Maskinorienterad programmering

Oscar Palm

2022

Contents

Lecture No. 1

Kursintroduktion

1.1 Varför läser vi MOP?

Kunna utveckla för ny hårdvara
Kunna skriva snabb kod
Kunna skriva säker kod
Förkunskaper för andra kurser

Använder oss av två olika IDE:er; ETERM och CodeLite.
Var inte rädd att be om hjälp, varje vecka finns simulationspass då man kan få direkt hjälp av TA:s, annars går det alltid att fråga via canvas.

1.2 Datorn

Kallas MD407
Kommer mest använda oss av GPIO-portarna och koppla in saker som tangentbord och lcd-skärmar

1.3 Laborationsinfo

Börja med labbarna innan passen, bättre att känna till problemen innan istället för att de uppstår mot slutet av passet.
4 pass totalt á 5 labbar, den sista är lite större, vanligt att man gör ett spel.

1.4 ARM-processorn

Kommer använda en ARM Contrex M4
Använder THUMB2 instruktionsset.

Lecture No. 2

Introduktion till C

2.1 Varför lär vi oss C?

Varför lär vi oss inget modernare språk, ex
C++, Java, Javascript, C#, Python, Objective-C, Rust

2.1.1 Högre performance

Det kompilerar ned direkt till maskinkod.
Garbage collection är hemsk i vissa språk.
Klasser och dylikt leder till massa overhead man inte har kontroll över.

2.1.2 Maskinnära

Språket kräver inget operativsystem.
Kan använda sig av absoluta minnesadresser/portar till skillnad från i språk som C# och Java.

2.1.3 Säkerhet

Dåligt skydd mot farlig kod.
Specifieringen har varit någotsånär konstant sedan 70-talet, kompilatorn är så säker det blir.
Existerande implementationer är extremt stabila.

2.2 Varför C istället för assembler

2.2.1 Portabilitet

Assemblyspråk är enhetsspecifika, när en ny processor kommer krävs ny assemblykod, med C är det bara att kompilera om istället för att skriva om allt.

2.2.2 Enkelhet/Läsbarhet

C ligger närmare normalt skriftspråk än Assemblyspråken, det är lättare att skriva fungerande kod och förstå vad andra skrivit i C.

2.2.3 Performance

Moderna kompilatorer är bättre på att kompilera ned C till effektiv assemblykod än vi är på att skriva assembly.

2.3 C vs Java

C har inga klasser eller polymorfism.
C gör inget i bakgrunden åt dig, inte ens säkerhetskontroller, mer kontroll men mänskliga faktorn farligare.
Att använda felaktigt index i C är inga problem, tills det visar sig att det du skrev över var returadressen och nu exekverar datorn arbiträr kod!

Det finns ingen exception handling i C
boolean variable? more like int32!

2.4 Vad händer vid Build

har en fil main.c
Börjar med att filen preprocessing, kontrollerar texten, tar bort onödigt bös och sånt.
main.c →main.i

Filen kompileras ned till assemblykod
main.i→main.asm

Filen assembleras ned till objektkod
main.asm→main.o

Objektkoden länkas ihop med bibliotek och lite annat och blir en exekverbar fil
main.o→main.exe/main.elf

2.4.1 Pre-processing

Pre-processorn utför textsubstitution

- Alla inkluderade filer kopieras texten från och #include byts ut mot texten i Filen
- Alla makron exekveras

2.4.2 Kompilering

Vid översättning konverteras koden till den relevanta assemblerkoden (ARM/X86/IA64)

2.4.3 Assemblering

Koden kompileras ned till objektkod, det är inte riktigt samma sak som maskinkod.
I maskinkod ligger exempelvis samtliga filer i samma fil, medan de i objektkod ligger i separata filer.
Den historiska anledningen är minnesbrist och den moderna är att det underlättar att multitaska.

2.4.4 Länkning

De olika objektfilerna kombineras till en exekverbar fil.
Symboler översätts till relativa adresser och funktioner som anropats men som inte finns i den relevanta objektfilen letas upp och kopplas ihop med anropet.

2.5 C

2.5.1 Variabler, deklarationer och tilldelningar

Finns Lokala och globala variabler.

