

Maskinorienterad programmering

Sammanfattning

Introduktion till Maskinorienterad programmering

Målsättningar:

- ✓ Bli bekant med utvecklingsmiljön
- ✓ Kunna redigera, kompilera och testa ett enkelt C-program

Ur innehållet:

- ✓ Programutveckling i C och assembler
- ✓ Programutvecklingsmiljö
- ✓ Så översätts ett C-program

Introduktion till C

- ✓ Variabeldeklarationer
- ✓ Heltalstyper
- ✓ Funktioner
- ✓ Textsträngar

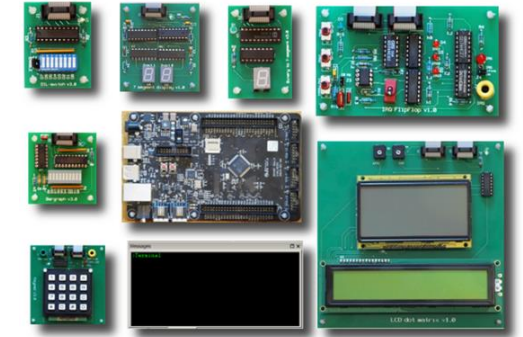


Ur innehållet:

- Vi rekapitulerar kursens syften
- Vi repeterar kursens "lärandemål"
- Vi belyser hur den skriftliga delen av examinationen genomförs.

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.



"ISA" – Instruction Set Architecture

- ✦ Hur lagras operanderna förutom i minnet?
Korttidslagring dvs. *register*
- ✦ Hur nås operand i minnet?

Primära instruktionsgrupper

Load/Store för att läsa från, och skriva till, minnet
Load Till,Från
Store Från,Till

Utrycksevaluering
Operation Till,Från1,Från2

Exempel: C

```
char c,d;  
void foo( void )  
{  
    c = ~d;  
}
```

```
char c,d;  
void foo( void )  
{  
    c = ~d;  
}
```

ARM/Thumb assembler

```
C: .SPACE 1  
d: .SPACE 1
```

```
foo:  
    LDR    R0,=d  
    LDRB  R0,[R0]  
    MVN   R1,R0  
    LDR   R0,=c  
    STRB  R1,[R0]  
    BX    LR
```

Programflödeskontroll

B Symbol
Bxx Symbol
BX Register

Diverse
MOV Till,Från
MOV Till,#konstant

mellan register
konstant till register

Bitvis operationer

Exempel:
Packa upp data från en variabel

Packa (och packa upp) (DAY/MONTH/YEAR)
i variabler med typen int.

empty year month day

Visa en funktion: int unpack_year(int date) i C och Thumb1 assembler som
packar upp heltalet som representerar år, och returnerar detta.

```
int unpack_year( int date ) {  
    int rval;  
    rval = date >> 9;  
    rval &= 0x1fff;  
    return rval;  
}
```

Konvention

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

RC

Flerdimensionella fält "matriser" – N × M element, indexeras [0..N-1] [0..M-1]

Exempel:

```
char mc [N][M];  
int i1,i2;
```

Referens: mc[i1][i2]

Minnesadress: =mc + ((i1*M)+i2)*sizeof(char)

Kompilator ersätter multiplikationer med skift/add
kombinationer eftersom mul-instruktionen tar
betydligt fler klockcykler att utföra.

M = 2³ + y

Exempelvis då M=10:

10 = 2³ + 2

11*10 =

11*(2³+2) =

11<<3 + 11 + 11

basadress

mc[0][0]

mc[0][1]

mc[0][2]

mc[0][3]

mc[0][4]

mc[0][5]

mc[0][6]

mc[0][7]

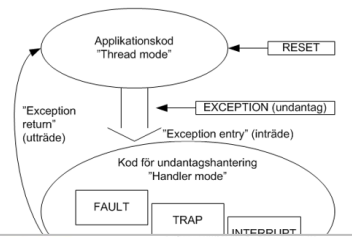
mc[0][8]

mc[0][9]

Undantagshantering - "exception"

Med "undantag" menar vi olika typer
av händelser:

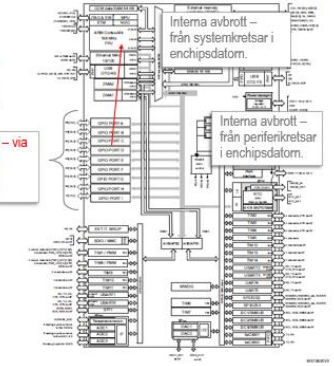
- RESET (återstart):
 - power on (kallstart)
 - warm reset (varmstart)
- FAULT:
 - Exekveringsfel – kan inte fortsätta
- TRAP:
 - Programmerat avbrott, initierat av maskininstruktion



Externa avbrott från periferikretsar



Externa avbrott – via
portpinnar från
kretsar utanför
enchipsdataom.



1. Programutveckling i C och assemblyspråk

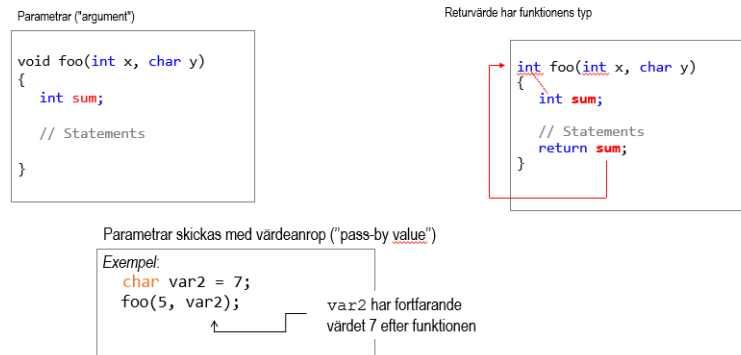
Kunna utföra programmering i C och assemblyspråk samt kunna:

- 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.*

Funktioners parametrar och returvärden.

C-funktioner, parametrar och returvärden



Funktionsparametrar, 8 eller 16 bitar

All aritmetik utförs med 32 bitar, alla parametrar måste vara 32 bitar. short och signed char ska därför typkonverteras före anrop.

Exempel: Funktionens returvärde

Konventionerna säger att vi använder register **R0** för 8,16 och 32-bitars returvärde...
...och registerparet **R0:R1** för 64 bitars returvärde

Exempel:
Funktionen:
`signed int rintval( void )`
och de glo
`signed`
är definiera
Visa hur fun
`z = f(`
kodas i ass

Exempel:
Funktionen:
`int rintval( void )`
{
 return 0x12345678;
}
kodas:
rintval:
 LDR R0,=0x12345678
 BX LR

Exempel:
Funktionen:
`long long rval( void )`
{
 return 0x1234567800000000;
}
kodas:
rval:
 MOV R0, #0
 LDR R1,=0x12345678
 BX LR

Lagringsklasser och synlighet.

Variabler och minnesanvändning

Ett deklarerat objekt karakteriseras av:

- Synlighet
- Varaktighet
- Typ av variabel

Synlighet

Synligheten kan begränsas av deklarationens block.

Varaktighet ("storage duration")

Varje deklarerat objekt har en *varaktighet* under vilken det kan användas. Det finns fyra olika typer av varaktighet i C:

- automatic: reserverat minnesutrymme finns i det block objektet deklarerats. ("tillfällig lagring")
- static: reserverat minnesutrymme under hela programexekveringen, initieras innan exekvering av main-funktionen. ("permanent lagring")
- allocated: minnesutrymme som reserverats med funktioner för dynamisk minneshantering (malloc/free).
- thread: reserverat minnesutrymme under hela exekveringen av den tråd där objektet deklarerats. Jämför med "static" men inte hela programmet.

Permanent lagring

Variabler med permanent adress i minnet:

I ARM/Thumb assemblerspråk:

Tillfällig lagring – i register eller på stacken

Exempel:
Gör en lämplig registerallokering och koda tilldelningssatserna i funktionen:

```
void f (int a, int b, int c)
{
    int d;
    int e;
    d = a;
    e = b;
    ...
}
```

Lösning:

Vi ser att R0,R1 och R2 redan är upptagna av parametrar, R3 upplåter vi för uttrycksutvärdering i funktionen och gör registerallokeringen:

d=R4, e=R5.

Vi får då följande kodning:

```
@void f (int a, int b, int c)
f:
@{
@   int d;
@   int e;
@   PUSH    {R4,R5,LR}
@   MOV     R4,R0
@   MOV     R5,R1
@   ...
@   POP     {R4,R5,PC}
@}
```

R7 Specialt används av GCC R7 som pekare till aktiveringspost (stack frame)
R6 Om dessa register är används för variabler och temporära
R5 Om dessa används måste de sparas och återställas av
R4 Om dessa används måste de sparas och återställas av
R3 Om dessa används måste de sparas och återställas av
R2 Om dessa används måste de sparas och återställas av
R1 Om dessa används måste de sparas och återställas av
R0 Om dessa används måste de sparas och återställas av

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

Registeranvändning

I princip kan man använda de generella registren som man vill men det är mycket bättre att följa ARM's rekommendationer:

Register	Användning
R15 (PC)	Programräknare
R14 (LR)	Länkregister
R13 (SP)	Stackpekare
R12 (IP)	
R11	Dessa register är avsedda för variabler och som temporära register. Om dom används måste dom sparas och återställas av den anropade (callee) funktionen
R10	
R9	
R8	
R7	Speciellt använder GCC R7 som pekare till aktiveringspost (stack frame) Också dessa register är avsedda för variabler och temporärbruk
R6	
R5	Om dom används måste dom sparas och återställas av den anropade (callee) funktionen
R4	
R3	parameter 4 / temporärregister
R2	parameter 3 / temporärregister

Funktionsparametrar

Konventionerna säger att vi använder register R0,R1,R2,R3, (i denna ordning) för parametrar som skickas till en subrutin. Övriga parametrar läggs på stacken (behandlas senare i kursen).

