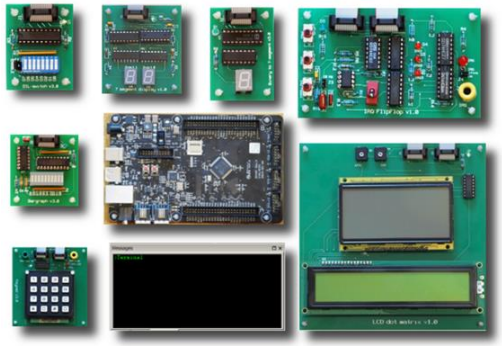
Maskinorienterad programmering

Sammanfattning

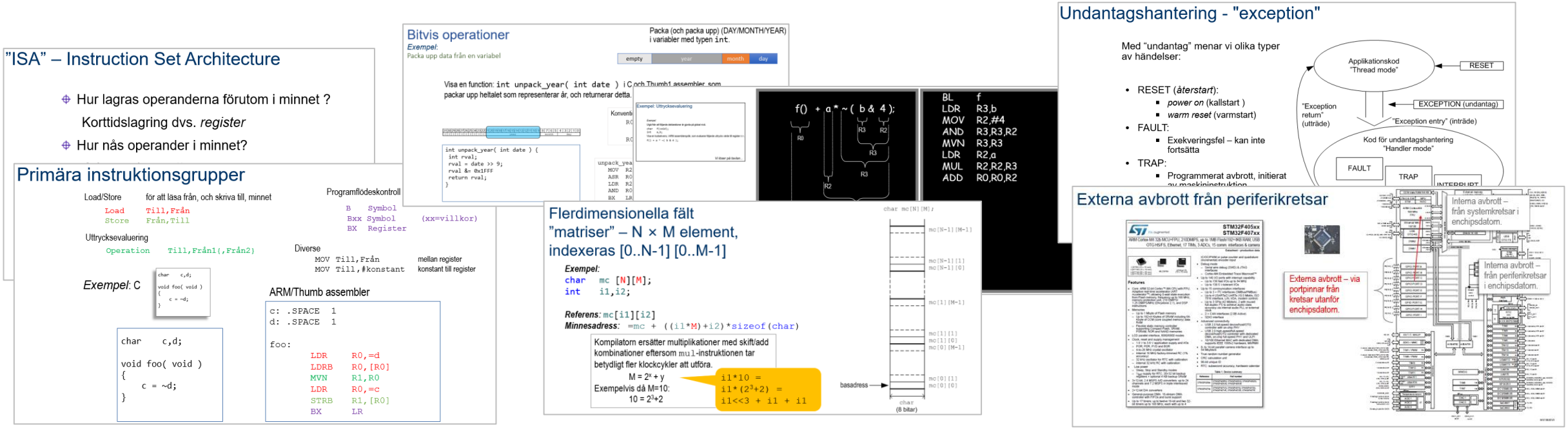
Ur innehållet:

- 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.



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.*

Funktioners parametrar och returvärden.

C-funktioner, parametrar och returvärden

Parametrar ("argument")

```
void foo(int x, char y)
{
   int sum;

   // Statements

}
```

Returvärde har funktionens typ

```
int foo(int x, char y)
{
   int sum;

   // Statements
   return sum;
}
```

Parametrar skickas med värdeanrop ("pass-by value")

Exempel:

```
char var2 = 7;
foo(5, var2);
```

var2 har fortfarande värdet 7 efter funktionen

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
Funktionen
`signed`
och de glob
`signed`
är definiera
Visa hur fu
`z = f(`
kodas i ass

Funktionens returvärde

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

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
```

Lagrings-klasser och synlighet.

Variabler och minnesanvändning

Ett deklarerat objekt
karakteris
- Synlighe
- Varaktigl
- Typ av b

Synlighet

Synligheten kan begränsas av deklarationens block.

fur
synlig
der
filens

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 uttrycksevaluering 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	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 — Dessa register sparas a

- *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.	
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	Dessa register sparas normalt sett
R2	parameter 3 / temporärregister	inte över funktionsanrop men om,

Funktionsparametrar

Konventionerna sägar 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:

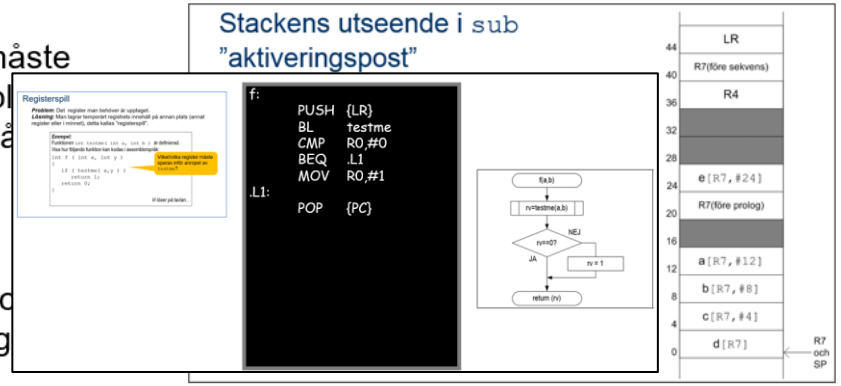
```
LDR R0,a
LDR R1,b
LDR R2,c
LDR R3,d
BL  sub
```

När inte registren räcker till

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

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

Stackens utseende i `sub` "aktiveringspost"



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

sub(p1,p2,...) — Inträde i och utträde ur subrutiner.
Inträde, typiskt med subrutinens namn och symbolisk representation av eventuella parametrar då sådana finns.
RETUR(rv) — **Utträde**, ("RETUR") typiskt med angivande av returvärde (rv) om sådant finns.

subcall(param, ...) — Subrutinanrop.
Med parametrar, symbolisk representation av eventuella aktuella parametrar som skickas med subrutinanropet.
id←subcall — **Med returvärde**, tilldelningsoperator placeras framför den anropade subrutinen.

Initieringar — Engångsinitieringar.
Placeras typiskt i omedelbar anslutning till en inträdessymbol

Sats 1
Sats 2 ; Sats 3
etc... — Exekveringsblock.
En eller flera satser som ordnats och exekveras sekvensiellt.

sant

Subrutiner ("funktioner")

C-operator	Betydelse	Operation	Instruktion	RTN
f()	funktionsanrop	Branch and link	BL <label>	LR←PC, PC←label
"return"		Branch and exchange	BX Rx	PC←Rx

EXEMPEL:

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

sub:
```

sub

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

Returadressen sparas i regi

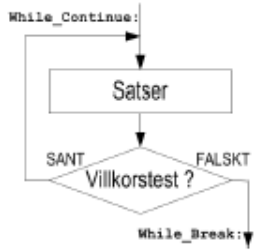
Kontrollstrukturer

```
if( villkor )
{
  Satser
}
```


```
if( villkor ){
   Satser1
}else{
   Satser2
}
```


```
while( villkor )
{
  Satser
}
```


```
do{
  Sats
} whil
```



Instruktioner för villkorlig programflödeskontroll

C-operator	Betydelse	Datatyp	Instruktion
==	Lika med	signed/unsigned	BEQ
!=	Skild från	signed/unsigned	BNE
<	Mindre än	signed	BLT
		unsigned	BCC
<=	Mindre än eller lika	signed	BLE
		unsigned	BLS
>	Större än	signed	BGT
		unsigned	BHI
>=	Större än eller lika	signed	BGE
		unsigned	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`
`short int`
`signed short`
`signed short int`
	Kort heltalstyp med tecken. Måste **minst** kunna representera talområdet [−32767, +32767], dvs. minst 16 bitar. Observera att talområdet [−32768, +32767] som fås vid 2-komplementsrepresentation, också är tillåtet.

