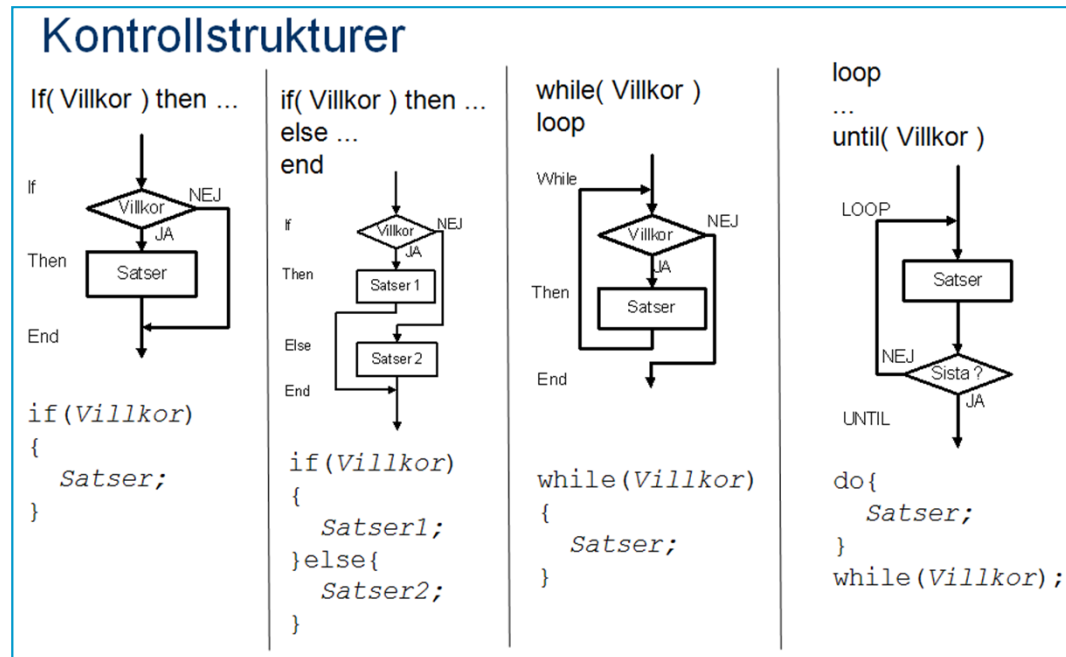
Returvärden via register

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

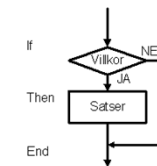
Storlek	Benämning	C-typ	Register
8 bitar	byte	char	B
16 bitar	word	short int	D
32 bitar	long	long int	Y/D

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

- *beskriva och använda olika kontrollstrukturer.*



If (...) {...}



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

Bättre kodning...

```
DipSwitch EQU $600
HexDisp EQU $400

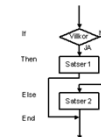
...
TST DipSwitch
BEQ end

LDAB DipSwitch
STAB HexDisp

end:
```

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

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



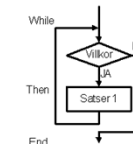
```
if (DipSwitch == 0)
    HexDisp = 1;
else
    HexDisp = 0;
```

```
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 (...) {...}



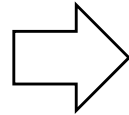
```
Delay( unsigned int count )
{
    while (count > 0)
        count = count - 1;
}
```

```
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

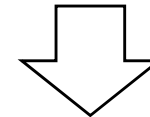
- *beskriva och använda sammansatta datatyper (fält och poster) och enkla datatyper (naturliga tal, heltal och flyttal).*

```
/*  
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
```