Exempel:

Följande deklarationer är gjorda:
`int a,b,c,d;`

Visa hur följande funktionsanrop kodas i assemblerspråk:

```
sub(a,b,c,d);
```

Assembler:

```
LDR R0,a  
LDR R1,b  
LDR R2,c  
LDR R3,d  
BL sub
```

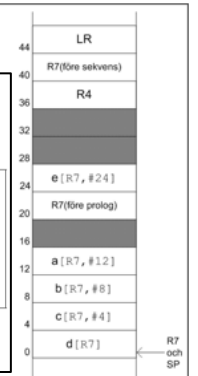
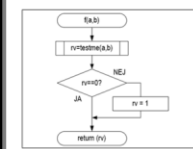
När inte registren räcker till

Då registren inte räcker till måste parametrar och lokala variabler lagras i minnet, stacken är då lämpligaste stället.

Detta är också sant då C-kompilatorn genererar kod som ska användas med en debug

Stackens utseende i sub "aktiveringspost"

```
f:  
PUSH {LR}  
BL testme  
CMP R0,#0  
BEQ L1  
MOV R0,#1  
L1:  
POP {PC}
```



Returvärden via register

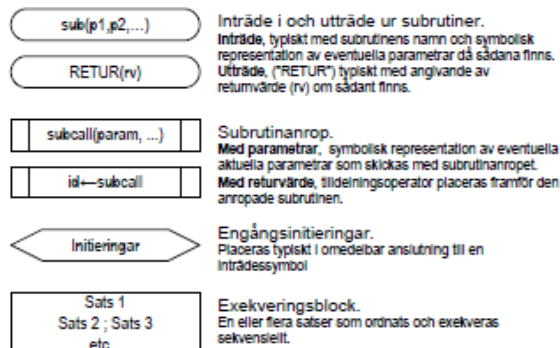
Storlek	C-typ	Register
8-32 bitar	char/short/int	R0
64 bitar	long long	R1:R0

Returvärden via stack

Krävs typiskt då en subrutin ska returnera en komplett post, eller ett helt fält. Den anropande funktionen ska då först reservera utrymme för returvärdet. En pekare till detta utrymme läggs sedan som första parameter vid anropet:

- *beskriva och använda olika kontrollstrukturer.*

Flödesdiagram för programstrukturer



Subrutiner ("funktioner")

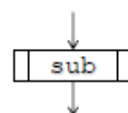
C-operator	Betydelse	Operation	Instruktion	RTN
$\mathbb{E}()$	funktionsanrop	Branch and link	BL <label>	LR=PC, PC=label
"return"		Branch and exchange	BX Rx	PC=Rx

EXEMPEL:

```

BL sub
@ returadress -> LR
...
sub:
...

```



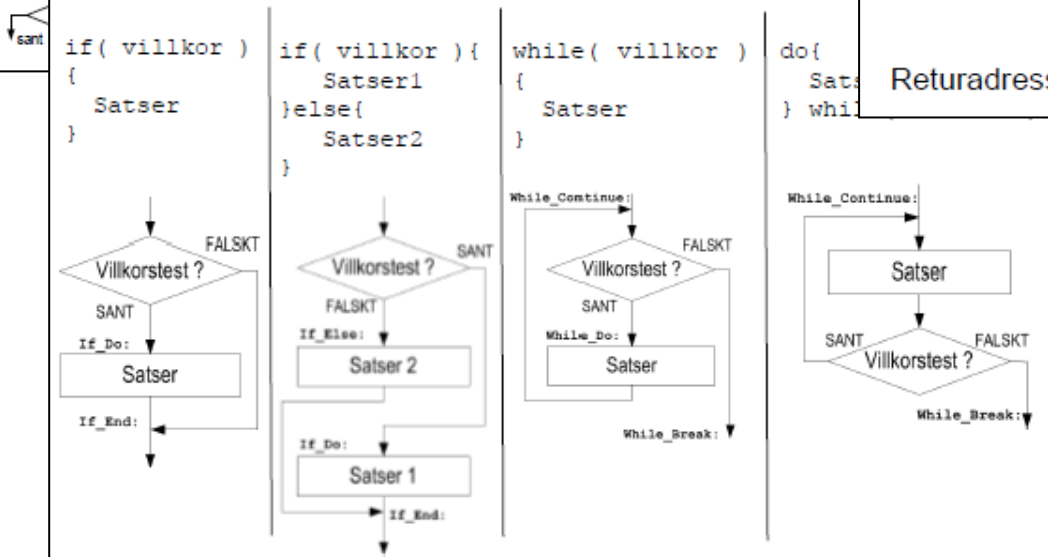
```

C:
...
sub ();
...

```

Returadressen sparas i regi

Kontrollstrukturer



Instruktioner för villkorlig programflödeskontroll

C-operator	Betydelse	Datatyp	Instruktion
==	Lika med	signed/unsigned	BEQ
!=	Skild från	signed/unsigned	BNE
<	Mindre än	signed unsigned	BLT BCC
<=	Mindre än eller lika	signed unsigned	BLE BLS
>	Större än	signed unsigned	BGT BHI
>=	Större än eller lika	signed unsigned	BGE BCS

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

Ordlängder och heltalstyper, ANSI C

Typ	Förklaring
char	Minsta adresserbara enhet som kan rymma en basal teckenuppsättning. Det är dock en heltalstyp som kan vara signed eller unsigned, detta är implementationsberoende.
signed char	Tolkas som heltal med tecken. Måste minst kunna representera talområdet [-127, +127], dvs. minst 8 bitar.
unsigned char	Tolkas som heltal utan tecken. Måste minst kunna representera talområdet [0, 255], dvs. minst 8 bitar.
short	Kort heltalstyp med tecken. Måste minst kunna representera talområdet [-32767, +32767], dvs. minst 16 bitar.
short int	Observera att talområdet [-32768, +32767] som råas vid 2-komplementrepresentation, också är tillåtet.
signed short	
signed short int	
unsigned short	Kort heltalstyp utan tecken. Måste minst kunna representera talområdet [0, +65535], dvs. minst 16 bitar.
unsigned short int	
long	Lång heltalstyp med tecken. Måste minst kunna representera talområdet [-2147483647, +2147483647], dvs. minst 32 bitar.
long int	
signed long	
signed long int	
unsigned long	
unsigned long int	

Typer för pekare, ARM-M4

```
char *cptr;      /* pekar på 8-bitars datatyp */
short *sptr;     /* pekar på 16-bitars datatyp */
int *iptr;       /* pekar på 32-bitars datatyp */
```

Adressutrymmet är 32 bitar

PC, SP och ALLA pekartyper är 32 bitar

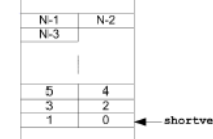
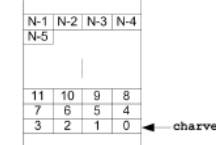
EXEMPEL:

```
cptr = ( char *) 0x40021010
sptr = ( short *) 0x40021010
iptr = ( int *) 0x40021010
```

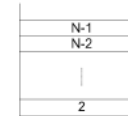
Vad är skillnaden?