`unsigned short` `unsigned short int`	Kort heltalstyp utan tecken. Måste **minst** kunna representera talområdet [0, +65535], dvs. minst 16 bitar.

`long`
`long int`
`signed lo`
`signed lo`
`unsigned`
`unsigned`
`int`
`signed`
`signed in`
	Lång heltalstyp med tecken. Måste **minst** kunna representera talområdet [−2147483647, +2147483647], dvs. minst 32 bitar.

`unsigned` `unsigned`	

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

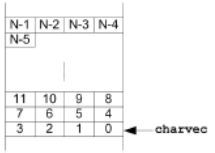
***EXEMPEL*:**

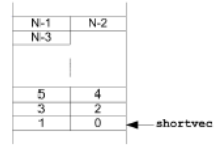
```
cptr = ( char  *) 0x40021010
sptr = ( short *) 0x40021010
iptr = ( int   *) 0x40021010
```

Vad är skillnaden?

Fält – ”vektorer” – N element, indexeras 0..N-1

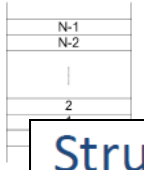
```
char  charvec[N];
sizeof (charvec) =
N bytes
```






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

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



.. with register index	[R
.. with scaled index	[R

EXEMPEL: Vi har deklarationerna:

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

Koda följande tilldelningar i assemblerspråk

```
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. 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.*

"Pre-processing"

Pre-processorn 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;
}
```

infoga en annan f

"macro" definition

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;
}
```

compilation

main.asm

```
main:
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x20,%rsp
{
        printf("Highest score is %d\n", 100);
        mov    $0x64,%edx
        lea    0x2a87(%rip),%rcx
        callq  402bb8 <printf>
        printf("Square of 3 is %d\n", 3*3 );
        mov    $0x9,%edx
        lea    0x2a8b(%rip),%
        callq  402bb8 <printf>
        return 0;
        mov    $0x0,%eax
}
        add    $0x20,%rsp
        pop    %rbp
        retq
```

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"
...
int
{
...
}

// foo2.c
#include "decl.h"

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

Assemblering, assemblerkod till maskinkod

Översättningen från assemblerkod till objektkod (maskinkod) görs av assemblatorn.
I objektkoden finns även symbolinformatio
externa funktioner och variabler

main.asm

```
main:
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x20,%rsp
{
        printf("Highest score is %d\n", 100
        mov    $0x64,%edx
        lea    0x2a87(%rip),%rcx
        callq  402bb8 <printf>
        printf("Square of 3 is %d\n", 3*3 )
        mov    $0x9,%edx
        lea    0x2a8b(%rip),%
        callq  402bb8 <printf>
        return 0;
        mov    $0x0,%eax
}
        add    $0x20,%rsp
        pop    %rbp
        retq
```

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`

main.o