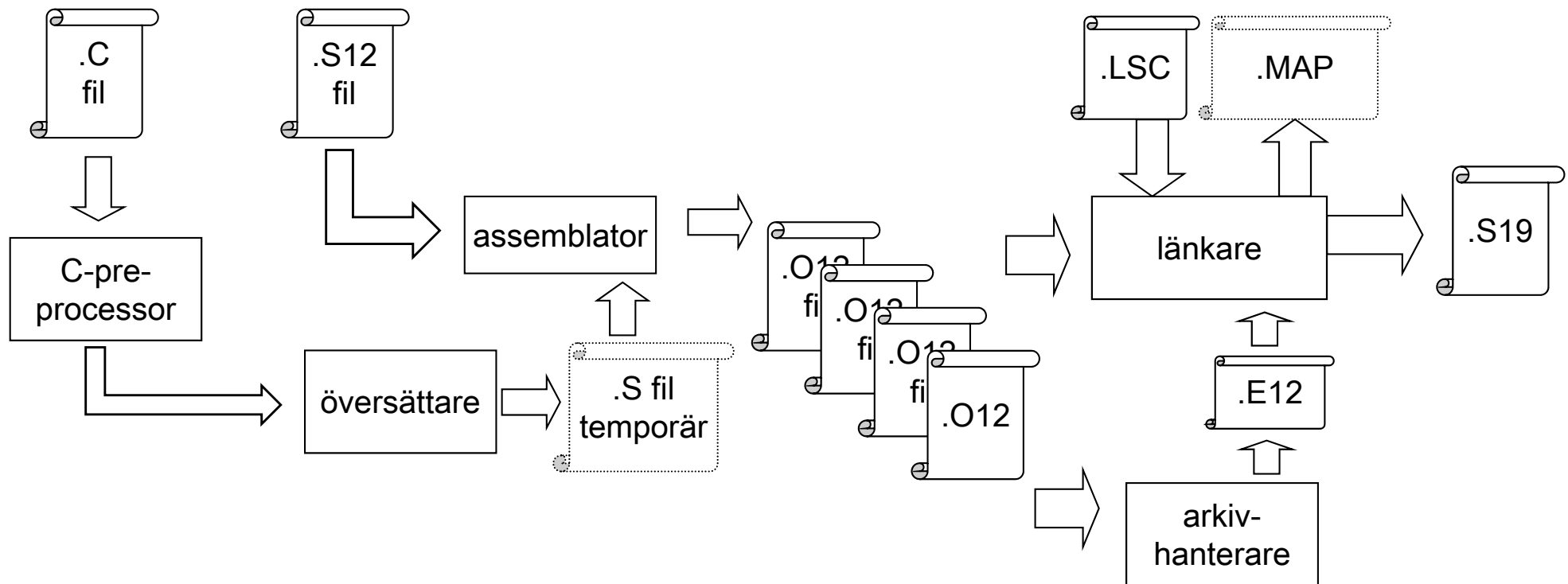
Kunna redogöra för olika lagringsklasser (GLOBAL, STATIC, LOCAL) och "synlighet".

2. Programutvecklingsteknik

Att kunna:

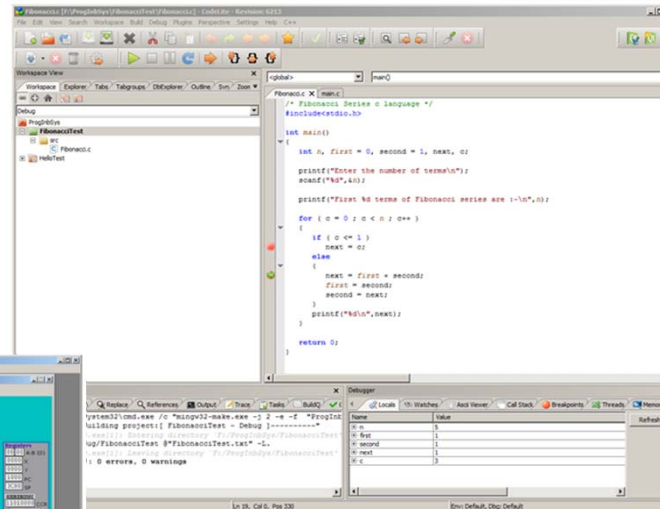
- beskriva översättningsprocessen, dvs. assemblatorns arbetssätt, preprocessorns användning, separatkompilering och länkning.
- konstruera, redigera och översätta (kompilera och assemblera) program
- testa, felsöka och rätta programkod med hjälp av avsedda verktyg.