Fält – "vektorer" – N element, indexerats 0..N-1

```
char charvec[N];
sizeof(charvec) =
N bytes
```



```
short shortvec[N];
sizeof(shortvec) =
N*2 bytes
```



```
int intvec[N];
sizeof(intvec) =
N*4 bytes
```

EXEMPEL: Vi har deklarationerna:

```
int i, j, *ip, ivec[20];
```

Koda följande tilldelningar i assemblyspråk

- a) `i = ivec[j];`
- b) `ip = &ivec[j];`

Structs (sammansatta datatyper)

I uppgift 39:

```
typedef struct tPoint{
    unsigned char x;
    unsigned char y;
} POINT;

#define MAX_POINTS 20

typedef struct tGeometry{
    int numpoints;
    int sizex;
    int sizey;
    POINT px[ MAX_POINTS ];
} GEOMETRY, *PGEOMETRY;
```

Skapa och initiera en variabel av typen GEOMETRY:

```
GEOMETRY ball_geometry = {
    12, 4, 4,
    { // POINT px[20]
        {0,1}, // px[...]
        {0,2},
        {1,0},
        {1,1},
        {1,2},
        {1,3},
        {2,0},
        {2,1},
        {2,2},
        {2,3},
        {3,1},
        {3,2} // Här utnyttjar vi ofullständig
              // initiering (12 av 20)
    }
};
```

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.*



"Pre-processing"

Pre-processor utför
textsubstitution!

```
// main.c
#include <stdio.h>
#define MAX_SCORE 100
#define SQUARE(x) (x)*(x)

int main()
{
    printf("Highest score is %d\n", MAX_SCORE);
    printf("Square of 3 is %d\n", SQUARE(3));
    return 0;
}
```

Kompilering (C-kod till assemblerkod)

Vid översättningen skapas
assemblerkod för den valda
processorn, här Intel X86

```
main.i
...
innehåll "stdio.h"
...
int main()
{
    printf("Highest score is %d\n", 100);
    printf("Square of 3 is %d\n", 3*3);
    return 0;
}
```

```
main.asm
main:
    push    %rbp
    mov     %rsp, %rbp
    sub     $0x20, %rsp

    printf@PLT("Highest score is %d\n", 100);
    lea     0x2a87(%rip), %rcx
    callq   402bb8@plt
    printf@PLT("Square of 3 is %d\n", 3*3);
    lea     0x9, %edx
    lea     0x2a8b(%rip), %rcx
    callq   402bb8@plt
    mov     %eax, %eax
    add     $0x20, %rsp
    pop     %rbp
    retq
```

Assemblering, assemblerkod till maskinkod

Översättningen från assemblerkod till objektкод
(maskinkod) görs av assemblern.

I objektkoden finns även symbolinformatik
externa funktioner och variabler

Länkning

Vid länkningen kombineras de olika objektfilerna till en
exekverbar fil.

Symboler översätts till relativa adresser.
Standardbiblioteket libc innehåller en lång rad användbara
funktioner, som exempelvis printf

Programstruktur

För att ge programmet en god struktur kan källtexterna delas upp i
olika delar.

- Filer med ändelsen .c, innehållet i dessa genererar så
småningom maskinkod.
- Filer med ändelsen .h (header-filer) används för deklarationer,
användardefinierade typer och makrodefinitioner. De sistnämnda
återkommer vi till

```
// main.c
#include <stdio.h>
#include "decl.h"

int main()
{
    int x;
    char c;
    ...
    printf("foo0= %d\n", foo0(x));
    foo1();
    printf("foo2= %c\n", foo2((short)x));
    return 0;
}
```

```
// decl.h
// user defined types
...
// macros
...
// function prototypes
int foo0(int x);
void foo1(void);
char foo2(short x);
```

```
// foo0.c
#include "decl.h"
// foo1.c
#include "decl.h"
// foo2.c
#include "decl.h"

// Function definition
char foo2(short x)
{
    ...
}
```

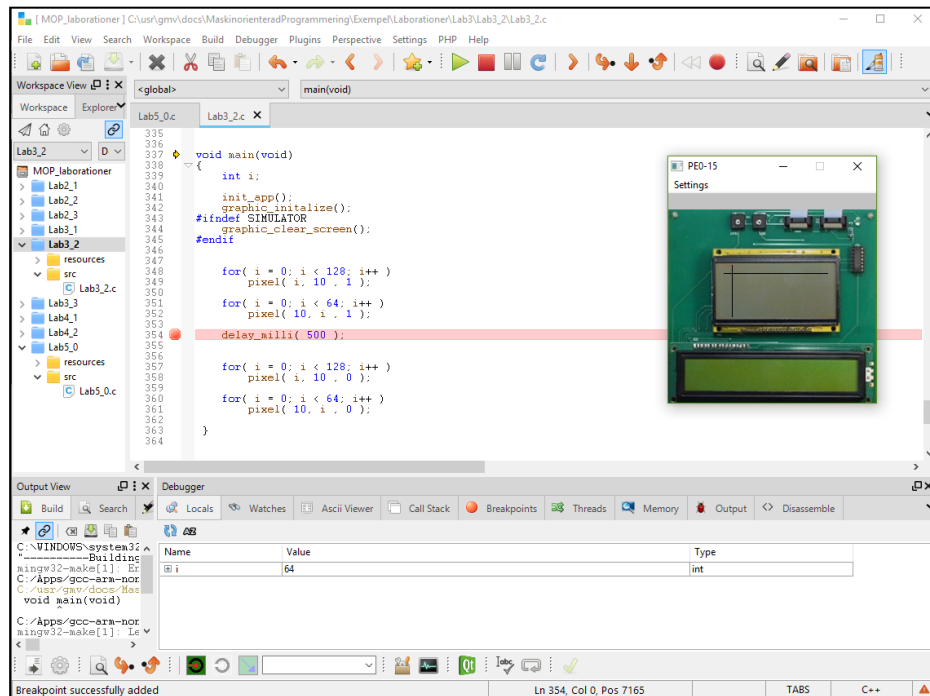


printf hämtas från programbibliotek libc:

Länkas
sammans till:

main.exe

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



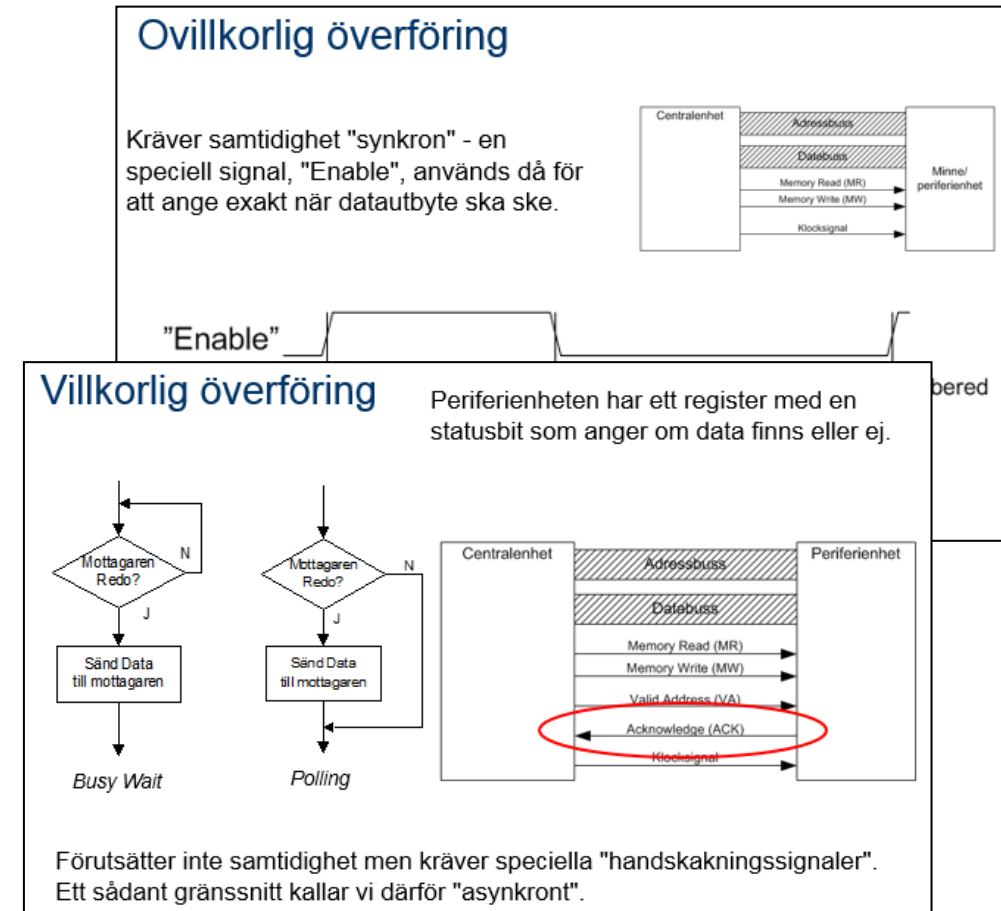
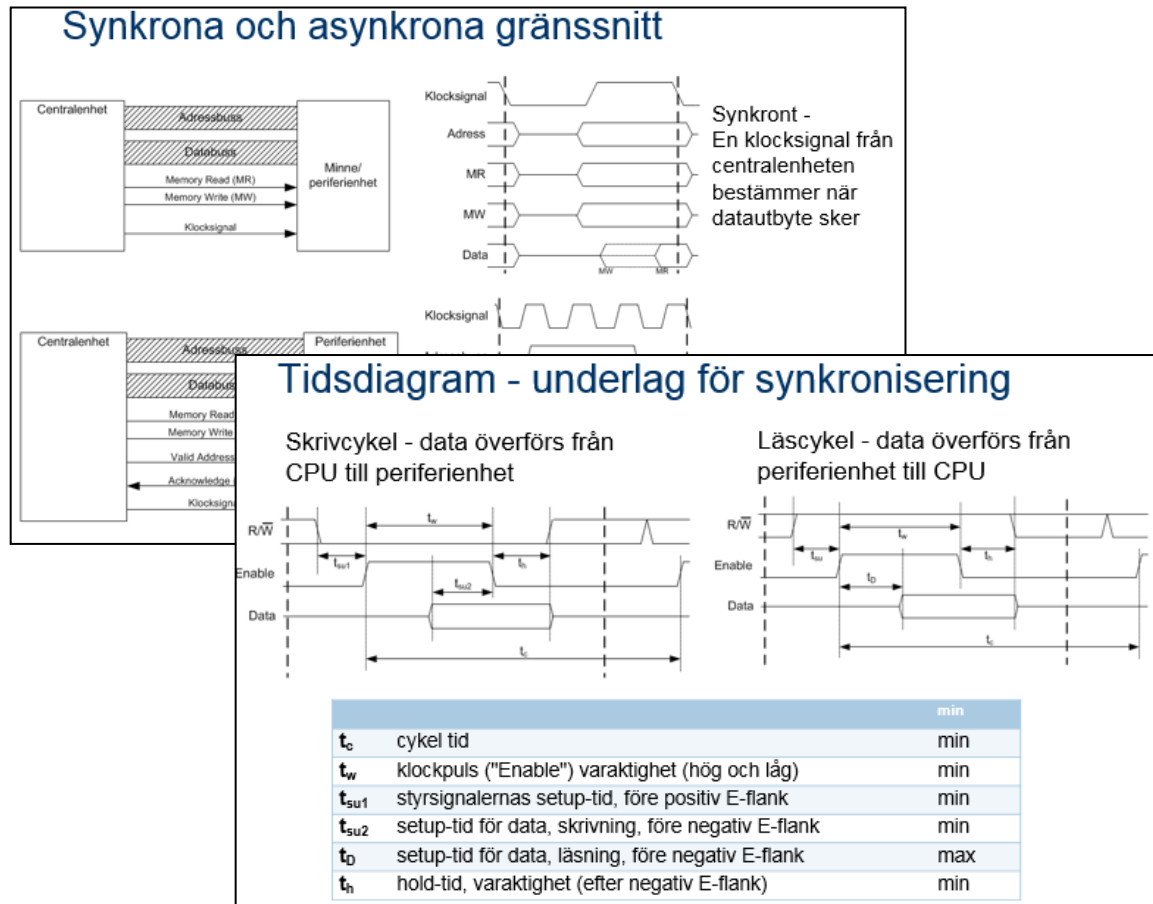
Dessa lärandemål har vi kontrollerat under laborationer.

