

Maskinorienterad Programmering DAT015 2012/2013

Sammanfattning

"Syftet med kursen är att vara en introduktion till konstruktion och programmering av små inbyggda system."

Ur innehållet:

- Vi repeterar kursens "lärandemål"
- Övriga frågor...

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

Funktioners parametrar och returvärdet.

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

```

int a;
static int b;

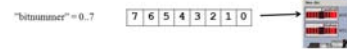
; 1 | int a;
SEGMENT bss
_a: RMB $2
EXPORT _a [r,2]

; 2 | static int b;
; 1: RMB $2
(symbolen 1 existerar endast under assemblering och motsvarar då symbolen 'b' i programmet. Symbolen 'b' exporteras inte.
    
```

Lagringsklass och synlighet.

Subrutiner för att manipulera styrregistret OUTONE och OUTZERO

- Subrutin OUTONE. Läser kopien av hörmaskinens styrgod på adress
- DCOpy. Inställer en av bitarna och skriver det nya styrgodet till utporten DCTM samt tillbaka till kopien DCOpy.
- Risen som mottar alla gas av innehåll i D-registret (0-7) vid anrop.
- OS (0) x 7 utföra instegning.
- Anrop: LDAB #bitnummer ; OS ; OUTONE
- Utdata: Inga
- Registerpåverkan: Ingen
- Anropad subrutin: Inga



- 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 (1a, 1b, 1c) :

```

översättas till:
LDD 1a
LDX 1b
LDY 1c
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 "In Line"

```

; "in line" parameteröverföring, värdet 10 ska
; överläsas till en subrutin:
SUB dummyfunc
FCB 10
...

dummyfunc:
LDAB [0,SP] ; parameter->B
LDX 0,SP ; återhoppadress->X
INX ; modifiera ++
STX 0,SP ; .. tillbaka till stack
...
RTS
    
```

Parameteröverföring via stacken

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

```

dummyfunc (1a, 1b, 1c):
översättas till:
LDD 1c
PSHD
; (alternativt STD 2,-SP)
LDD 1b
PSHD
LDD 1a
PSHD
BSR dummyfunc
LEAS 6,SP
    
```

Innehåll	Kommentar	Adressering via SP i subrutinen
1c.msb	Parameter 1c	6, SP
1b.msb	Parameter 1b	4, SP
1a.msb	Parameter 1a	2, SP
1a.msb	Återhoppadress	2, SP
PC.msb	Återhoppadress	0, SP

```

dummyfunc:
; ..
LDD 2,SP ; parameter 1a till register D
; ..
LDD 4,SP ; parameter 1b till register D
; ..
LDD 6,SP ; parameter 1c till register D
    