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



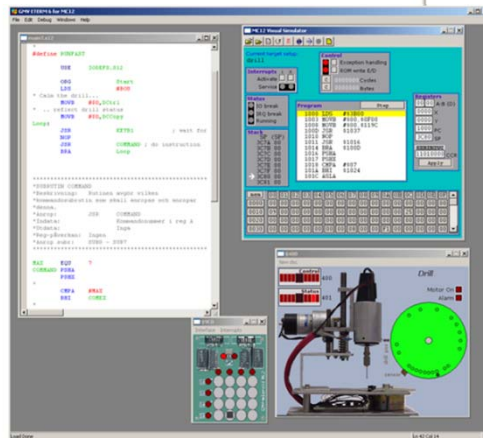
- *konstruera, redigera och översätta (kompilera och assemblera) program*
- *testa, felsöka och rätta programkod med hjälp av avsedda verktyg.*

CodeLite för
moment 3
"Länkad lista".



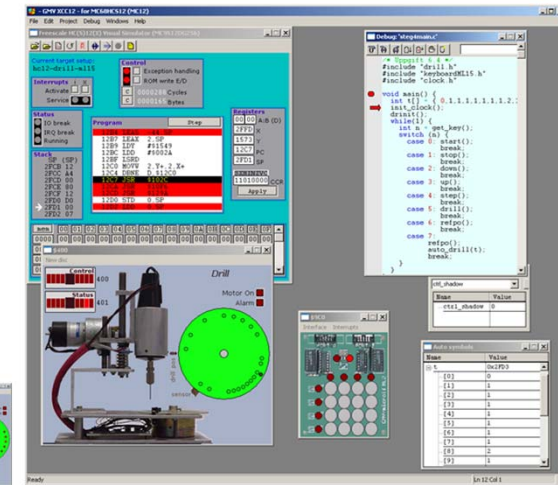
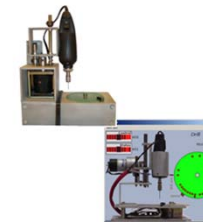
Moment 1 och 2:

ETERM för
Simulator och
laborationssystem



Moment 4:

XCC12
för Simulator
och laborations-
system



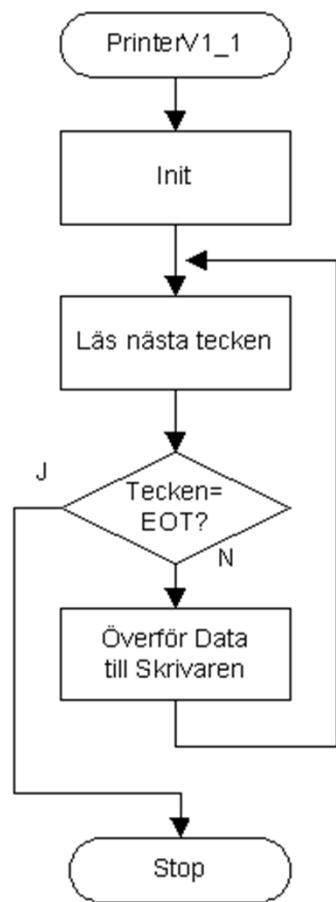
Dessa lärandemål har vi kontrollerat under laborationer.