3. Systemprogrammerarens bild

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.
- 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.

Vektortabellen

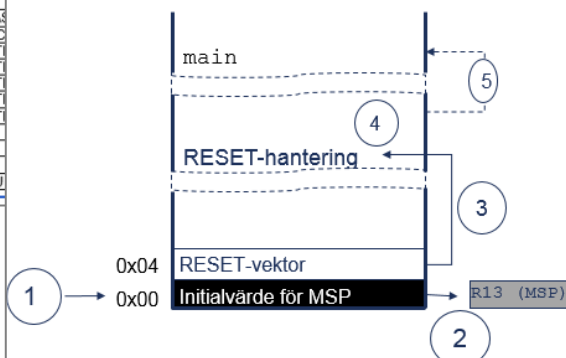
Anger position för de olika undantagsvektorer.

De 16 första positionerna är samma för alla Cortex-M4.

Resten av tabellen är specifik för den aktuella microcontrollern.

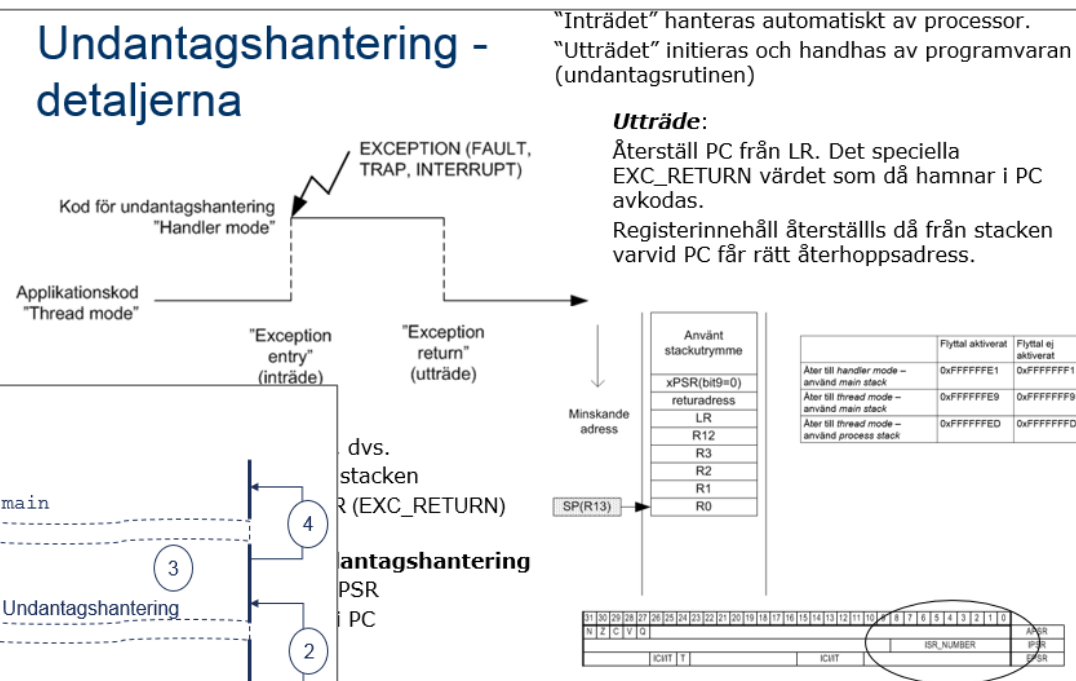
-3	fixed	Reset	Reserverad stack	0x0000 0000
-2	fixed	NMI	Reset	0x0000 0004
			Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-1	fixed	HardFault	All class of fault	0x0000 000C
0	settable	MemManage	Memory management	0x0000 0010
1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
			Reserved	0x0000 001C - 0x0000 002B
3	settable	SVCall	System service call via SWI instruction	0x0000 002C
4	settable	Debug Monitor	Debug Monitor	0x0000 0030
			Reserved	0x0000 0034
5	settable	PendSV	Pendable request for system service	0x0000 0038
6	settable	SysTick	System tick timer	0x0000 003C
7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
8	settable	PVD	PVD through EXTI line detection	0x0000 0044
2	9	settable	TAM	
3	10	settable	RTC	
4	11	settable	FLASH	
5	12	settable	RCC	
6	13	settable	EXTI	
7	14	settable	EXTI	
8	15	settable	EXTI	
9	16	settable	EXTI	
10	17	settable	EXTI	
81	88		FPU	

Exekvering vid undantag



- Återstart (Reset aktiverad)
- Ladda MSP (Main Stack Pointer) från adress 0x00
- Ladda RESET-vektor från adress 0x04
- RESET-hantering exekveras i "Thread mode"
- Systemets huvudprogram startas

Undantagshantering - detaljerna



"Inträdet" hanteras automatiskt av processor.
"Utrådet" initieras och handlas av programvaran (undantagsrutinen)

Utråde:

Återställ PC från LR. Det speciella EXC_RETURN värdet som då hamnar i PC avkodas.
Registerinnehåll återställs då från stacken varvid PC får rätt återhoppadress.

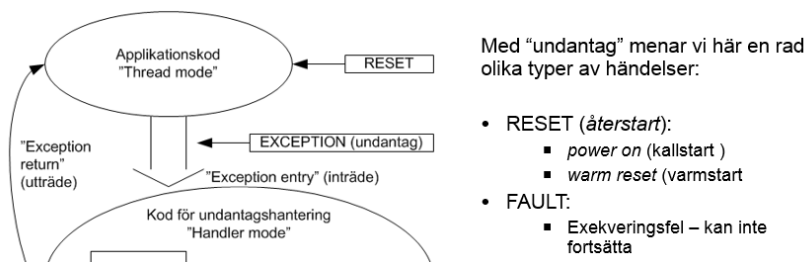
	Flyttal aktiverat	Flyttal ej aktiverat
Åter till handler mode – använd main stack	0xFFFFFE1	0xFFFFFFF1
Åter till thread mode – använd main stack	0xFFFFFE9	0xFFFFFFF9
Åter till thread mode – använd process stack	0xFFFFFED	0xFFFFFFF5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N Z C V Q																										APSR					
																ISR_NUMBER										IPSR					
ICVT T																ICVT										PCSR					

- Något undantag inträffar
 - Pågående instruktion utförs klart
 - Vektortabellen adresseras
- Den specifika undantagsvektorn hämtas
- Undantagshantering i "Handler Mode"
- Återgång efter undantagshantering