2.5.2 Heltalstyper i C

- char, minst 1 byte
- short, minst 2 byte
- unsigned short, minst 2 byte, bara positiv
- long, minst 4 byte
- unsigned long, minst 4 byte, bara positiv
- int, minst 2 byte
- unsigned, minst 2 byte, bara positiv

Tycker man det är jobbigt att skriva ex. unsigned long kan man inkludera stdint.h, skrivs exemplet istället uint32/uint64

2.5.3 Type casting

Inga problem att konvertera mellan typerna,
kan göras implicit, eller explicit, ex i=(int) s;
Teckenutvidgning och trunkering kan genomföras vid konvertering.

2.5.4 Hur stor är en typ?

Kan använda sizeof(typ) får du ut antalet bytes.

2.5.5 Funktioner, parametrar och returvärden

returvärde funktionnamn(parametrar)
statements;
return;

Allt i C skickas by-value

2.6 Programstruktur

Finns två typer av filer, c-filer och header-filer.
C-filerna genererar till slut maskinkod och headerfilerna kan appendas till c-filer.

Standard att ha en funktion int main().
Man KAN inkludera c-filer i c-filer men då blir Erik arg, vill man istället ha kod från en annan fil, skapa en headerfil med funktionsprototyper (endast signaturen).

2.7 Att styra villkorlig kompilering

define,undef,if,elif,else,endif,ifdef,ifndet (# framför alla)
Multiple definitions är dåligt, inkludera include guards i din kod.

2.8 Textsträngar

En textsträng är bara en array av chars.
printf(”Hej mitt namn är %s och jag är %d år!\n”,”saxen”,20);

2.9 Åtkomlighet och synlighet

Fungerar ungefär som i java.
static används för att gömma en global variabel i andra filer.

Lecture No. 3

Korsutveckling

3.1 Korsutveckling i C och Assembler

Vi programmerar antingen i C eller assembler.
Det går att kombinera, skriva vissa filer i Assembly och några i C.

När man skapar ett projekt i codelite, välj:

- Category: User templates
- Type: md407-startup
- Compiler: Cross GCC (arm-none-eabi)
- Debugger: GNU gdb debugger
- Build System: CodeLite Makefile Generator

Koden i startup.c börjar med
__attribue((naked))__, innebär att koden inte ska omformateras utan exekveras som den står?
section(".start_section") - remember this.

__asm__ volatile("") anänds för att exekvera assemblykod i dit c-program.

Programet börjar med att köra void startup(void)

Minnets uppbyggnad:
.start_section: början på startup.c
.text: koden
.data: Alla variabler med deklarerade värden
.rodata: Erik osäker på användning
.bss: better save space? Var ej initierade variabler (lokala osv) kan hamna, allokerat tomt utrymme.

3.2 Adressrum

Adressrummet begränsas av adressbussens storlek.
Adressrymden är 2^{32}bytes=4GB men det faktiska adressrummet är mindre.

Hur stor är databussen på en modern maskin?
Normalt är databussen 64 bytes, detta läggs i cachen och man hämtar sen den relevanta från cachen.

3.3 alignment

Om man lägger en instruktion mellan två ord kan det leda till onödigt många read/write instruktioner.
När man skriver i assembler finns därför .ALIGN, vilken vi kan använda för att aligna datan korrekt.

3.4 Register

Har 8 general purpose-register, kan spara data och sånt.
Har Några till som inte alltid kan användas, eftersom för att förhålla sig till 16-bitarsinstruktioner finns det inte alltid plats för dessa =/
En stackpekare, eller två
Ett länkregister för returadresser.
En programräknare, för att hålla koll på vad som ska exekveras typ

3.4.1 Adresseringssätt

Namn	Syntax	Exempel	RTN
Register direct	`Rx`	`MOV R0,R1`	`R0←R1`
Direct	`Symbol`	`LDR R0,symbol`	`R0←M(symbol)`
Immediate	`#const`	`MOV R0,#0x15`	`R0←0x15`
Register indirect	`[Rx]`	`LDR R0,[R1]`	`R0←M(R1)`
.. with offset	`[Rx,#offset]`	`LDR R0,[R1,#4]`	`R0←M(R1+4)`
.. with register offset	`[Rx,Ri]`	`LDR R0,[R1,R2]`	`R0←M(R1+R2)`