3. Systemprogrammerarens bild

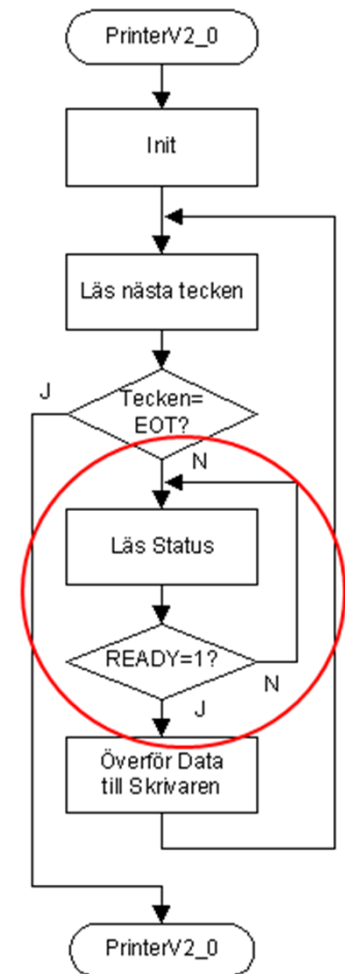
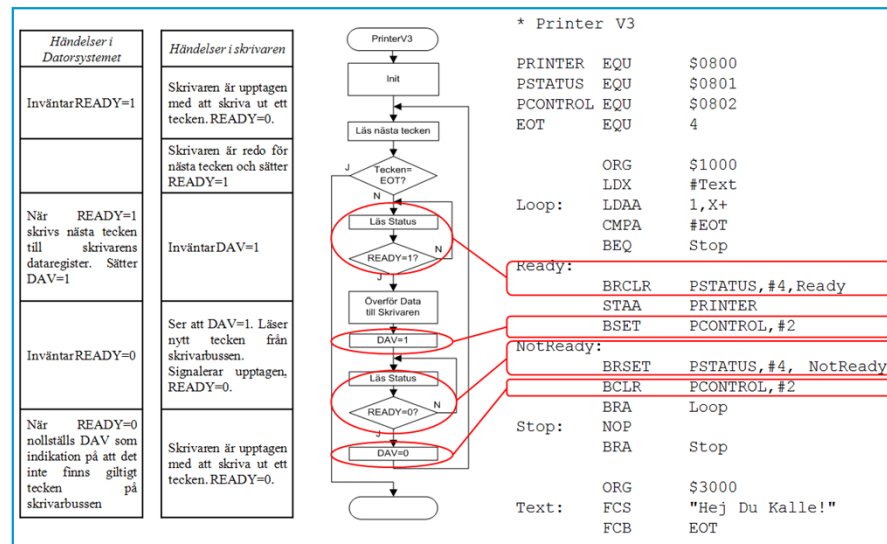
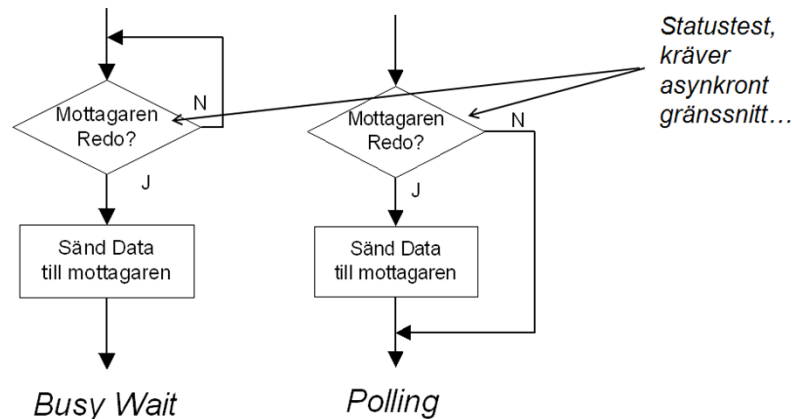
Att kunna:

- 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 rundfråganing.



Villkorlig överföring



- *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.

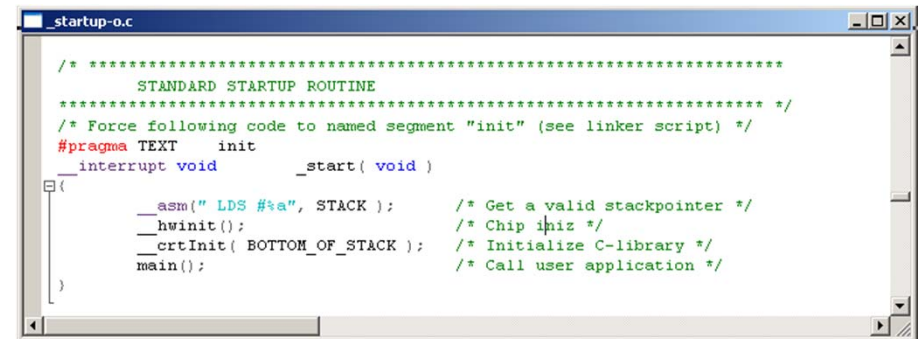
```
ORG      $FFF2
FDB      irq_service_routine
FDB      xirq_service_routine
FDB      software_interrupt_service_routine
FDB      illegal_opcode_service_routine
FDB      cop_service_routine
FDB      clock_monitor_fail_service_routine
FDB      Application_Start
```

; Symbolen "Application_Start_Address" kan vara godtycklig.

```
ORG      Application_Start_Address
Application_Start:
LDS      #TopOfStack
...
...
ANDCC    #$FE      ; nollställ I-flagga
JSR      _main
```