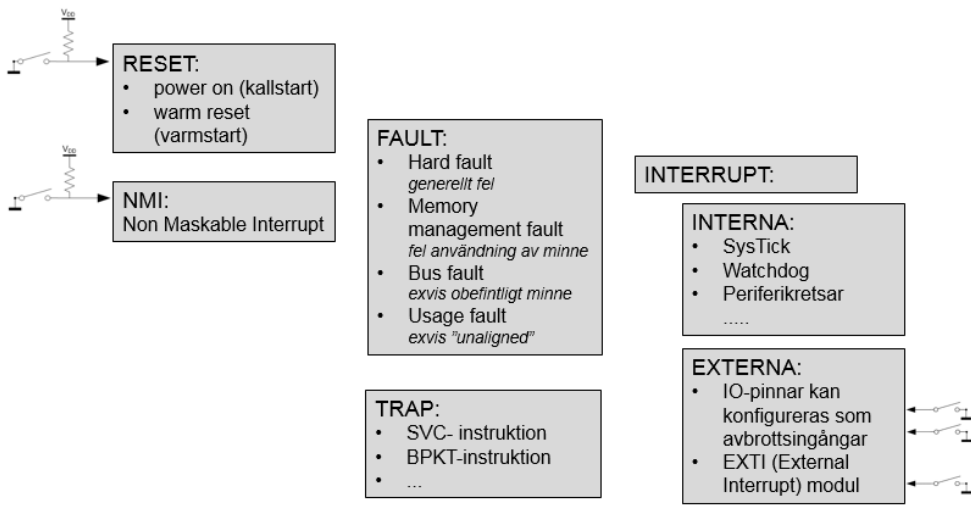
- *beskriva och exemplifiera olika undantagstyper: interna undantag, avbrott och återstart.*

Undantagshantering - "exception"

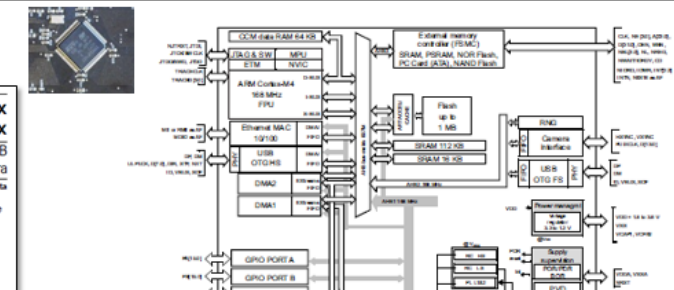
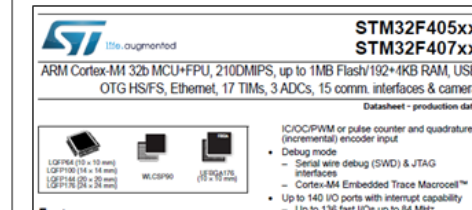


Undantagstyper

De olika undantagstyperna har en naturlig, fallande prioritet



Interna avbrott

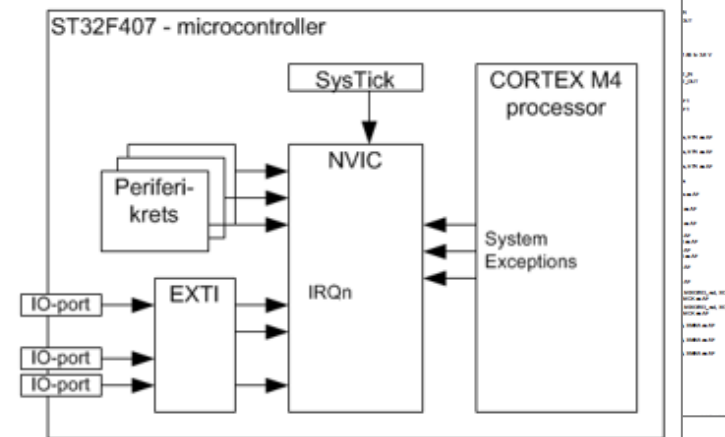


Externa avbrott

Externa avbrott är en form av undantag.

Registerinnehåll (xPSR, PC, LR, R12, R3-R0) sparas och återställs automatiskt av processorn

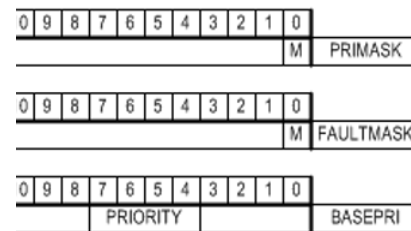
Avbrotts hanterare kan skrivas uteslutande med 'C'-kod.



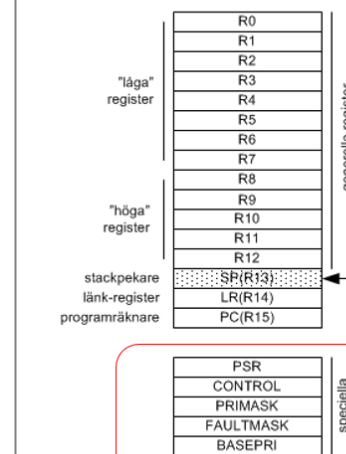
6	settable	SysTick	System tick timer
7	settable	WWDG	Window Watchdog interrupt
8	settable	PVD	PVD through EXTI line detection interrupt
9	settable	TAMP_STAMP	Tamper and TimeStamp interrupts through the EXTI line
10	settable	RTC_WKUP	RTC Wakeup interrupt through the EXTI line
11	settable	FLASH	Flash global interrupt
12	settable	RCC	RCC global interrupt
13	settable	EXTI0	EXTI Line0 interrupt
14	settable	EXTI1	EXTI Line1 interrupt
15	settable	EXTI2	EXTI Line2 interrupt
16	settable	EXTI3	EXTI Line3 interrupt
17	settable	EXTI4	EXTI Line4 interrupt
18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt
19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt

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

				Reserverad stack
-3	fixed	Reset		Reset
-2	fixed	NMI		Non maskable interrupt
				Clock Security System to the NMI vector.
-1	fixed	HardFault		All class of fault
0	settable	MemManage		Memory management
1	settable	BusFault		Pre-fetch fault, memory access fault
2	settable	UsageFault		Undefined instruction
				Reserved
3	settable	SVCall		System service call via exception
4	settable	Debug Monitor		Debug Monitor
				Reserved
5	settable	PendSV		Pendable request for system tick
6	settable	SysTick		System tick timer
7	settable	WWDG		Window Watchdog interrupt
8	settable	PVD		PVD through EXTI line
9	settable	TAMP_STAMP		Tamper and TimeStamp
10	settable	RTC_WKUP		RTC Wakeup interrupt
11	settable	FLASH		Flash global interrupt
12	settable	RCC		RCC global interrupt
13	settable	EXTI0		EXTI Line0 interrupt
14	settable	EXTI1		EXTI Line1 interrupt
15	settable	EXTI2		EXTI Line2 interrupt
16	settable	EXTI3		EXTI Line3 interrupt
17	settable	EXTI4		EXTI Line4 interrupt
88		FPU		FPU global interrupt

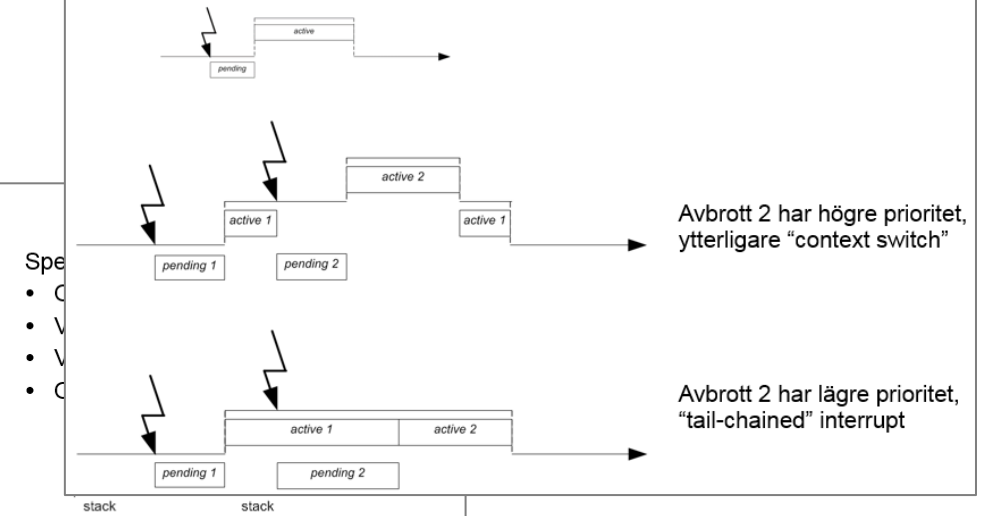


Register



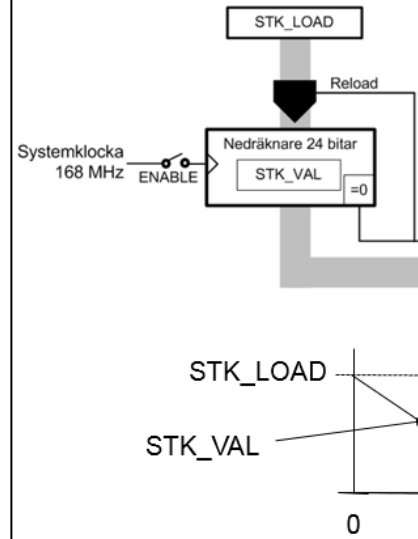
Avbrottsprioritet

Flera avbrott, pending, active och prioritet



- *beskriva och använda kretsar för tidmätning.*

Räknarkrets - "SysTick"



offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0																																	STK_CTRL
4																																	STK_LOAD
8																																	STK_VAL
0xC	r	r																															STK_CALIB