Ibland ligger var för långt bort, går inte att köra

```
LDR r0,var
```

skriv istället

```
LDR r0,=var
LDR r0,r0
```

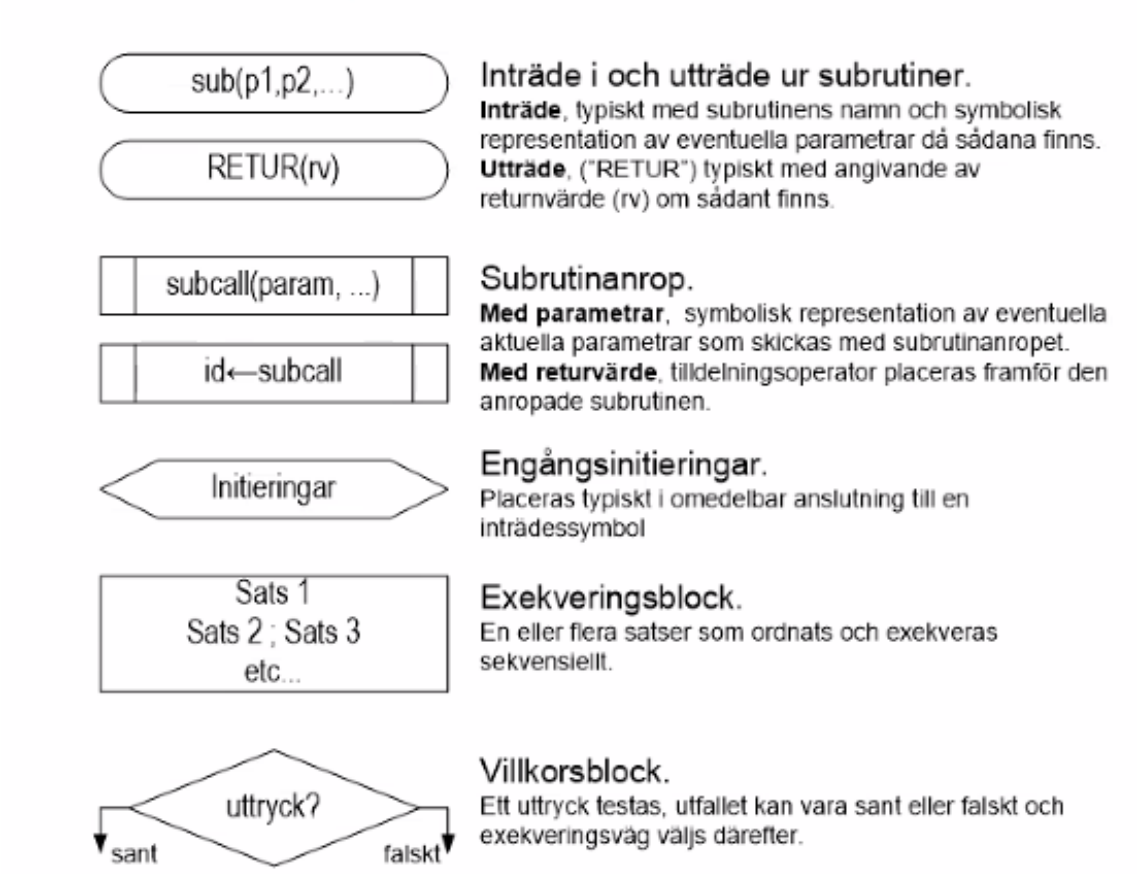
Lecture No. 4

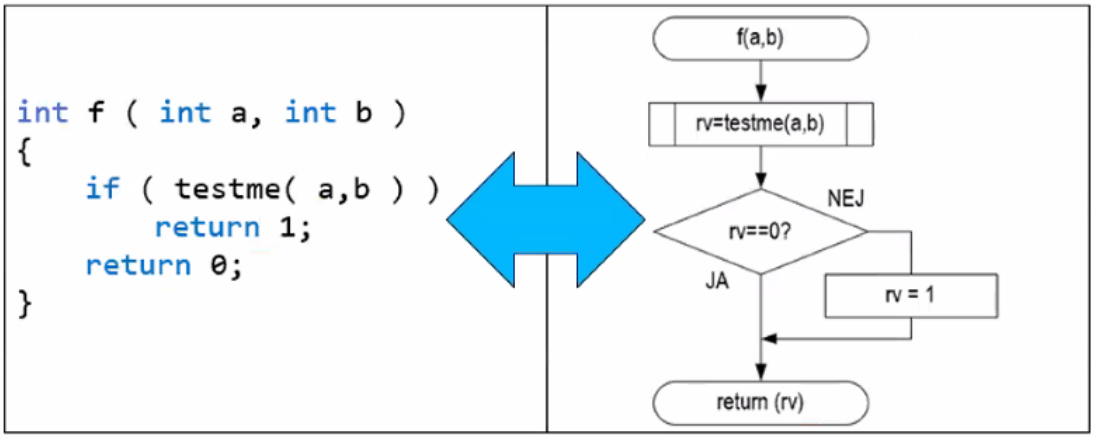
Korsutveckling, del 2

4.1 Vad är korsutveckling?

Att vi skriver vår kod på en annan typ av maskin än den vi skriver åt, exempelvis använda en amd64-dator för att skriva kod till arm.