```
<main>:
55
48 89 e5
48 83 ec 20
e8 c3 01 00 00
        printf("Highest score is
%d\n", 100);
ba 64 00 00 00
48 8d 0d 87 2a 00 00
48 3a 16 00 00
        printf("Square of 3 is
%d\n", 3*3 );
ba 09 00 00 00
48 8d 0d 8b 2a 00 00
e8 29 16 00 00
        return 0;
b8 00 00 00 00
}
48 83 c4 20
5d
c3
```

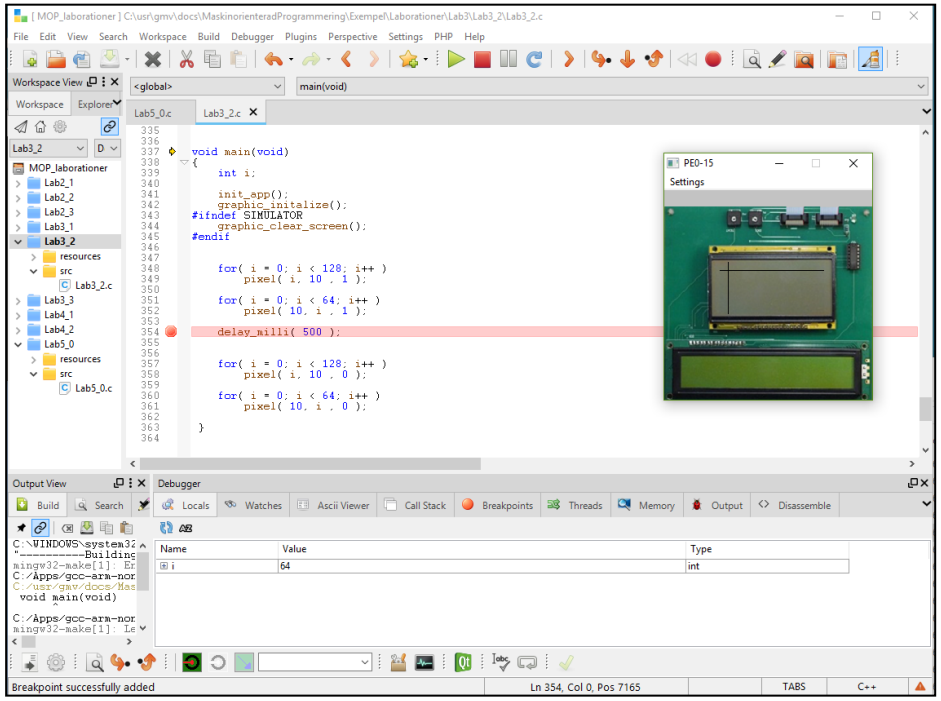
Länkas samman till:

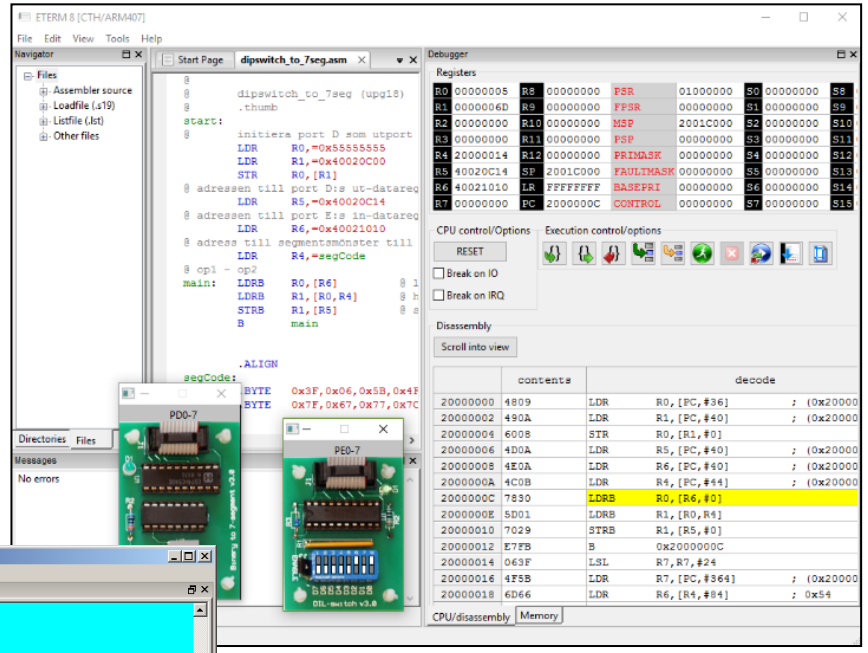
main.exe

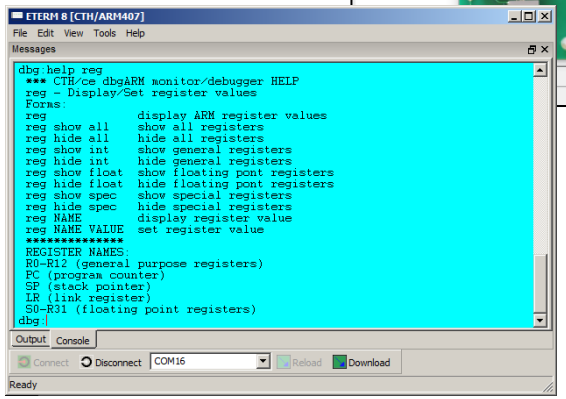
printf hämtas från programbiblotek libc:

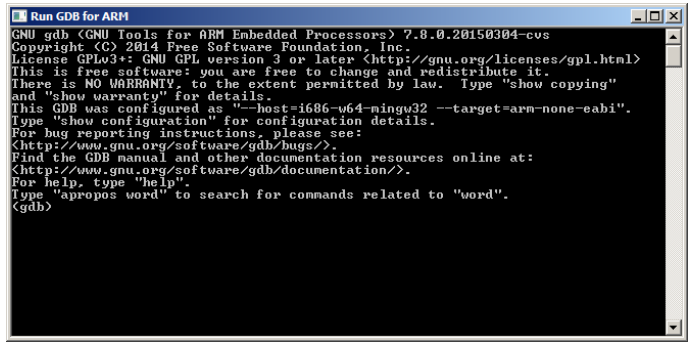
```
0000000000402bb8 <printf>:
  402bb8:       ff 25 86 57 00 00
                jmpq   *0x5786(%rip)        #
408344 <__imp_printf>
  402bbe:       90
                nop
  402bbf:       90
                nop
```

- *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.

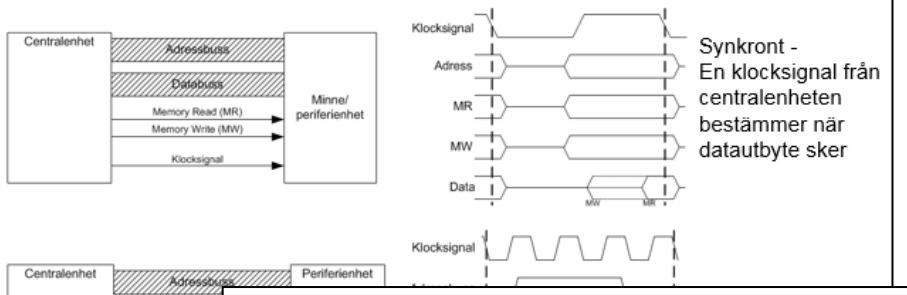
3. Systemprogrammerarens bild

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.*

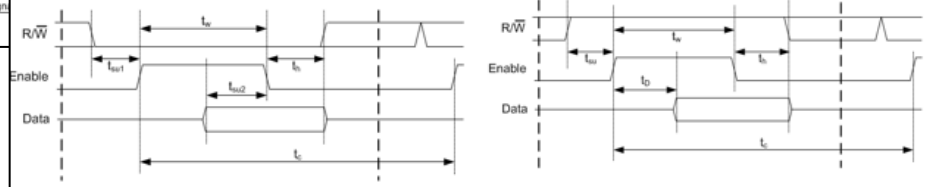
Synkrona och asynkrona gränssnitt



Tidsdiagram - underlag för synkronisering

Skrivcykel - data överförs från CPU till periferienhet

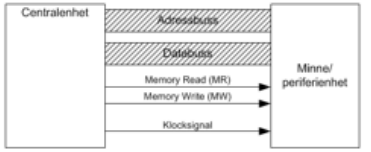
Läscykel - data överförs från periferienhet till CPU



		min
t_c	cykel tid	min
t_w	klockpuls ("Enable") varaktighet (hög och låg)	min
t_{su1}	styrsignalernas setup-tid, före positiv E-flank	min
t_{su2}	setup-tid för data, skrivning, före negativ E-flank	min
t_D	setup-tid för data, läsning, före negativ E-flank	max
t_h	hold-tid, varaktighet (efter negativ E-flank)	min

Ovillkorlig överföring

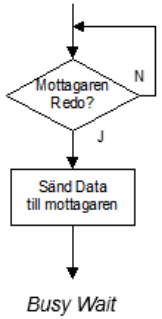
Kräver samtidighet "synkron" - en speciell signal, "Enable", används då för att ange exakt när datautbyte ska ske.

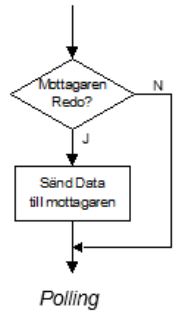


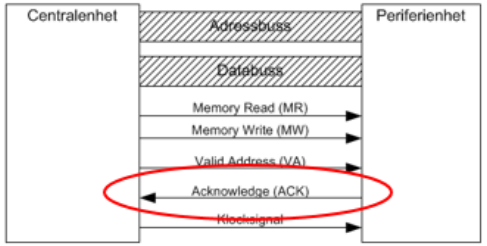
bered

Villkorlig överföring

Periferienheten har ett register med en statusbit som anger om data finns eller ej.







Förutsätter inte samtidighet men kräver speciella "handskakningssignaler". Ett sådant gränssnitt kallar vi därför "asynkront".

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

Vektortabellen

Anger position för de olika undantagsvektorerna.

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

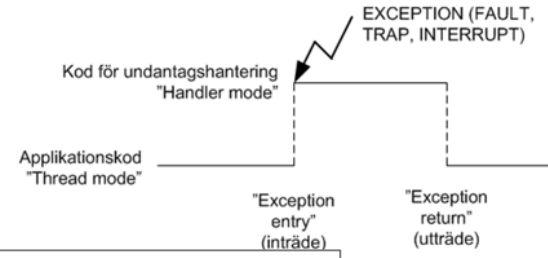
Resten av tabellen är specifik för den aktuella microcontrollern.

	.	.	.	Reserverad stack	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is inked 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
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD	PVD through EXTI line detection	0x0000 0044
2	9	settable	TAM		
3	10	settable	RTC		