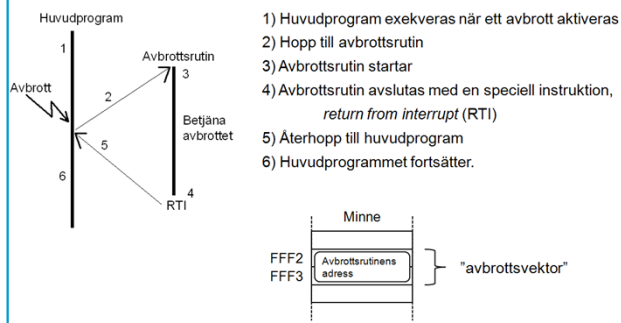
Vår slutliga "appstart" blir nu:

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

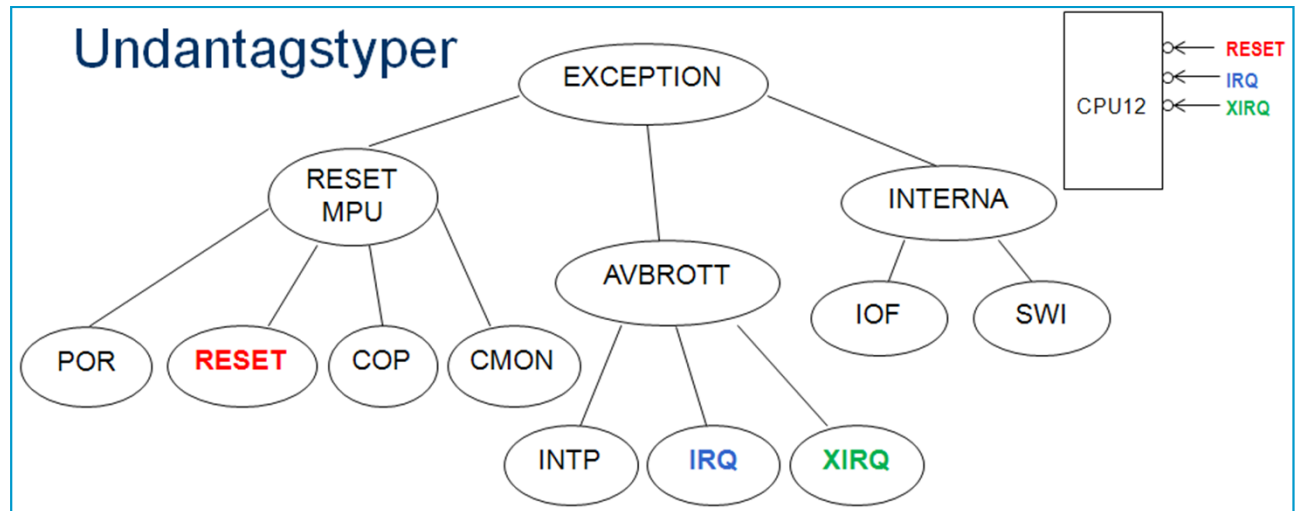


- beskriva och exemplifiera olika undantagstyper: interna undantag, avbrott och återstart.

Avbrott

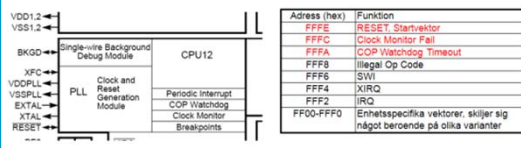


Undantagstyper

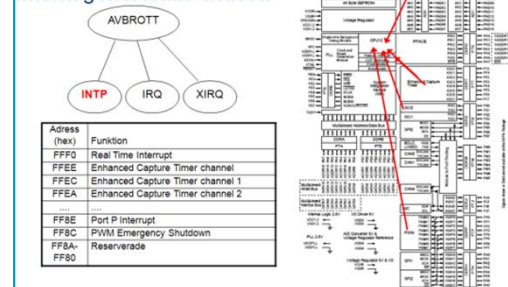


Fatale fel, kräver RESET av CPU

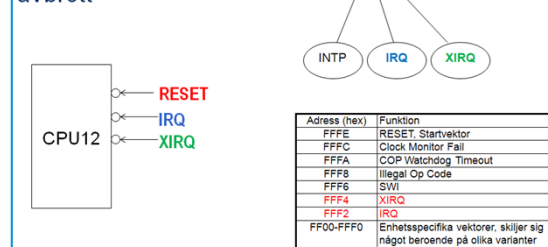
POR, Power On Reset, vid spänningstillslag
RESET, insignal till processorn aktiveras.
COP, Computer Operating Properly, så kallad watchdog-funktion.
CMON, Clock Monitor Reset, övervakar E-klockan, om frekvensen sjunker under 10 kHz genereras RESET.