4.2 Flödesdiagram för programstrukturer





4.3 Registeranvändning

4.3.1 R0-R3

kallas temporärregister
Stor risk att de skrivs över vid funktionsanrop.
Spara innan funktionsanrop om de behövs senare.

4.3.2 R4-R7

Dessa ska sparas undan av det anropade kodstycket om de ska användas.
Avsedda för variabler.

4.3.3 R8-R11

Dessa ska också sparas undan av det anropade kodstycket.

4.3.4 Funktionsparametrar

R0-R3 används som parametrar till funktioner.
R0 används för returvärde upp till 32-bitars.
Vid ett 64-bitars returvärde hamnar det både i R0 och R1.

4.3.5 Registerspill

Problem

Registret vi behöver är upptaget.

Lösning

Spara undan registrets innehåll på stacken eller i ett annat register.

4.4 Instruktioner för villkorlig programflödeskontroll

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

4.4.1 Funktionsparametrar 8 eller 16 bitar

Vi sparar de fortfarande i registren som 32-bitars-värden
Detta innebär att short samt signed char ska typkonverteras innan anrop.

4.4.2 Temporära/tillfälliga/lokala variabler

Om det går, håll dina temporära variabler i registren R4 och uppåt.
Om vi anropar en annan metod senare i metoden måste vi spara undan datan någonstans.
Eftersom saker redan kan vara sparade i dessa register pushar vi värdena på stacken innan vi börjar använda oss av dem, och innan return poppar vi ut dem igen.

4.4.3 När registren inte räcker till

Vi måste i sådana fall lagra saker på andra ställen, förslagsvis stacken.

Lecture No. 5

Fält, Pekare och portar

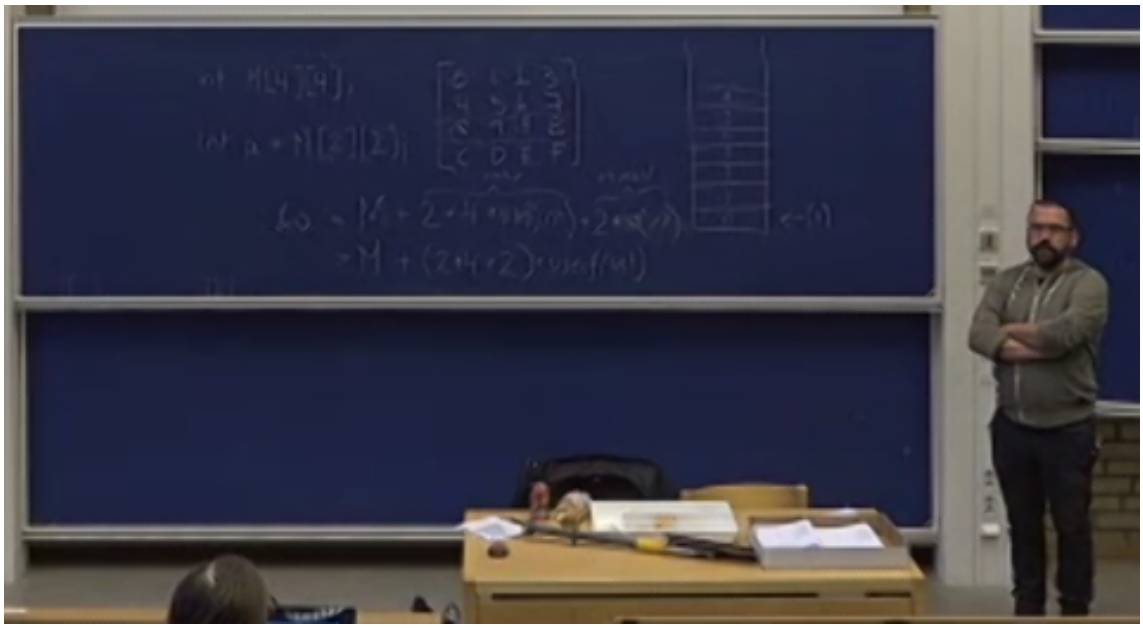
Kan hämta någonting på studentexpeditionen

5.1 Vad är ett ”Fält”?

En datastruktur som mångfaldigar förekomsten av element med samma typ.
En array helt enkelt!