Räknarkrets - register

STK_CTRL (0xE000E010) Status och styrregister

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0																rw																

STK_LOAD (0xE000E014) Räknarintervall

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	
4									r	w	r	w	r	w	r	w	r	w	r	w	r	w	r	w

STK_VAL (0xE000E018) Räknarvärde

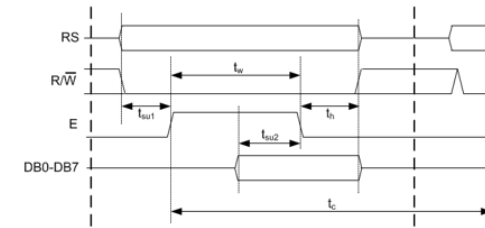
offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
8																																	STK_VAL

Algoritm:

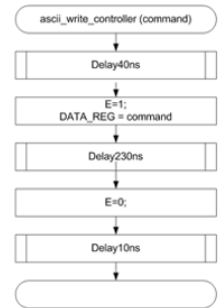
```
STK_CTRL = 0      Återställ SysTick
STK_LOAD = CountValue
STK_VAL = 0       Nollställ räknarregistret
STK_CTRL = 5      Starta om räknaren
Vänta till COUNTFLAG=1
STK_CTRL = 0      Återställ SysTick
```

Skrivcykel - karakteristika

Databladets tidsdiagram illustrerar hur styrsignaler och data ska läggas ut



Algoritm:

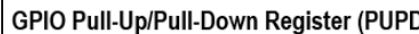


	min
t_c cykel tid	500 ns
t_w klockpuls ("Enable") varaktighet (hög och låg)	230 ns
t_{su1} styrsignalernas setup-tid, före positiv E-flank	40 ns
t_{su2} setup-tid för data, skrivning, före negativ E-flank	80 ns
t_h hold-tid, varaktighet (efter negativ E-flank)	10 ns

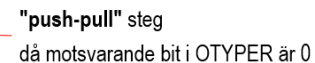
- ## Parallell överföring



Vi kan programmera "pull-up" eller "pull-down"-funktion, inget externt motstånd behövs.



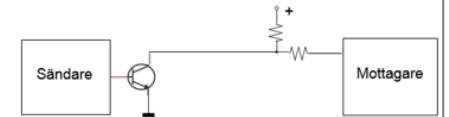
IO-pinne konfigurerad som utgång



"open drain" steg
då motsvarande bit i OTYPE är 1

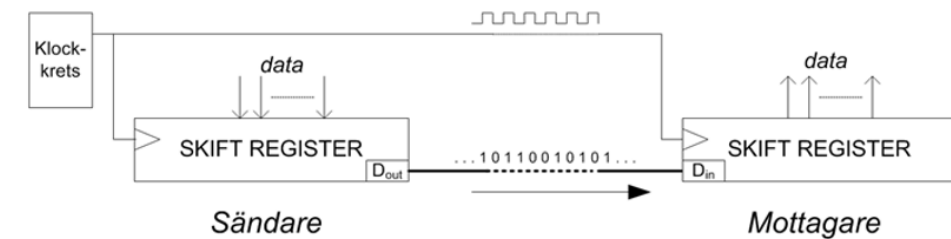
GPIO Output Type Register																																mnemonic		
offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0x04																																		GPIO_OTYPER
																	0x05				0x04													

Seriell överföring



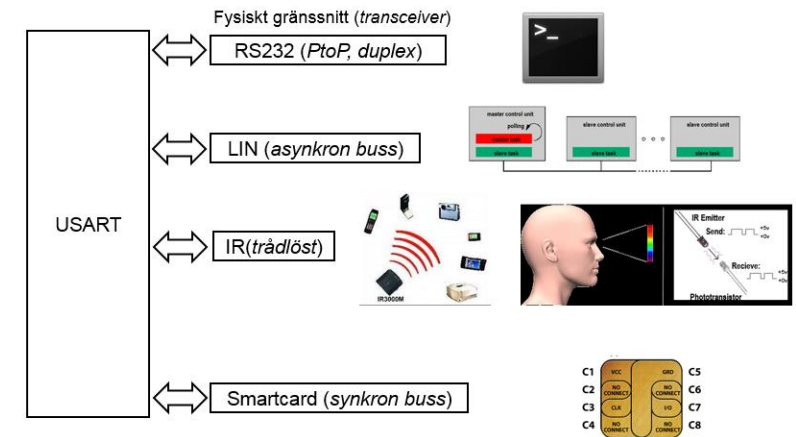
Synkron överföring

Klocksignal och data på skilda ledningar



Kräver

USART – Universal Synchronous/Asynchronous Receiver/Transmitter



Av speciell vikt: "maskinnära programmering..."

Föredrödesvis i 'C' men också i assemblyspråk

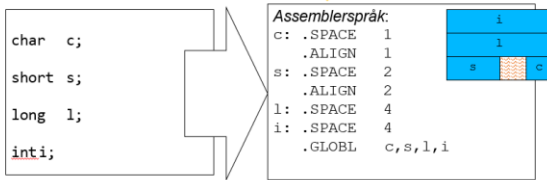
- Läsa/skriva på fasta adresser i minnet (portar)
- Läsa/skriva processorns register
- Pekarhantering
- Heltalstyper, storlek (8,16 eller 32 bitar...) med/utan tecken
- Bitoperationer $\&$, $|$, $\wedge$ (AND, OR, XOR)
- Skiftoperationer $\ll$, $\gg$ (vänster, höger)

Assemblerprogrammering: översätta C-deklarationer (heltalstyper, fält och sammansatta typer) till assemblerspråk och att koda uttrycksevaluering i assemblerspråk.

Nytt för i år: flervälsfrågor...

Deklarationer

```
char c; /* 8-bitars .ALIGN 1 @ align to half (2 byte)
short s; /* 16-bitars .ALIGN 2 @ align to word (4 byte)
long l; /* 32-bitars .ALIGN 4 @ align to word
int i; /* 32-bitars data, .CORLEK word */
```



Assemblerspråk:
c: .SPACE 1
 .ALIGN 1
s: .SPACE 2
 .ALIGN 2
l: .SPACE 4
i: .SPACE 4
 .GLOBL c,s,l,i

Registeranvändning

I princip kan man använda de generella registren som man vill men det är mycket bättre att följa ARM's rekommendationer:

Register	Användning
R15 (PC)	Programräknare
R14 (LR)	Länkregister
R13 (SP)	Stackpekare
R12 (IP)	
R11	Dessa register är avsedda för variabler och som temporära register.
R10	Om dom används måste dom sparas och återställas av
R9	den anropade (callee) funktionen
R8	
R7	Speciellt använder GCC R7 som pekare till aktiveringspost (stack frame)
R6	Också dessa register är avsedda för variabler och temporärbruk
R5	Om dom används måste dom sparas och återställas av
R4	den anropade (callee) funktionen
R3	parameter 4 / temporärregister
R2	parameter 3 / temporärregister
R1	parameter 2 / resultat 2 / temporärregister
R0	parameter 1 / resultat 1 / temporärregister

EXEMPEL: Vi har deklarationerna

```
short a,b;
int x,i,j;
int ai[32][8]
```

på "toppnivå". Visa hur tilldelningarna:

a) $x = (a+b) * (a-b)$

b) $ai[i][j] = x$

kodas i ARMv6 assemblerspråk.

EXEMPEL:

Vi har följande typdeklaration:

```
struct coord;
typedef struct {
    unsigned int a;
    unsigned char b;
    unsigned short c;
    struct coord *d;
} ST;
```

Visa hur följande deklaration görs i ARM v6 assemblerspråk.

ST point;

Visa en kodsekvens som evaluerar följande uttryck i register R0.

point.a = point.b + point.c;

EXEMPEL: Vi har deklarationerna:

```
signed char a;
signed short *b;
unsigned int c;
signed char foo( unsigned int );
```

Visa hur såväl variabeldeklarationerna som följande funktionsanrop kodas i ARMv6 assemblerspråk.:

*b = (signed short) foo(a+c);

Assemblerprogrammering: analysera översättningar av enklare C-kod till assemblerspråk, med kompilatorkonventioner.

Nytt för i år: flervalfrågor...

Följande C-funktion pack packar födelsedata (år, månad dag) i en unsigned int, enligt:

31	30	29	28	27	26	25	34	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	year												month				day					

```
int pack( unsigned short year, unsigned short month, unsigned short day)
{
    unsigned int rval = year << 9;
    rval |= month << 5;
    rval |= day;
    return rval;
}
```

Ange det alternativ (A-G) som visar en korrekt implementering (hur funktionen kodas) i ARM v6 assemblerspråk.