Internt genererade avbrott



Externt genererade avbrott



Interna undantag

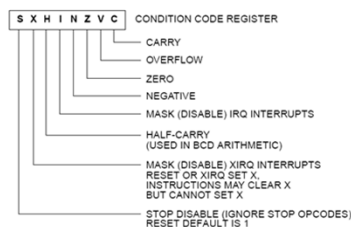
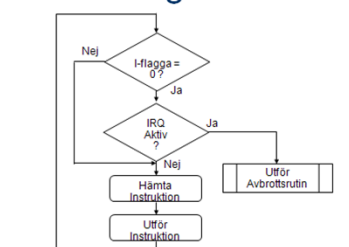
Om processorn avkodar en otillåten operationskod kallas detta *Illegal Opcode Fetch* (IOF).
Processorn avbryter då, *sparar registerinnehåll på stacken*, läser autovektorn för IOF och utför undantagshantering.

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

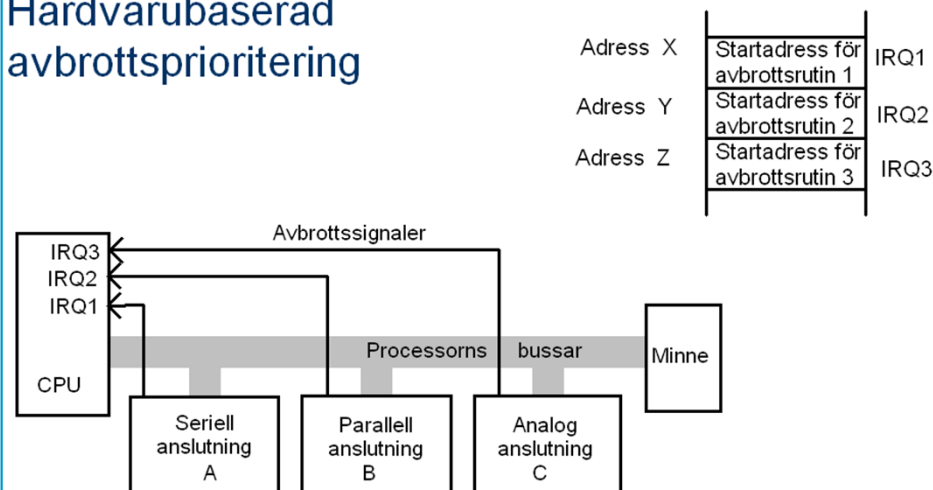
ANDCC #11101111

Demaskera X-avbrott:

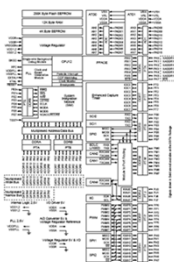
ANDCC #10111111

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.

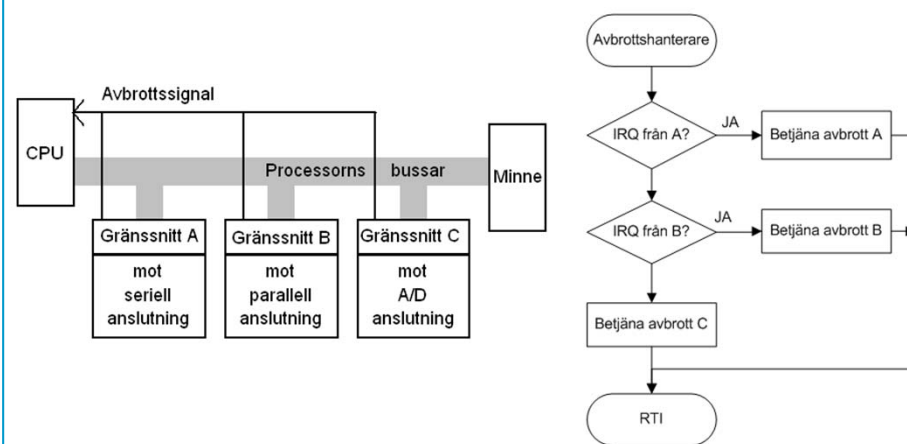
Det finns vissa möjligheter att ändra detta programmässigt.