En textsträng er ett fält av char som avslutas med ’\0’
Använd strlen för att räkna ut längden av en sträng, inklusive terminatorn.

5.2 Pekare



5.2.1 What is pekare?

En pekare är en datatyp för en minnesadress. En pekarvariabel innehåller minnesadressen till en variabel, port, etc. snarare än variabelns/portens värde.

En pekare är en adress helt enkelt.

5.2.2 Pekaroperatorer

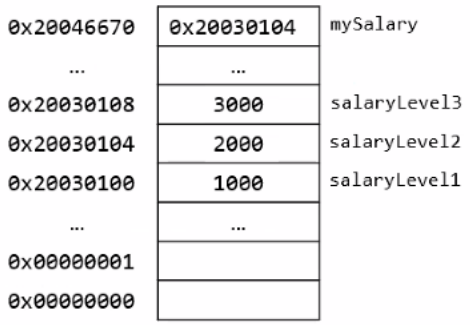
1. Pekarens värde är en adress (&)
2. Pekarens typ anger hur vi ska tolka bitarna hos innehållet på adressen
3. '*' används för att referera innehållet på adressen (dereferera).

Exempel:

```
int salaryLevel1 = 1000;
int salaryLevel2 = 2000;
int salaryLevel3 = 3000;

int* mySalary = &salaryLevel2;
```

```
&mySalary är 0x20046670
 mySalary är 0x20030104
*mySalary är 2000
```



5.2.3 Grundläggande pekartyper

En pekare är alltid en adress, i vårt fall en 32-bitars adress, men den kan peka på vilken minnestyp som helst.
En charpekare pekar exempelvis på en enbytes-minnesrymd medan en intpekare pekar på 4 bytes.

5.2.4 Dereferens

Vid dereferering av pekare får vi objektet på pekarens adress, antal bytes och hur vi tolkar bitarna beror på pekarens typ, om vi exempelvis sparat värdet 0xFF kan vi tolka det som antingen 255 eller -1 beroende på om pekaren är signed char* eller unsigned char*.

5.2.5 Varför behöver vi pekare?

Pekare låter oss referera till en variabel utan att först skapa en kopia.

5.2.6 Vad betyder *?

I en deklarering anger stjärnan en pekartyp.
I ett uttryck anger stjärnan en dereferens.

5.3 Vad är en port?

En port är en minnesadress vi använder för att skicka eller ta emot data från olika interna och externa komponenter i/till datorn, exempelvis grafikkort, datorskärm, tangentbord.
För att kunna läsa eller skriva till porten måste vi kunna dereferera en konstant.

Lecture No. 6

Digital IO

6.1 Parallell in- och utmatning

6.1.1 Ideala grindar - idealiserade signaler

Verkliga signaler slår inte om direkt, det tar tid, därför sätter vi ett tröskelvärde. Detta kan dock bli ett problem om en ”brusig” insignal svajar runt tröskelvärdet.
I verkliga kretsar implementerar man en s.k. Schmitt-trigger, där tröskelvärdet för hög och låg är separerade.

6.1.2 Anslutningar

Portarna vi framförallt kommer använda oss av är GPIO(General Purpose Input Output) Port D och E.

6.2 GPIO-port, programmerarens bild

6.2.1 Mode register

Används för att bestämma vilka pins som ska behandlas som I och vilka ska behandlas som O.

6.2.2 Spänning

Om vi stänger av flödet från en spänningskälla vid 0 kan vi inte vara säkra på att spänningen är 0, den är flytande, därför använder vi s.k pull-up/push-down, med en konstant låg spänning och ibland hög.