4	11	settable	FLA		
5	12	settable	RCC		
6	13	settable	EXT		
7	14	settable	EXT		
8	15	settable	EXT		
9	16	settable	EXT		
10	17	settable	EXT		
...					
...					
...					
81	88		FPU		

Undantagshantering - detaljerna

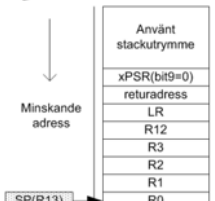
"Inträdet" hanteras automatiskt av processor.
"Utträdet" initieras och handhas av programvaran (undantagsrutinen)

Utträde:
Återställ PC från LR. Det speciella EXC_RETURN värdet som då hamnar i PC avkodas.
Registerinnehåll återställls då från stacken varvid PC får rätt återhoppsadress.

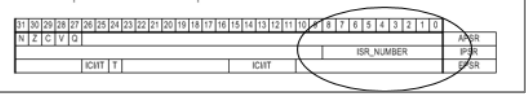


dvs.
stacken
R (EXC_RETURN)

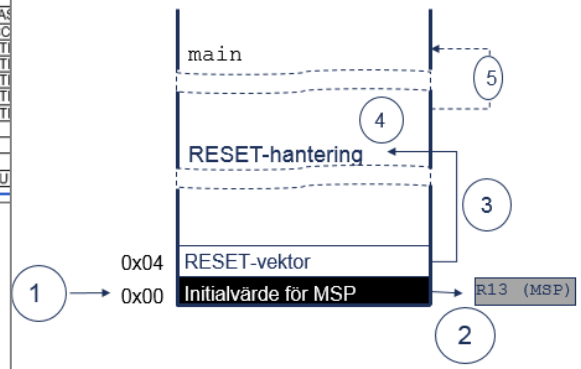
antagshantering
PSR
PC



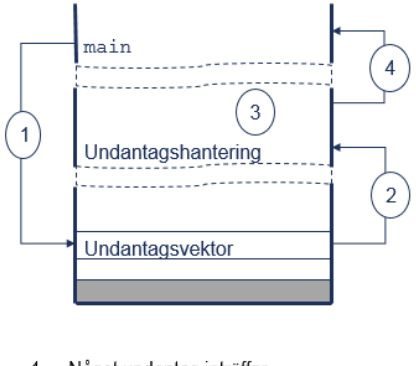
	Flyttal aktiverat	Flyttal ej aktiverat
Återtill *handler mode* – använd *main stack*	0xFFFFFFE1	0xFFFFFFF1
Återtill *thread mode* – använd *main stack*	0xFFFFFFE9	0xFFFFFFF9
Återtill *thread mode* – använd *process stack*	0xFFFFFFED	0xFFFFFFFD



Exekvering vid undantag



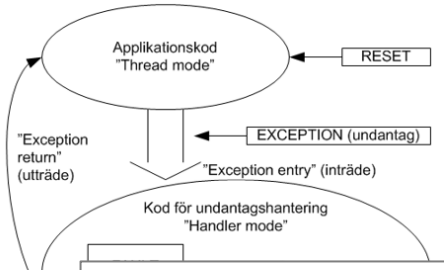
1. Återstart (Reset aktiverad)
2. Ladda MSP (Main Stack Pointer) från adress 0x00
3. Ladda RESET-vector från adress 0x04
4. RESET-hantering exekveras i "Thread mode"
5. Systemets huvudprogram startas



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

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

Undantagshantering - "exception"



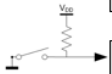
Med "undantag" menar vi här en rad olika typer av händelser:

- RESET (*återstart*):
  - *power on* (kallstart)
  - *warm reset* (varmstart
- FAULT:
  - Exekveringsfel – kan inte fortsätta

Undantagstyper

De olika undantagstyperna har en naturlig, fallande prioritet

RESET:
- power on (kallstart)
- warm reset (varmstart)



NMI:
Non Maskable Interrupt

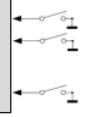
FAULT:
- Hard fault
  generellt fel
- Memory management fault
  fel användning av minne
- Bus fault
  exvis obefintligt minne
- Usage fault
  exvis "unaligned"

TRAP:
- SVC- instruktion
- BPKT-instruktion
- ...

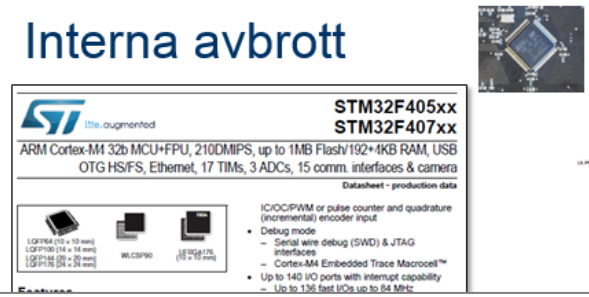
INTERRUPT:

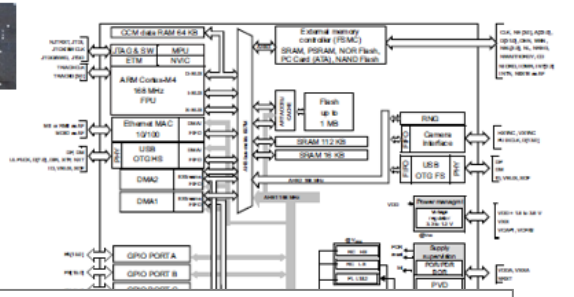
INTERNA:
- SysTick
- Watchdog
- Periferikretsar
  .....

EXTERNA:
- IO-pinnar kan konfigureras som avbrottsingångar
- EXTI (External Interrupt) modul



Interna avbrott



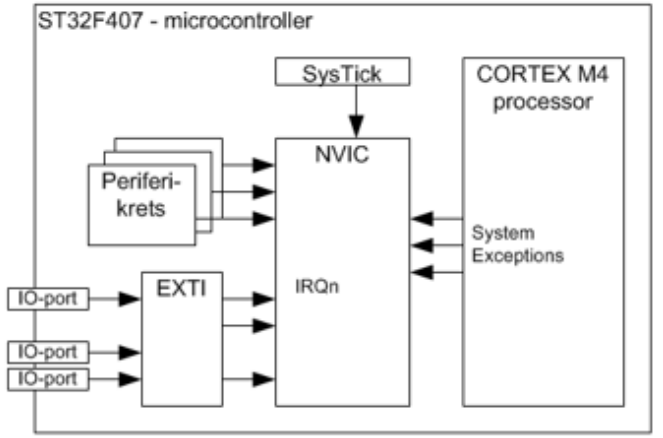


Externa avbrott

Externa avbrott är en form av undantag.

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

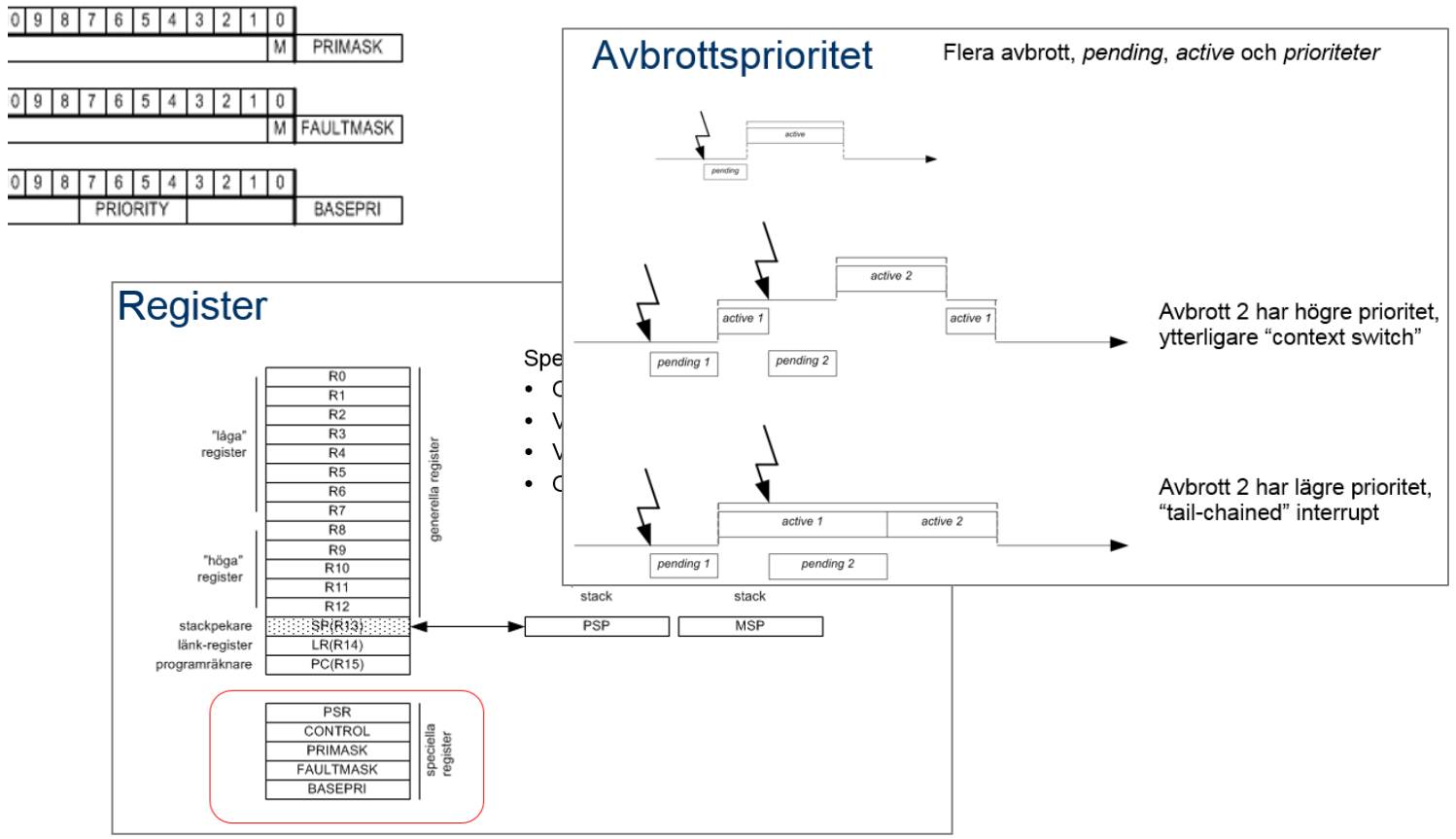
Avbrottshanterare kan skrivas uteslutande med 'C'-kod.



	6	settable	SysTick	System tick timer