Högre prioritet

Lägre prioritet

Adress (hex)	Funktion
FF00	Real Time Interrupt
FF01	Enhanced Capture Timer channel
FF02	Enhanced Capture Timer channel 1
FF03	Enhanced Capture Timer channel 2
FF04	Enhanced Capture Timer channel 3
FF05	Enhanced Capture Timer channel 4
FF06	Enhanced Capture Timer channel 5
FF07	Enhanced Capture Timer channel 6
FF08	Enhanced Capture Timer channel 7
FF09	Enhanced Capture Timer overflow
FF0A	Pulse accumulator A overflow
FF0B	Pulse accumulator input edge
FF0C	SPI0
FF0D	SCIO
FF0E	SC11
FF0F	AT00
FF10	AT01
FF11	Port A
FF12	Port B
FF13	Port C
FF14	Modulus Down Counter underflow
FF15	Pulse Accumulator B overflow
FF16	PLL lock
FF17	CRG Self Clock Mode
FF18	Arvänder (BOLC)
FF19	IIC Bus
FF1A	SPI1
FF1B	Reserved
FF1C	Reserved
FF1D	EEPROM 1-bit
FF1E	PLASH 1-bit
FF1F	CAN0 wake-up
FF20	CAN0 errors
FF21	CAN0 receive
FF22	CAN0 transmit
FF23	CAN0 wake-up
FF24	CAN0 errors
FF25	CAN0 receive
FF26	CAN0 transmit
FF27	Port P Interrupt
FF28	PWM Emergency Shutdown
FF29	Reserved
FF2A	Reserved
FF2B	Reserved
FF2C	Reserved
FF2D	Reserved
FF2E	Reserved
FF2F	Reserved

Programbaserad avbrottsprioritering



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

Processtillstånd

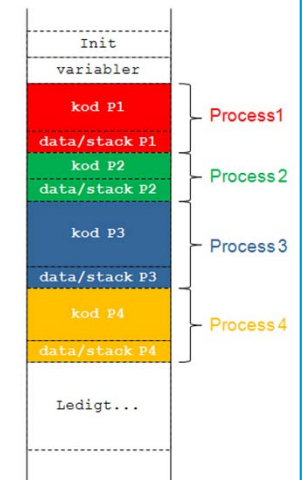
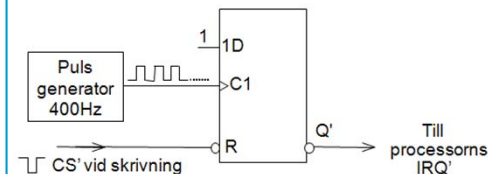


Processbyte

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

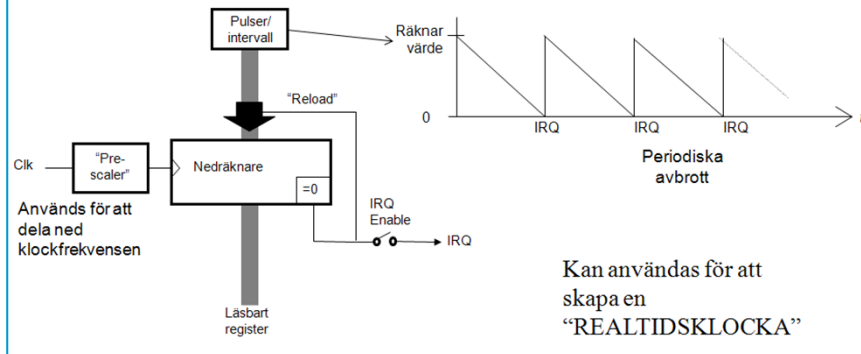
HDW krav: En avbrottskälla som ger regelbundna avbrott (Ex Timer)

SW krav: En avbrottsrutin (SWITCH) som växlar process



- *beskriva och använda kretsar för tidmätning.*

Räknarkrets ("timer"), principiell funktion



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.