6.2.3 Output speed

Bestämmer hur ofta registrets innehåll överförs till utgångssteget, vid lägre frekvens, lägre strömförbrukning.
,

Lecture No. 7

Pekare

7.1 Absolut adressering

Vid deklarering av pekare används följande syntax;

```
//Example of pointer declaration and use
int main(){
    int * pointyMcPointface=0x7f000001; // deklarering av pekare samt angivande av pekad adress.
    if(* pointyMcPointface){
        *(++pointyMcPointface)=0;
    }
    return 0;
}
```

7.2 Farliga kompilatorer

om vi exempelvis har ett kodblock som väntar på input från tangentbordet, dvs vi väntar på att ett minnesvärde ska ändras av en extern källa, finns risk att kompilatorn tolkar att läsa minnet på nytt varje loop som onödigt och därför istället enbart läser en gång, för att undvika detta beteende deklareras pekaren som volatile.

7.3 Pekararitmetik

Tillåtna operationer:

- Adressoperator
- Dererefering
- Addition av heltalskonstanter
- Subtraktion av heltalskonstanter
- Subtraktion av pekare med samma typ

Det är med andra ord ej tillåtet att exempelvis bitshifta eller multiplicera med en pekare.
Addition involverande pekare fungerar inte som normal addition, istället multipliceras det adderade talet med antalet bytes per element av pekartypen, dvs. är det en int32-pekare läggs vid varje inkrementering till 4 till pekaren medan inkluderande en int16/short-pekare hade 2 lagts till per inkrementering.

7.4 Fält och pekare

Fält kan ses som ett specialfall av pekare.
Användandet av ett fält begränsar hur vi får lov att använda pekaren, exempelvis får vi inte utföra aritmetik hursom med dem, s+1 är tillåtet men inte s=s+1.

7.5 Funktionspekare

Pekare går inte enbart att använda till klassiska variabler utan till vadsom i din kod, inklusive funktioner.
Samtliga funktioner ligger sparade i minnet, vilket också betyder att programmet anropar dem mha minnesadresser, dessa går också att spara över i egna funktionspekare, deklareras void (* pekarNamn)(parameterTyp);

Lecture No. 8

Synkronisering

8.1 LCD-skärm

Skärmen har två olika register, ett dataregister samt ett styrregister.
I dataregistret skickar vi in vad vi vill visa på skärmen och till styrregistret finns flaggor för att kontrollera utskriften.

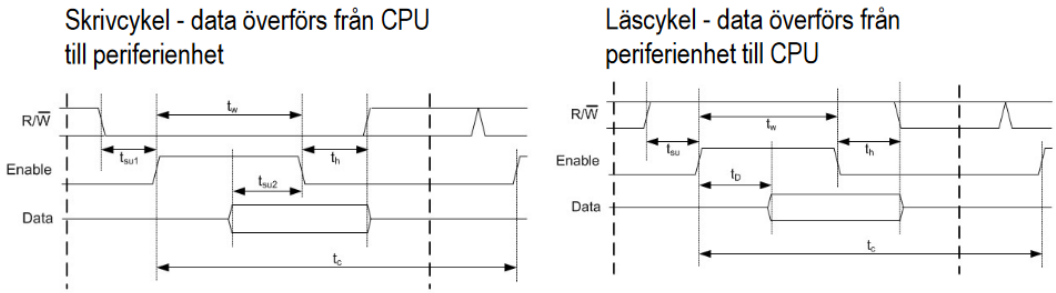
8.1.1 Styrregister

- b0, RS: är datan i dataregistret en bokstav eller ett kommando
- b1, R/W Vill vi läsa från skärmen eller skriva till
- b2 SELECT, ska alltid vara 1
- b6 ENABLE, används för att synkronisera

Enable

Vi sätter enable till 1 när vi vill starta en cykel, och sedan finns det bestämda följdprocedurer vi ska genomföra.