0	7	settable	WWDG	Window Watchdog interrupt
1	8	settable	PVD	PVD through EXTI line detection interrupt
2	9	settable	TAMP_STAMP	Tamper and TimeStamp interrupts through the EXTI line
3	10	settable	RTC_WKUP	RTC Wakeup interrupt through the EXTI line
4	11	settable	FLASH	Flash global interrupt
5	12	settable	RCC	RCC global interrupt
6	13	settable	EXTI0	EXTI Line0 interrupt
7	14	settable	EXTI1	EXTI Line1 interrupt
8	15	settable	EXTI2	EXTI Line2 interrupt
9	16	settable	EXTI3	EXTI Line3 interrupt
10	17	settable	EXTI4	EXTI Line4 interrupt
11	18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt
12	19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt

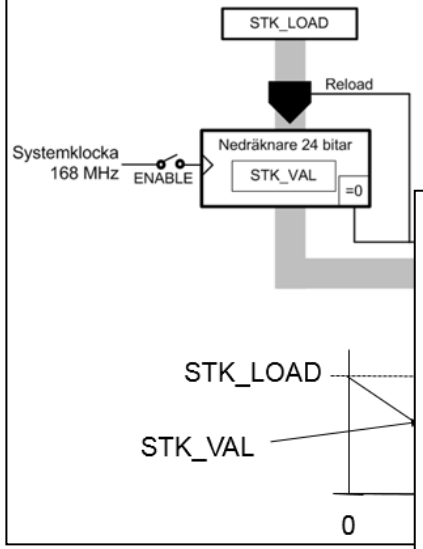
- *beskriva och tillämpa olika metoder för prioritetshantering vid multipla avbrottskällor (mjukvarubaserad och hårdvarubaserad prioritering, avbrottsmaskering, icke-maskerbara avbrott).*

	.	.	.	Reserverad stack
	-3	fixed	Reset	Reset
	-2	fixed	NMI	Non maskable interru Clock Security Syste to the NMI vector.
	-1	fixed	HardFault	All class of fault
	0	settable	MemManage	Memory managemen
	1	settable	BusFault	Pre-fetch fault, memc
	2	settable	UsageFault	Undefined instruction
	.	.	.	Reserved
	3	settable	SVCall	System service call v instruction
	4	settable	Debug Monitor	Debug Monitor
	-	-	-	Reserved
	5	settable	PendSV	Pendable request for
	6	settable	SysTick	System tick timer
0	7	settable	WWDG	Window Watchdog in
	8	settable	PVD	PVD through EXTI lin interrupt
2	9	settable	TAMP_STAMP	Tamper and TimeSta through the EXTI line
3	10	settable	RTC_WKUP	RTC Wakeup interrup EXTI line
4	11	settable	FLASH	Flash global interrupt
5	12	settable	RCC	RCC global interrupt
6	13	settable	EXTI0	EXTI Line0 interrupt
7	14	settable	EXTI1	EXTI Line1 interrupt
8	15	settable	EXTI2	EXTI Line2 interrupt
9	16	settable	EXTI3	EXTI Line3 interrupt
10	17	settable	EXTI4	EXTI Line4 interrupt
..				
..				
..				
81	88		FPU	FPU global interrupt



- *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			rw	rwrwrw	STK_CTRL
4		rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	STK_LOAD
8		rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	STK_VAL
0xC	r r	r r r r r r r r	r r r r r r r r	r r r r r r 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 8	7 6 5 4 3 2 1 0
4		rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw

STK_VAL (0xE000E018) Räknarvärde

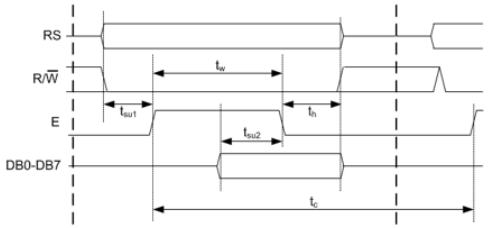
offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	Register
8		rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	rwrwrwrwrwrwrwrw	STK_VAL

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
```