```

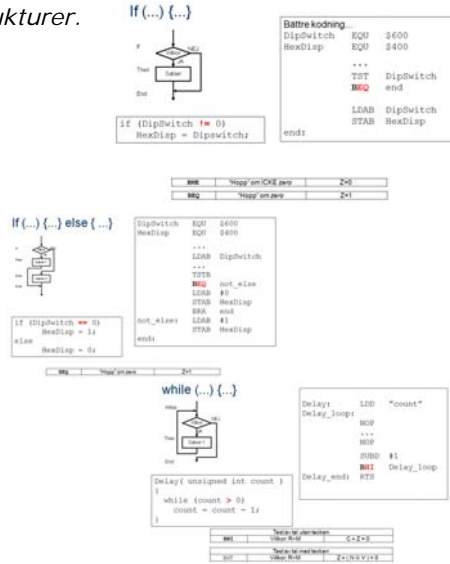
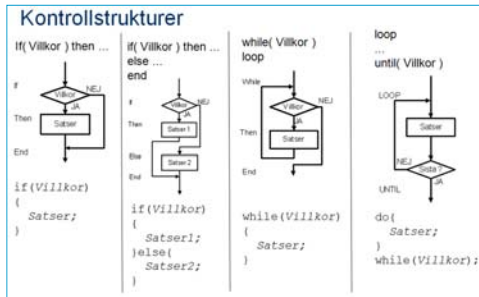
Returvärden via register

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

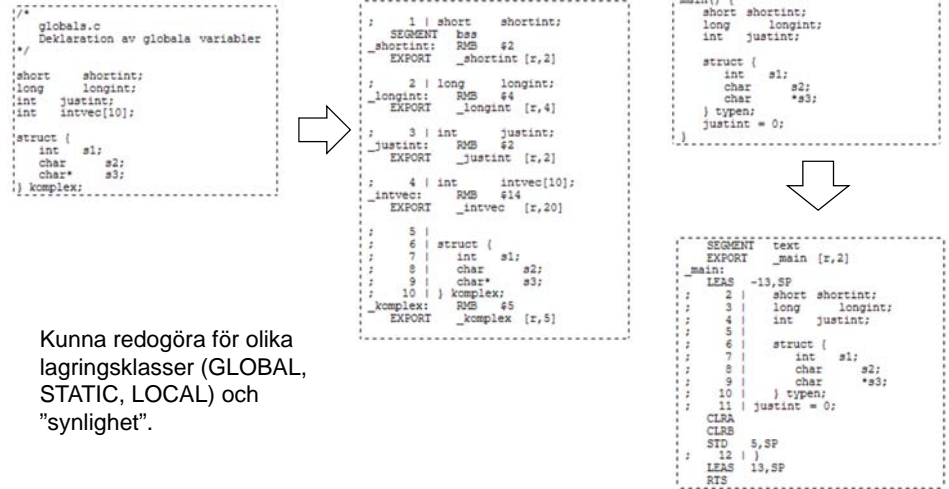
Storlek	Benämning	C-typ	Register
8 bitar	byte	char	B
16 bitar	word	short int	D
32 bitar	long	long int	Y/D

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

- beskriva och använda olika kontrollstrukturer.



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



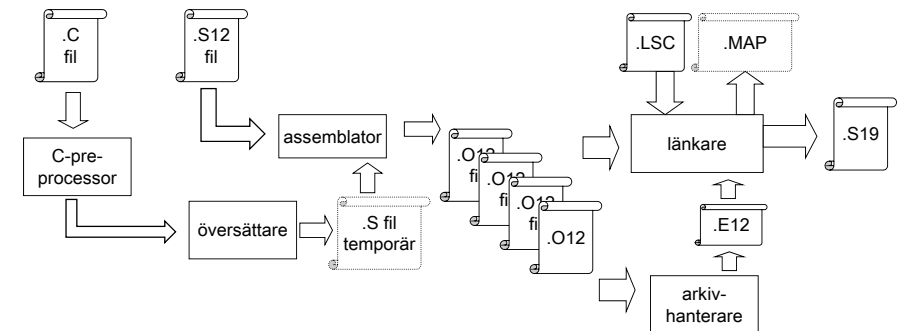
Kunna redogöra för olika lagringsklasser (GLOBAL, STATIC, LOCAL) och "synlighet".

2. Programutvecklingsteknik

Att självständigt 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.

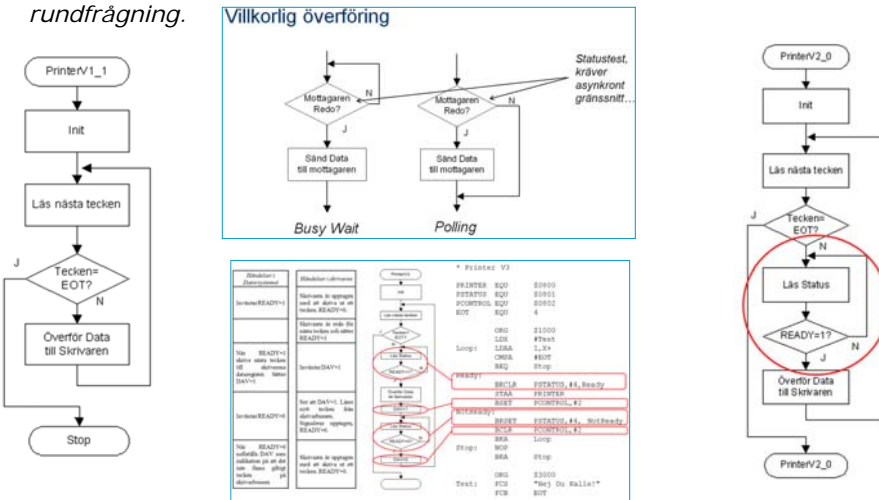
Dessa lärandemål har vi kontrollerat under laborationer.

3. Systemprogrammerarens bild av inbäddade system

Att självständigt 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.
- 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 programkoden illustrerar hur några avbrottsrutiner respektive avbrottsvektorer kan definieras i en fristående HCS12-applikation.

```

ORG    $FFFF
FDB    irq_service_routine
FDB    irq_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.
Application_Start:
LDS    #TopOfStack
...
ANDCC  #SFE    ; nollställ I-flagga
USR    _main

```

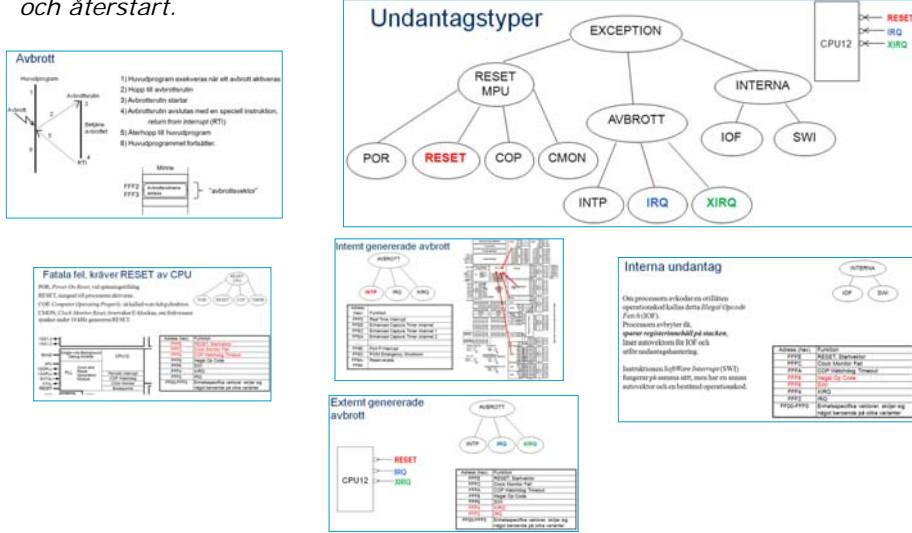
Vår slutliga "appstart" blir nu:

```

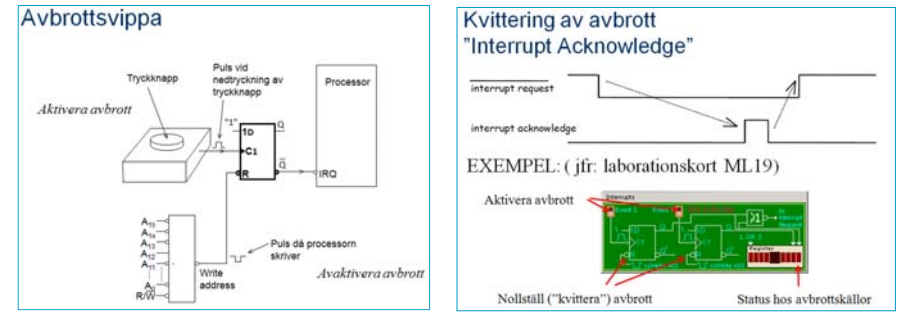
segment    init
export    _exit
import    _main
function  _start, _start_end
* Här börjar exekveringen...
_start
LDS    #S2FFF
JSR    _main
_exit:  NOP
BR    _exit
_start_end

```

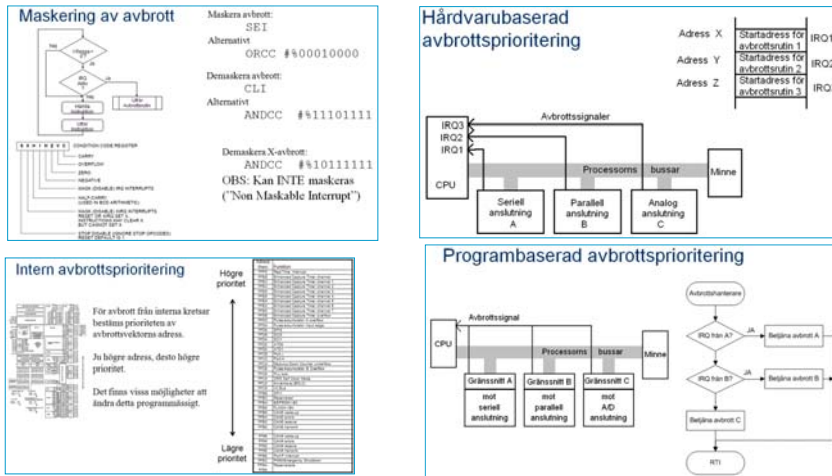

- beskriva och exemplifiera olika undantagstyper: interna undantag, avbrott och återstart.



- konstruera enklare avbrottssystem med användning av digitala komponenter.



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



Av speciell vikt: "maskinorienterad programmering..."

2.24 En strömbrytare och en sju-sifferindikator (se figur) är anslutna till adresser 0x600 respektive 0x601 i ett MC12 mikrodatorsystem.

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

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

- Bit 3-0 på porten anger vad som skall visas på sifferindikatorn.
- Om indatä är i intervallet [0,5] skall motsvarande decimala siffror visas på sifferindikatorn.
- Om indatä är i intervallet [A-F] skall ett 'E' (Error) visas på sifferindikatorn.
- Bitarna 6-4 på porten 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[] = { 0x7F, 0x2D, 0x3B, 0x49, 0x56, 0x6D, 0x7D, 0x23,
                             0x7E, 0x8F, 0x9F, 0x8C, 0x93, 0x9A, 0x8D, 0x8E, 0x91 };

```

Segmentkoden för bokstaven 'E' ges av:

```

#define ERROR_CODE 0x10

```

- 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 assemblyspråk...

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

```
void funktion( 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.

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.
- *Epi-log* 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: a) LEAS -4,SP

b)

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

Pekare och dess användning...

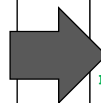
```
/*
  strpbrk.c
  C-library function "strpbrk"
*/
#include <string.h>
char *strpbrk(char *s, char *breakat)
{
    char *sscan, *bscan;

    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:
; }
; }
```

**Maskinorienterad Programmering
2012/2013**

Sammanfattad...

Torsdag 13/12, 13.15-17.00

"Öppet hus" i lab 4220

Onsdagen 19/12

Tentamen, 14.00-18.00