8.1.2 Tidsdiagram



		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

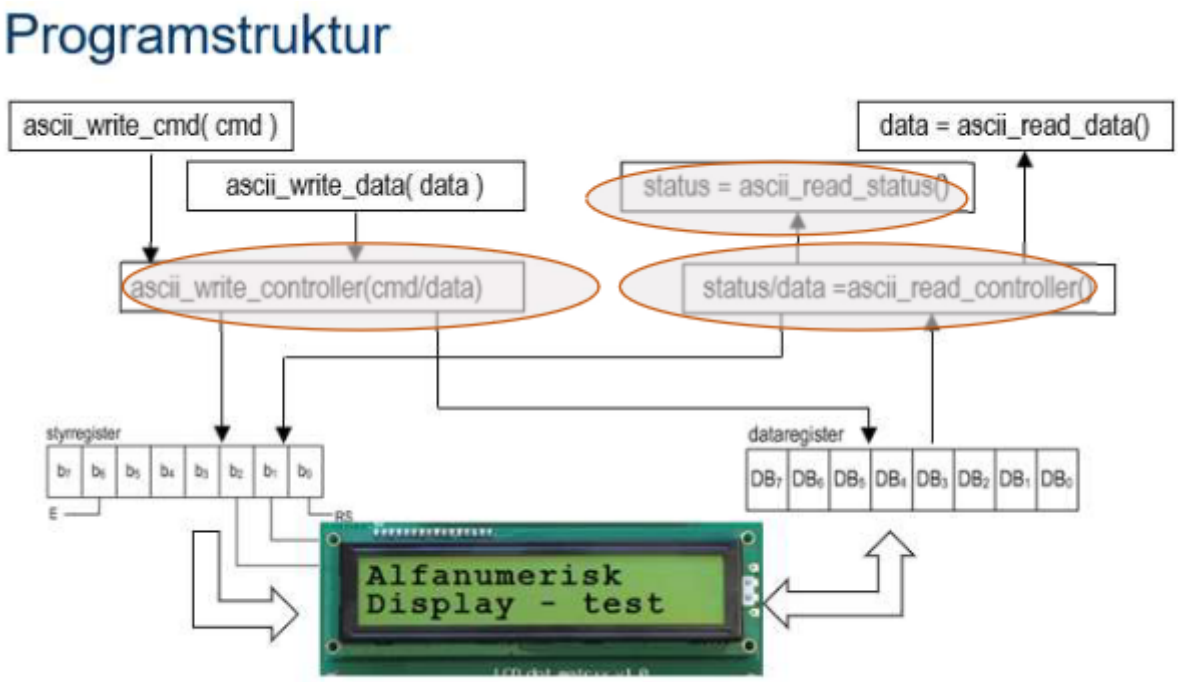
8.2 Sys Tick

Används för att räkna, mata in ett värde och den räknar ned från värdet till 0, görs separat från all annan exekvering.

8.3 Skapa en funktion för att vänta 250ns

```
void delay_250(void){
    * STK_CTRL=0;
    * STK_LOAD=168/4;
    * STK_VAL=0;
    * STK_CTRL=5;
     while(!(* COUNTFLAG&0x10000));
     * STK_CTRL=0;
     return;
}
void delay_mikro(unsigned int us){
    #ifdef SIMULATOR
        us=us/1000;
        us++;
    #endif
    while(us-->0){
        delay_250();
        delay_250();
        delay_250();
        delay_250();
    }
}
```

8.4 Labbinfo




```
#define B_E    0x40   //Enable signal
#define B_SELECT  4   //Choose display
#define B_RW      2   //0=write,1=read
#define B_RS      1   //0=control,1=Data
void ascii_ctrl_bit_set(char x){
    char c;
    c=*GPIO_E_ODRLOW;
    *GPIO_E_ODRLOW=B_SELECT|x|c;
}
void ascii_ctrl_bit_clear(char x){
    char c;
    c=*GPIO_E_ODRLOW;
    c=c&~x;
    *GPIO_E_ODRLOW=B_SELECT | c;
}
char ascii_read_status(void){
    char c;
    *GPIO_E_MODER=0x00005555;
    ascii_ctrl_bit_set(B_RW);
    ascii_ctrl_bit_clear(B_RS);
    c=ascii_read_controller();
    *GPIO_E_MODER=0x55555555;
    return c;
}
void ascii_write_controller(char c){
    ascii_ctrl_bit_set(B_E);
    *GPIO_E_ODRHIGH=c;
    delay250();
    ascii_ctrl_bit_clear(B_E);
}
```

Lecture No. 9

Sammansatta datatyper

9.1 typedef - alias för en typ

typedef används för att skapa ett alias, oftast för att förenkla och förkorta typuttryck med målet att förtydliga kod och öka läsbarheten.
konstrueras:

```
typedef typ alias_typnamn [,alias_typnamn]... ;
```

9.2 struct, en sammansatt datatyp

Har en eller flera medlemar

```
struct structnamn{
   medlem;
   annan_medlem;
};
struct structnamn variabel;
structnamn variabel; //Ogiltig
```

vill man skippa skriva struct vid variabeldeklaration får man skriva

```
typedef struct structnamn{
   medlem;
   annan_medlem;
} STRUCTNAMN;
STRUCTNAMN variabel;
struct structnamn variabel2;
```

men det är onödigt med flera namn, det kan vara praktiskt att göra den namnlös, såsom:

```
typedef struct{
   medlem;
   annan_medlem;
} STRUCTNAMN;
STRUCTNAMN variabel;
struct structnamn variabel2;  //Ogiltig
```