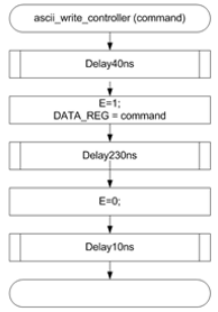
Skrivcykel - karakteristika

Databladets tidsdiagram illustrerar hur styrsignaler och data ska läggas ut



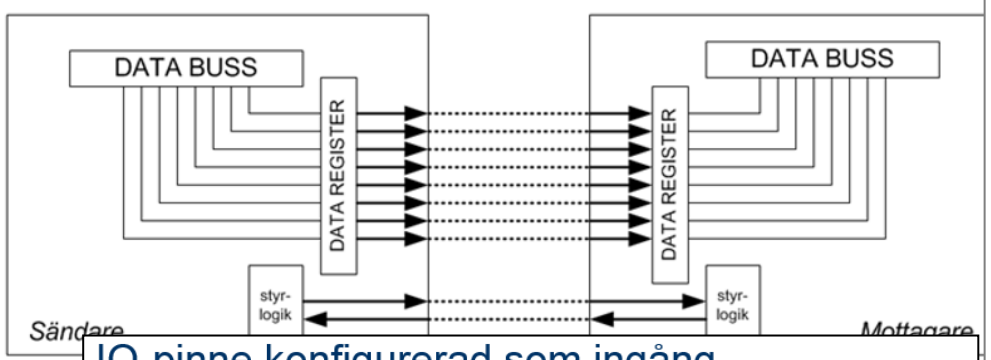
		min
t_c	cykel tid	500 ns
t_w	klockpuls ("Enable") varaktighet (hög och låg)	230 ns
t_{su1}	styrsignalernas setup-tid, före positiv E-flank	40 ns
t_{su2}	setup-tid för data, skrivning, före negativ E-flank	80 ns
t_h	hold-tid, varaktighet (efter negativ E-flank)	10 ns

Algoritm:

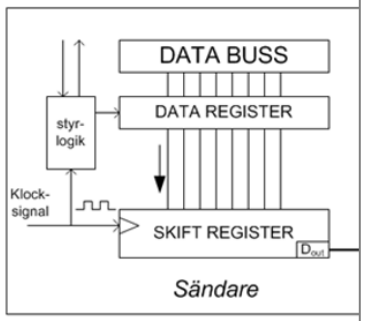


- *beskriva och använda kretsar för parallell respektive seriell överföring.*

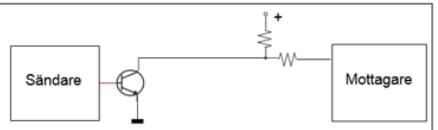
Parallell överföring



Seriell överföring

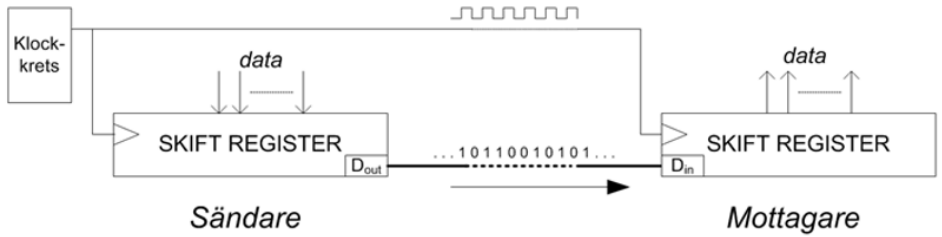


Asynkron överföring, Startbitsdetektering



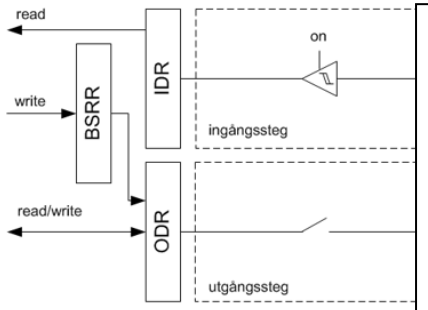
Synkron överföring

Klocksignal och data på skilda ledningar



IO-pinne konfigurerad som ingång

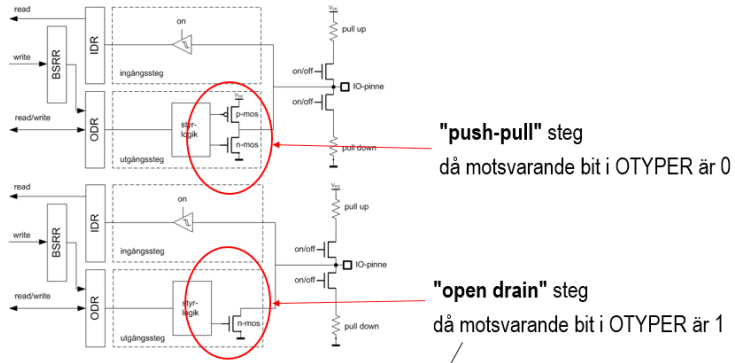
Vi kan programmera ”pull-up” eller ”pull-down”-funktion, inget externt motstånd behövs.



GPIO Pull-Up/Pull-Down Register (PUPD

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14
0x0C	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
	0x0F								0x0E									

IO-pinne konfigurerad som utgång



"push-pull" steg
då motsvarande bit i OTYPER är 0

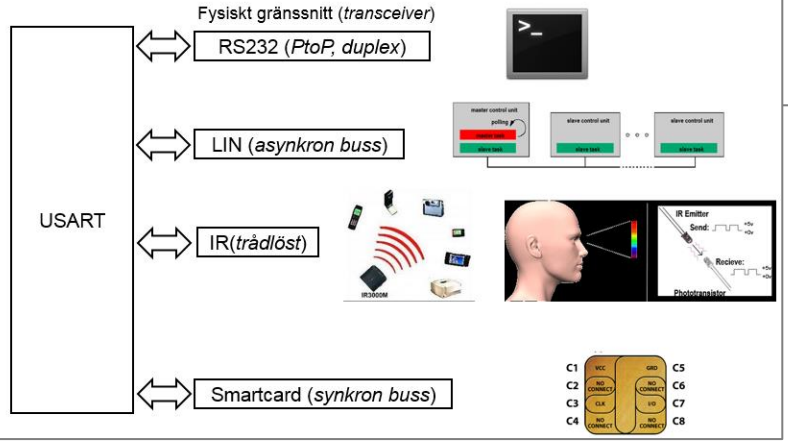
"open drain" steg
då motsvarande bit i OTYPER är 1

GPIO Output Type Register

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	mnemonic
0x04																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	GPIO_OTYPER
																	0x05								0x04							

Kräver

USART – Universal Synchronous/Asynchronous Receiver/Transmitter



Av speciell vikt: ”maskinnära programmering...”
Företrädesvis i ’C’ men också i assemblerspråk

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

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

Deklarationer

```
char    c;    /* 8-bitars
short   s;    /* 16-bitar
long    l;    /* 32-bitar
int     i;    /* 32-bitars datat...orlek word */
```

```
.ALIGN 1 @ align to half (2 byte)
.ALIGN 2 @ align to word (4 byte)
.ALIGN   @ default, align to word
```

```
char  c;

short s;

long  l;

inti;
```

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	Dessa register sparas normalt sett
R2	parameter 3 / temporärregister	inte över funktionsanrop men om,
R1	parameter 2 / resultat 2 /temporärregister	så är det den anropande (*caller*)
R0	parameter 1 / resultat 1/ temporärregister	funktionens uppgift

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 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);
```

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;
```

Assemblerprogrammering: översätta enklare C-kod (eller från en annan specifikation, text eller flödesdiagram) till assemblerspråk, med kompilatorkonventioner.