A	B	C	D
pack: SHL R0,R0,#9 SHL R1,R1,#5 ORR R0,R0,R2 ORR R0,R0,R1 BX LR	pack: LSL R0,R0,#9 LSL R1,R1,#5 ORR R0,R0,R1 ORR R0,R0,R2 BX LR	pack: LSR R0,R0,#9 LSR R1,R1,#5 ORR R0,R0,R1 ORR R0,R0,R2 BX LR	pack: LSL R0,R0,#9 LSL R1,R1,#5 OR R0,R0,R1 OR R0,R0,R2 BX LR
E	F	G	
pack: SHL R2,R2,#9 SHL R1,R1,#5 ORR R2,R2,R1 ORR R0,R0,R2 BX LR	pack: LSL R2,R2,#9 LSL R1,R1,#5 ORR R2,R2,R1 ORR R0,R0,R2 BX LR	pack: LSR R2,R2,#9 LSR R1,R1,#5 ORR R2,R2,R1 ORR R0,R0,R2 BX LR	

Följande deklarationer är givna på "toppnivå".

static unsigned short a,b;

static int index;

static int vi[100];

Vi har nu tilldelningen:

vi[a++] = index;

Vilken av följande implementeringar i ARM v6-kod är korrekt?

A		
LDR R0,=index LDR R0,[R0] LDRH R1,=a LDR R2,[R1] ADD R3,R2,#1 STRH R3,[R1] LSL R2,R2,#4 LDR R3,vi STR R0,[R3+R2]	ADD R3,R2,#1 STR R3,[R1] LSL R2,R2,#2 LDR R3,=vi STR R0,[R3,R2]	ADD R3,R2,#1 STRH R3,[R1] LSL R2,R2,#2 LDR R3,=vi STR R0,[R3,R2]
D	E	F
LDR R0,=index LDR R0,[R0] LDR R1,=a LDRH R2,[R1] ADD R3,R2,#1 STRH R3,[R1] LSL R2,R2,#4 LDR R3,=vi STR R0,[R3,R2]	LDR R0,=index LDR R0,[R0] LDR R1,=a LDRH R2,[R1] ADD R3,R2,#1 STRH R3,[R1] LSL R2,R2,#4 LDR R3,=vi STR R0,[R3+R2]	LDR R0,=index LDR R0,[R0] LDRH R1,=a LDR R2,[R1] ADD R3,R2,#1 STRH R3,[R1] LSL R2,R2,#4 LDR R3,vi STR R0,[R3+R2]

Pekare och deras användning, programmeringsuppgifter *eller* kortare frågor.

Pekararitmetik

Följande operationer är tillåtna på pekare:

- Adressoperator
- Dereferens
- Addition av heltalskonstant
- Subtraktion av heltalskonstant
- Subtraktion av pekare med samma typ

Alla andra operationer, som exempelvis skift, negation, bitvis komplement, multiplikation eller division etc. är meningslösa och därför förbjudna.

Exempel 2:

"Volatile qualifier" - bestämning/tillägg till deklaration

volatile förhindrar viss optimering (som annars är bra och nödvändig) dvs. indikerar att kompilatorn måste förmoda att innehållet på en adress kan ändras "från utsidan" (dvs. en ändring kan inte upptäckas vid statisk analys av koden).

Exempel 1:
`char c;
cp1 =
cp2 =`

Pekare till struct, pilnotation

Det är vanligt att använda pekare till struct.
Eftersom en derefererad pekare utgör ett klumpigt skrivsätt har man infört pilnotation.

```
volatile char *  
void foo(){  
    while(*import  
{  
}
```

Exempel:

Exempel: Koordinater i en länkad lista

Tilldelning från fält

Exempel: Vi har deklarationerna:

```
char c; int i;  
char vec[15];
```

Visa hur följande tilldelning kan kodas:

THUMB assembler

```
R0, =vec  
R1, i  
R0, [R0, R1]  
R1, =c  
R0, [R1]
```

Referens av objekt i fält

Exempel: Vi har deklarationerna:

```
int i, j;  
int veci[80];  
int vm[10][5];
```

Visa kodsekvenser som evaluerar följande uttryck till register R0.

a) `veci[j]`

Adress:
`=veci + j*sizeof(int)`

b) `vm[i][j]`

Adress:
`=vm + ((i*5)+j)*sizeof(int)`

Vi löser på tavlan...

A
`unsigned short*
ConvertList(unsigned int* a, int* num_elems)
{`

En lista av objekt med typen `unsigned int` skall konverteras till `unsigned short`. Om talet är för stort för att få plats i en `unsigned short` skall det inte vara med i den nya listan. Funktionen `ConvertList` skall returnera pekaren till det första elementet i den nya listan, och parametern `num_elems` skall uppdateras med den nya storleken. Huvudprogrammet till höger skall exempelvis ge utskriften: 345 2333 123.

B
`unsigned short*
ConvertList(unsigned int* a, int* num_elems)
{
 unsigned short* b = (unsigned short*)a;
 int c = 0;
 for (int i = 0; i < (*num_elems); i++) {
 if (a[i] >= (1 << 16)) { c++; }
 else {
 *b = a[i];
 b += 2;
 }
 }
 *num_elems -= c;
 return (unsigned short*)a;
}`

D
`unsigned short*
ConvertList(unsigned int* a, int* num_elems)
{
 unsigned short* b = (unsigned short*)&a[0];
 int c = 0;
 for (int i = 0; i < (*num_elems); i++) {
 if (a[i] >= (1 << 16)) { c++; }
 else {
 *b = a[i];
 b += 1;
 }
 }
 *num_elems -= c;
 return (unsigned short*)a;
}`

E
`unsigned short*
ConvertList(unsigned int* a, int* num_elems)
{
 unsigned short* b = (unsigned short*)a;
 int c = 0;
 for (int i = 0; i < num_elems; i++) {
 if (a[i] >= (1 << 16)) { c++; }
 else {
 *b = a[i];
 b += 1;
 }
 }
 num_elems -= c;
 return (unsigned short*)a;
}`

F
`unsigned short*
ConvertList(unsigned int* a, int* num_elems)
{
 unsigned short* b = (unsigned short*)&a[0];
 int c = 0;
 for (int i = 0; i < (*num_elems); i++) {
 if (a[i] >= 0xFFFF) { c++; }
 else { *b++ = *a++; }
 }
 *num_elems -= c;
 return (unsigned short*)a;
}`

Nytt för i år: flervälsfrågor...

Pekare och deras användning, beskrivning/användning av sammansatta typer för portar

Inkapsling av data, ett "objekt"

struct ("post"), en sammansatt datatyp

- Har en eller flera medlemmar (*fields*) av godtycklig typ, exempelvis:
 - `int`, `char`, `long` (signed/unsigned), `float`, `double`
 - Fält och textsträngar
 - Alla typer av pekare
 - Tidigare deklarerad typalias (med "typedef")
 - Sammansatt typ (dvs. en annan `struct`).

Exempel:

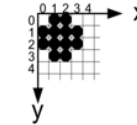
```
struct tPoint
{
    int x,y;
};
```

Exempel:

```
struct tLine
{
    int linje_nr;
    struct tPoint start;
    struct tPoint end;
};
```

Deklarationssyntax:

```
struct structnamn{
    medlem;
    [medlem; [medlem;]... ]
};
```



Exempel: Arbetsboken, avsnitt 3.4

```
// Skapa och initiera ett object av typen GEOMETRY:
GEOMETRY ball_geometry = {
    12, 4, 4,
    { // POINT
        {0,1},
        {0,2},
        {1,0},
        {1,1},
        {1,2},
        {1,3},
        {2,0},
        {2,1},
        {2,2},
        {2,3},
        {3,1},
        {3,2}
    } // (1
};
```

Poster med funktionspekare

Inkapsling av pekare till funktioner.

Exempel:

```
typedef struct tObj {
    PGEOMETRY geo;
    int dirx, diry;
    int posx, posy;
    void (*draw)(struct tObj *);
    void (*clear)(struct tObj *);
    void (*move)(struct tObj *);
    void (*set_speed)(struct tObj *, int, int);
} OBJECT, *POBJECT;
```

draw, clear, move och set_speed är pekartyper.

Kan uppfattas som "metoder" i objektorienterat språk.