.. 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 avbrottsintervall enligt;

```

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

```

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	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	DDRA
	W	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	
\$03	R	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	1=OUT	DDRB
	W	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	0=IN	
\$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:

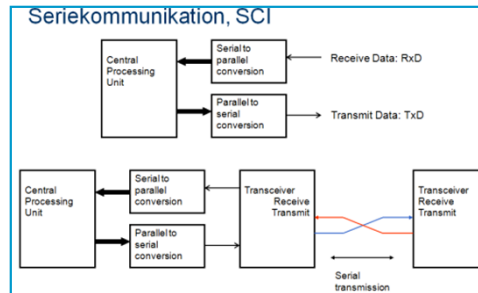
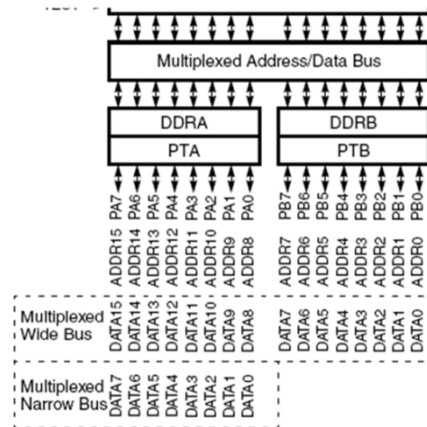
```
PORTA EQU 0
PORTB EQU 1
DDRA EQU 2
DDRB EQU 3
```

```
... CLR DDRA
... MOV $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;
```

**Bestämma Baudrate-värde**

$$BR = \frac{PLLCLK}{16 \times \text{baudrate}} \quad \text{baudrate} = \frac{PLLCLK}{16 \times BR}$$

9 600	$\frac{48 \times 10^6}{16 \times 9600} = 312,5$	$\frac{48 \times 10^6}{16 \times 312} = 9615$	$\frac{48 \times 10^6}{16 \times 313} = 9585$
57 600	$\frac{48 \times 10^6}{16 \times 57600} = 52,08333$	$\frac{48 \times 10^6}{16 \times 52} = 57692$	
256 000	$\frac{48 \times 10^6}{16 \times 256000} = 11,71875$	$\frac{48 \times 10^6}{16 \times 12} = 250000$	

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

Initiering, "busy-wait"

Basadress = \$C8

Algorithm:
1. Initiera
BAUDRATE

2. Aktivera
Transmitter
Receiver

Serial Communication Interface (SCI)									
Offset	7	6	5	4	3	2	1	0	Mnemonic
\$00	R	0	0	0	0	0	0	0	SCI0BD
\$01	R	0	0	0	0	0	0	0	SCI0CR2
\$02	R	0	0	0	0	0	0	0	SCI0CR1
\$03	R	0	0	0	0	0	0	0	SCI0SR1
\$04	R	0	0	0	0	0	0	0	SCI0SR2
\$05	R	0	0	0	0	0	0	0	SCI0DR
\$06	R	0	0	0	0	0	0	0	SCI0DRH
\$07	R	0	0	0	0	0	0	0	SCI0DRL

```
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 TDRE=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
```

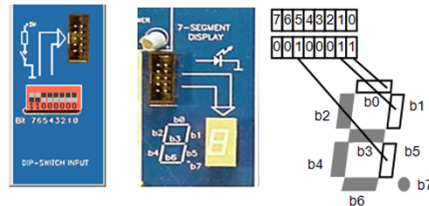
Av speciell vikt: "maskinorienterad programmering..."

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;  
    .....  
}
```

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

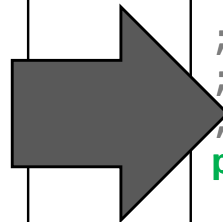
```
/*
    strpbrk.c
    C-library function "strpbrk"
*/
#include <string.h>
char *strpbrk(char *s, char *breakat)
{
    char *sscan, *bscan;

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

```
/*
    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...

Måndag 10/3

Tentamen, 14.00-18.00

Tillåtna hjälpmedel:

Instruktionslista för CPU12

En av böckerna:

Vägen till C

The C programming language