Endast 7,5 hp, EDA483, DIT153

1.7.10. Funktionen `int g( int )` är definierad. Visa hur följande funktion kan kodas i assemblerspråk:

```
int f( int val )
{
    int i;
    int bits = 0;

    for( i=0 ; i< val; i++ )
    {
        bits = bits | g(i);
    }
    return bits;
}
```

```
@ Registerallokering:
@ R0, parameter och returvärde
@ R4 "i"
@ R5 "bits"
@ R6 "val" spill från R0
f:
PUSH            {R4,R5,R6,LR}
@ 'prolog'
        MOV             R6,R0   @ spillregister R6
        MOV             R5,#0           @ bits = 0;
@ 'uttryck 1'
        MOV             R4,#0   @ i = 0;
@ 'For_continue' - 'uttryck 2'
.L1:
        CMP             R4,R6   @ i - val
        BGE             .L2             @ i < val, komplementvillkor
@ 'satser'
        MOV             R0,R4
        BL              g                       @ g(i);
        ORR             R5,R5,R0                @ bits = bits | g(i);
@ 'uttryck 3'
        ADD             R4,R4,#1                @ i++;
        B               .L1
@ 'For_Break'
.L2:
        MOV             R0,R5   @ "bits" -> R0 (returvärde)
        POP             {R4,R5,R6,PC}
```

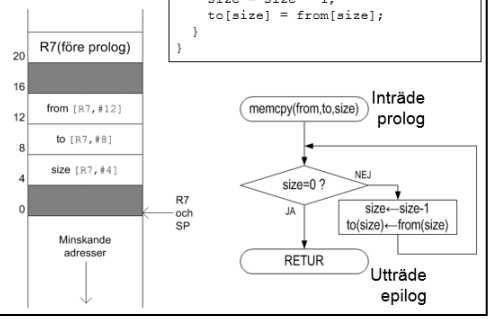
Kodoptimering – "för hand" - exempel

```
void memcpy(  unsigned char from[],
    unsigned char to[],
    unsigned int size )
{
  while (size > 0){
    size = size - 1;
    to[size] = from[size];
  }
}
```

Anropssekvens: (formellt):

```
LDR     R0,=from
LDR     R1,=to
LDR     R2,size
BL      memcpy
```

Stackens utseende i "memcpy" efter prolog



Kodförbättring, ingen onödig minnesanvändning, dvs. *registerallokering*, "förbättrad för hand"

Registerallokering:
`R0: from`, ersätter [R7,#12]
`R1: to` , ersätter [R7,#8]
`R2: size` , ersätter [R7,#4]
`R3:` temporär
`R4:` temporär

Vi gör dessa substitutioner och kommenterar därefter ut de instruktioner som då blir överflödiga...

```
memcpy:
memcpy_prolog:
        PUSH  {R7}
        SUB   SP,SP,#20
        MOV   R7,SP
        STR   R0,[R7,#12]
        STR   R1,[R7,#8]
        STR   R2,[R7,#4]
memcpy_1:
        LDR   R3,[R7,#4]
        CMP   R3,#0
        BEQ   memcpy_epilog
memcpy_2:
        LDR   R3,[R7,#4]
        SUB   R3,R3,#1
        STR   R3,[R7,#4]
        LDR   R2,[R7,#8]
        LDR   R3,[R7,#4]
        ADD   R3,R3,R2
        LDR   R1,[R7,#12]
        LDR   R2,[R7,#4]
        ADD   R2,R2,R1
        LDRB  R2,[R2]
        STRB  R2,[R3]
        B     memcpy_1
memcpy_epilog:
        ADD   SP,SP,#20
        POP   {R7}
        BX    lr
```

```
memcpy:
memcpy_prolog:
        PUSH  {R4,LR}
@       SUB   SP,SP,#20
@       MOV   R7,SP
@       STR   R0,[R7,#12]
@       STR   R1,[R7,#8]
@       STR   R2,[R7,#4]
memcpy_1:
@       LDR   R3,[R7,#4]
        CMP   R2,#0
        BEQ   memcpy_epilog
memcpy_2:
@       LDR   R3,[R7,#4]
        SUB   R3,R3,#1
@       STR   R3,[R7,#4]
@       LDR   R2,[R7,#8]
@       LDR   R3,[R7,#4]
        MOV   R3,R0
        ADD   R3,R3,R2
        MOV   R4,R1
        ADD   R4,R4,R2
@       ADD   R3,R3,R2
@       LDR   R1,[R7,#12]
@       LDR   R2,[R7,#4]
@       ADD   R2,R2,R1
        LDRB  R3,[R3]
        STRB  R3,[R4]
        B     memcpy_1
memcpy_epilog:
@       ADD   SP,SP,#20
        POP   {R4,PC}
@       BX    lr
```

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

Pekararitmetik

Följande operationer är tillåtna på pekare:

- Adressoperator
- Dereferens
- Addition av heltalskonstant
- Subtraktion av heltalskonstant
- Subtraktic

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

"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 address kan ändras "från utsidan" (dvs. en ändring kar

Pekare till **struct**, pilnotation

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

Tilldelning från fält

Exempel: Vi har deklarationerna:

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

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...

ssembler

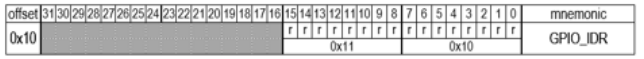
```
R0,=vec
R1,i
R0,[R0,R1]
R1,=c
R0,[R1]
```