Portadressering

Exempel:
GPIO port E

```
#define GPIO_E 0x40021000
#define GPIO_E_MODER ((volatile unsigned int *) (GPIO_E))
#define GPIO_E_OTYPER ((volatile unsigned short *) (GPIO_E+4))
#define GPIO_E_OSPEEDR ((volatile unsigned int *) (GPIO_E+8))
...

#define GPIO_E_ODR ((volatile unsigned short *) (GPIO_E+0x14))
#define GPIO_E_ODR_LOW ((volatile unsigned char *) (GPIO_E+0x14))
#define GPIO_E_ODR_HIGH ((volatile unsigned char *) (GPIO_E+0x15))
...

value = *GPIO_E_ODR_LOW; // Läs från 0x40021014
*GPIO_E_ODR_HIGH = value; // Skriv till 0x40021015
```

Portadressering med poster

struct-typen är också användbar för att deklarerar portar

```
// Som alternativ till :
#define portModer ((volatile unsigned int *) (GPIO_BASE))
#define portOTyper ((volatile unsigned short *) (GPIO_BASE+0x4))
#define portOSpeedr ((volatile unsigned int *) (GPIO_BASE+0x8))
ovr.

// kan vi använda en post-definition som:
typedef volatile struct {
    unsigned int moder;
    unsigned int otyper; // +0x4
    unsigned int ospeedr; // +0x8
    unsigned int pupdr; // +0xC (12)
    unsigned int idr; // +0x10
    unsigned int odr; // +0x14
    unsigned int bsrr; // +0x18
    unsigned int ickr; // +0x1C
    unsigned int afr1; // +0x20
    unsigned int afrh; // +0x24
} GPIO, *PGPIO;
```

Exempel:

```
#define GPIO_D (*(volatile PGPIO) 0x40020c00)
#define GPIO_E (*(volatile PGPIO) 0x40021000)

GPIO_E.moder = 0x55555555;
GPIO_D.otyper = 0x00000000;
GPIO_D.moder = 0x55555000;
GPIO_D.pupdr |= 0x00005555;
```

Adressering med bitfält

```
// GPIO
typedef volatile struct tag_gpio {
    ...
    union {
        uint32_t idr;
        struct {
            uint8_t idrLow;
            uint8_t idrHigh;
            uint8_t col;
        };
        short reserved;
    };
    union {
        uint32_t odr;
        struct {
            uint8_t odrLow;
            uint8_t odrHigh;
            uint8_t unused;
            uint8_t row;
        };
        short reserved;
    };
    ...
} GPIO;
```

```
void kbdActivate( unsigned int row )
{
    switch( row )
    {
        case 1: *GPIO_D_ODRIGH = 0x10; break;
        case 2: *GPIO_D_ODRIGH = 0x20; break;
        case 3: *GPIO_D_ODRIGH = 0x40; break;
        case 4: *GPIO_D_ODRIGH = 0x80; break;
        case 0: *GPIO_D_ODRIGH = 0x00; break;
    }
}

void kbdDeact( unsigned int row )
{
    switch( row )
    {
        case 1: GPIO_D.col = 1; break;
        case 2: GPIO_D.col = 2; break;
        case 3: GPIO_D.col = 4; break;
        case 4: GPIO_D.col = 8; break;
        case 0: GPIO_D.col = 0; break;
    }
}
```

Tillämpningar för laborationssystemet MD407, variationer kring laborationsuppgifterna.

En tidtagaranläggning med tre oberoende tidtagarur (Clock1, Clock2, Clock3) ska konstrueras. Tid ska visas som 4 hexadecimala siffror på två moduler av typ "7-segment display" med upplösningen 10 ms. Alltså kan ett 16 bitars tal visas på de båda displayerna där den mest signifikanta delen ansluts till GPIO port D15-8 och den minst signifikanta delen ansluts till GPIO port D7-0. Det kan förutsättas att en tidmätning aldrig överskrider en unsigned short (dvs. c:a 11 minuter).



Anläggningen styrs via en 8-polig DIL-omkopplare, kopplad till GPIO port E7-0 enligt följande:

	b7-b6: Väljer visning: 00: visar '0000' 01: Clock1 10: Clock2 11: Clock3	b5-b3: Clock "reset": b5=1: Clock1 RESET b4=1: Clock2 RESET b3=1: Clock3 RESET	b2-b0: Clock start/stop b2: Clock1 b1: Clock2 b0: Clock3 Startar vid positiv flank Stoppar vid negativ flank
--	--	---	---

Konstruera programpaketet för tidtagaranläggningen. Start/stopp-funktionen ska implementeras med hjälp av avbrottsmekanismer. För full poäng ska du dessutom använda lämpliga definitioner av typer och makron så som anvisats under kursen.

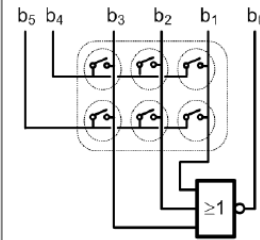
Ledningar:

Låt PE2, PE1 och PE0 generera avbrott både vid positiv och negativ flank.

Inför variabler som anger om en klocka är startad eller stoppad. Uppdatera endast startade klockor vid avbrott.

- Definiera de symboliska adresser med lämpliga typkonverteringar som krävs för uppgiften. Deklarera också lämpliga globala variabler. (3p)
- Visa en funktion `void portInit(void)` som initierar GPIO-portarna. (2p)
- Använd SYSTICK för att skapa en realtidsklocka som genererar avbrott med 10 ms intervall. Vid varje avbrott ska de klockor som är startade uppdateras. Systemets klockfrekvens är 168 MHz. Två funktioner ska implementeras (4p):
 - `void systickInit(void)` som gör alla nödvändiga initieringar och
 - `void systick_irq_handler(void)` som hanterar avbrotten från SYSTICK.
- Använd EXTI (0,1,2) för att implementera start/stopp- funktionerna. Tre avbrottsrutiner och en initieringsfunktion ska implementeras (6p):
 - `void extiX_irq_handler( void )` X=0,1,2 hanterar de olika avbrotten
 - `void extiInit(void)` gör alla nödvändiga initieringar för att använda PE-portpinnar för avbrott.
- Konstruera ett huvudprogram som: Initierar systemet med de specificerade initieringsfunktionerna och därefter, kontinuerligt, skriver ut det valda klockvärdet till 7-segmentsdisplayerna. En klocka som hålls i RESET ska stoppas och nollställas.(5p)

Ett tangentbord för inmatning av sex olika tecken ska konstrueras. Sex stycken återfjädrande omkopplare ansluts därför till port E hos MD407, på följande sätt:



Bit 0-3 kopplas till ingångar, bit 4 och bit 5 kopplas till utgångar hos port E.

Nedtryckta tangenter kan detekteras genom att '1' skrivs till någon av bit 4 eller bit 5, därefter avläses bit 3-bit 1. För att detta ska vara tillförlitligt måste dessa bitars ingångar förses med "pull-down" samtidigt som utgångarna ska vara "push-pull". Metoden kallas *koincidensavsökning*.

Kretsen kan också användas för att känna av en tangentnedtryckning med hjälp av avbrott. För detta kopplas kolumnerna till en NOR-grind, vars utgång är kopplad till bit 0 hos port E, denna måste ha "pull-up".

Anm: Vi påminner om NOR-grindens funktion: utsignalen är '1' så länge alla ingångar är '0'. Då någon av ingångarna blir '1' blir utsignalen '0'.

- Visa en initieringsrutin `void init( void )` (8p) där:
 - GPIO modulen initieras för dessa portpinnar. Port E ska initieras för användning med tangentbordet. Bitarna b6 och b7 används inte, observera att endast konfigurationen för des portpinnar som används får ändras vid konfigureringen.
 - SYSCFG, EXTI och NVIC konfigureras för avbrott via b0.
 - Avbrottsvektor initieras med adress till avbrottsfunktionen `void at_interrupt(void)`. Antag att vektortabellen börjar på adressen 0x2001C000 i minnet.
- Visa avbrottsrutinen `void at_interrupt( void )` som kontrollerar om avbrott begärs via b0 och i så fall kvitterar avbrottet. (2p)

Följande port som utgör ett gränssnitt mot en yttre periferienhet är placerad på adress 0xFF600000:

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0																																
4																																
8																																
0xC																																

Bitarna IRQ och ER ingår alltså i status-delen av registret medan biten RS och bitfältet COLUMN ingår i ctrl-delen av registret med offset 0.

- Visa lämpliga makrodefinitioner för referens (åtkomst) av portens register status respektive data. (2p)
- Visa med en typdefinition PORT, hur porten kan avbildas med en *struct*-definition. (2p)
Visa speciellt hur delen data då refereras med hjälp av din typdefinition (2p).

Utformning av tentamen

Endast 7,5 hp, EDA483, DIT153

Tentamen består av uppgifter hämtade i första hand från följande problemområden:

- Översättning av C till assemblerspråk
 - ✓ hur tillämpas kompilatorkonventioner (parametrar och returvärden) ?
 - ✓ hur kodas evaluering av uttryck i assemblerspråk ?
 - ✓ hur kodas enkla funktioner i assemblerspråk **och hur kan man styra kodgenerering hos en C-kompilator ?**
- Maskinnära programmering i C
 - ✓ hur används typsystemet för att skapa deklARATIONER av portar (sammansatta typer och bitfält) ?
 - ✓ vad menas med *pekare* och hur kan dom användas ?
 - ✓ **hur använder man och skapar nya programbibliotek?**
- Konfigurering och användning av systemenheter och periferikretsar
 - ✓ hur konfigureras GPIO för digital in- och utmatning ?
 - ✓ hur används systemenheten SYSTICK (med eller utan avbrott) för synkronisering i realtid ?
 - ✓ hur konfigureras SYSCFG, EXTI och NVIC för användning med externa enheter ?

Typexempel hittar du i exempelsamlingen såväl som tidigare tentamina (se kursens hemsida).