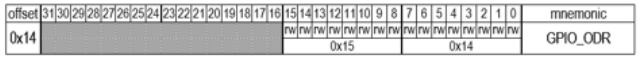
```
int find_pos_of_setmsb(unsigned int value,
                       unsigned char *position){
  *position=0;
  while(value){
    if(value==1)
      return 1;
    (*position)++;
    value = value >> 1;
  }
  return 0;
}
```

```
#define USE_ASM
void app_init ( void )
{
#ifdef USE_ASM
__asm__ volatile(" LDR R0,=0x00005555\n");
__asm__ volatile(" LDR R1,=0x40020C00\n");
__asm__ volatile(" STR R0,[R1]\n");
#else
  *GPIO_MODER = 0x00005555;
#endif
}

void main(void)
{
  char c;
  app_init();
  while(1){
    c = *GPIO_IDR_HIGH ;
    *GPIO_ODR_LOW = c;
  }
}
```

```
#define GPIO_D                  0x40020C00
#define GPIO_MODER    ((volatile unsigned int *)  (GPIO_D))
#define GPIO_OTYPER   ((volatile unsigned short *) (GPIO_D+0x4))
#define GPIO_PUPDR    ((volatile unsigned int *)  (GPIO_D+0xC))
#define GPIO_IDR_LOW  ((volatile unsigned char *) (GPIO_D+0x10))
#define GPIO_IDR_HIGH ((volatile unsigned char *) (GPIO_D+0x11))
#define GPIO_ODR_LOW  ((volatile unsigned char *) (GPIO_D+0x14))
#define GPIO_ODR_HIGH ((volatile unsigned char *) (GPIO_D+0x15))
```





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

`struct` ("post"), en sammansatt datatyp

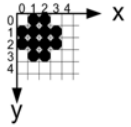
- Har en eller flera medlemmar (*fields*) av godtycklig typ, exempelvis:
  - `int`, `char`, `long` (`signed`/`unsigned`) , `float`, `double`
  - Fält och textsträngar
  - Alla typer av pekare
  - Tidigare deklarerad typalias (med "`typedef`")
  - Sammansatt typ (dvs. en annan `struct`).

Deklarationssyntax:
```
struct structnamn{
    medlem;
    [medlem; [medlem;]… ]
};
```

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

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

Inkapsling av data, ett "objekt"



Exempel: Arbetsboken, avsnitt 3.4

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

Poster med funktionspekare

Inkapsling av pekare till funktioner.

`draw`, `clear`, `move` och `set_speed` är pekartyper.

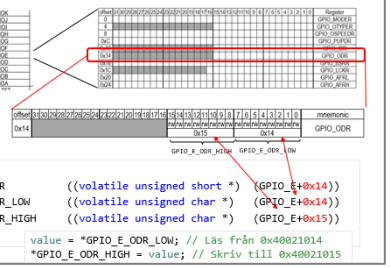
Exempel:
```
typedef struct tObj {
  PGEOMETRY geo;
  int dirx, diry;
  int posx, posy;
  void (*draw)(struct tObj *);
  void (*clear)(struct tObj *);
  void (*move)(struct tObj *);
  void (*set_speed)(struct tObj *, int, int);
} OBJECT, *POBJECT;
```

Kan uppfattas som "metoder" i objektorienterat språk.

Portadressering

Exempel:
GPIO port E

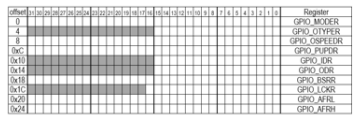
```
#define GPIO_E          0x40021000
#define GPIO_E_MODER    ((volatile unsigned int *)    GPIO_E)
#define GPIO_E_OTYPER   ((volatile unsigned short *) (GPIO_E+4))
#define GPIO_E_OSPEEDR  ((volatile unsigned int *)    (GPIO_E+8))
...
```



```
...
#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 deklarera portar



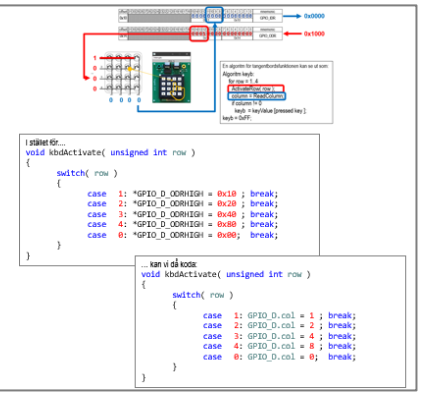
```
// Som alternativ till :
#define portModer   ((volatile unsigned int *) (GPIO_BASE))
#define portOtyper  ((volatile unsigned int *) (GPIO_BASE+0x4))
#define portOspeedr ((volatile unsigned int *) (GPIO_BASE+0x8))
osv.
// kan vi använda en post-definition som:
typedef volatile struct {
  unsigned int  moder;    // +0x0
  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  lckr;     // +0x1C
  unsigned int  afrl;     // +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_E.otyper  =  0x00000000;
GPIO_D.moder   =  0x55550000;
GPIO_D.pupdr   |= 0x00005555;
```

Adressering med bitfält

```
// GPIO
typedef volatile struct tag_gpio {
  ...
  union {
    uint32_t    idr;
    struct {
      uint8_t   idrLow;
      union{
        uint8_t idrHigh;
        uint8_t col:4;
      };
      short     reserved;
    };
  };
  union {
    uint32_t    odr;
    struct {
      uint8_t   odrLow;
      union{
        uint8_t odrHigh;
        uint8_t unused:4,row:4;
      };
      short     reserved;
    };
  ...
} GPIO;
```



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

```
... kan vi då koda:
void kbdActivate( 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:

b_7-b_6: Väljer visning:	b_5-b_3: Clock "reset":	b2-b0: Clock start/stop
00: visar '0000' 01: Clock1 10: Clock2 11: Clock3	b5=1: Clock1 RESET b4=1: Clock2 RESET b3=1: Clock3 RESET	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.

a) Definiera de symboliska adresser med lämpliga typkonverteringar som krävs för uppgiften. Deklarara också lämpliga globala variabler. (3p)

b) Visa en funktion `void portInit(void)` som initierar GPIO-portarna. (2p)

c) 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):

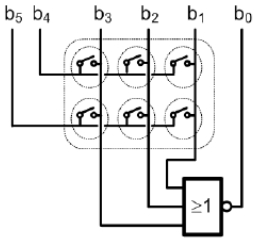
- `void systickInit(void)` som gör alla nödvändiga initieringar och
- `void systick_irq_handler(void)` som hanterar avbrotten från SYSTICK.

d) 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.

e) 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 omkopplaare 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*.

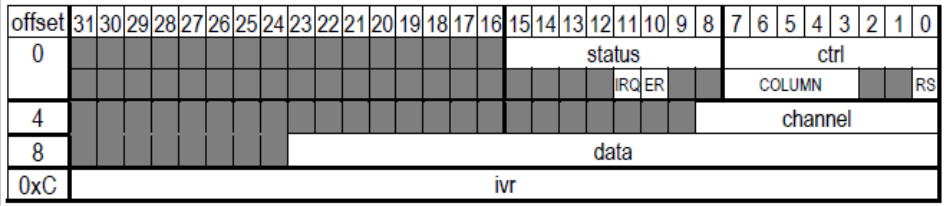
Kretsen kan också användas för att känna av en tangentnedtryckning med hjälp av avbrott. För detta kopplas kolumnerna till en NOR-grind, vars utgång är kopplad till bit 0 hos port E, denna måste ha "*pull-up*".

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'.

a) 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.

b) Visa avbrottsrutinen `void at_interrupt( void )` som kontrollerar om avbrott begärts via b0 och i så fall kvitterar avbrottet. (2p)

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



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.

a) Visa lämpliga makrodefinitioner för referens (åtkomst) av portens register status respektive data. (2p)

b) 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 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).