9.2.1 Användning

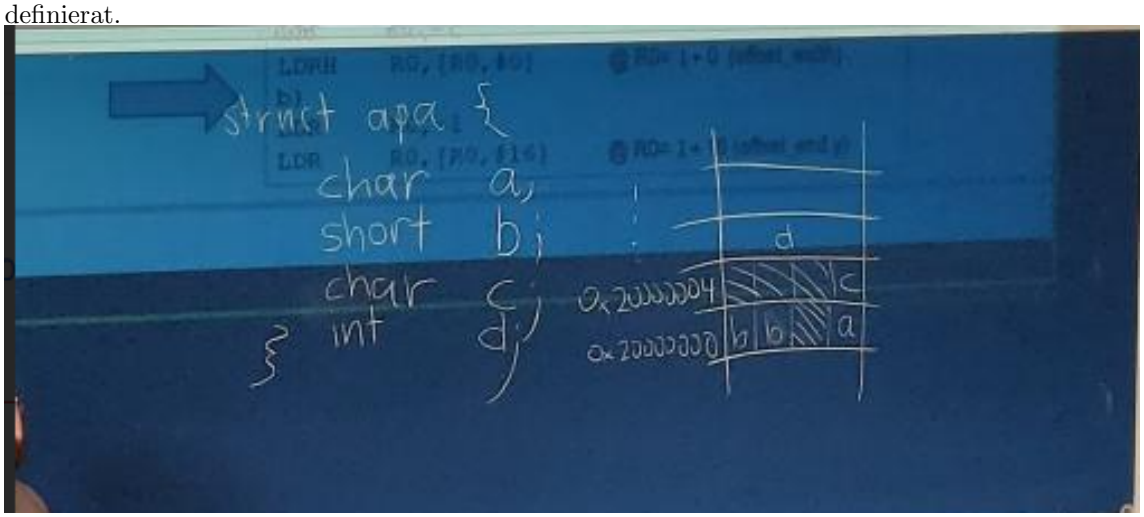
```
   #include <stdio.h>
typedef struct coord{ int x,y; } COORD;
int main()
{
struct coord start;
COORD end;
start.x = 10; start.y = 10;
end.x = 20; end.y = 20;
printf("Line starts at (x,y) %d,%d \n
and ends at (x,y) %d,%d \n",
start.x, start.x,
end.x, end.x);
return 0;
}
```

9.3 Referenser

```
   typedef struct{
      int,x,y;
   } COORD;

   //Vi har deklarationerna
   COORD start, end;
   //Koda tilldelningen:
   start.y=end.y;
   //I ARM/THUMB
   /*
   LDR R0,=end
   LDR R0, [R0,#4]
   LDR R1,=start
   STR R0,[R1,#4]
   */
```

När vi använder oss av structen måste allt i den vara minnesalignat och kommer ligga i samma ordning som vi definierat.



9.4 Initiering

9.4.1 Fullständig

```
struct Course{
   char * name;
   float credits;
   int numOfParticipans;
};
struct Course c1={"MOP",7.5,110};
```

9.4.2 Ofullständig

```
struct Course{
   char * name;
   float credits;
   int numOfParticipans;
};
struct Course c1={"MOP",7.5}; //Lamnar ute definitionen av de kvarvarande variablerna
struct Course c2={.credits=7.5,.numOfParticipants=110};
//Ar inte bara de sista variablerna man kan utelamna, kan specifiera vilka man vill ge varden
```

9.5 Ofullständig deklaration

Så länge variabeln är en pekare till typen istället för en direkt variabel kan man använda namn på ej deklarerade typer.

9.6 Pekare till struct, pilnotation

```
//hard to write and understand
(*ptr).x=value;
//Cleaner to write
ptr->x=value;
```

9.7 Portadressering med poster

struct är väldigt användbar för att deklarera portar.

9.8 Unions, ”sneak peek”

Kan användas för att lägga två variabler i samma minnesrymd.

Lecture No. 10

Undantagshantering och interna avbrott

10.1 Varför behövs exceptions?

- Tillåta flera processer att köra simultant
- Förhindra program från att köras på felaktiga eller osäkra sätt
- Hantera olika hårdvaruenheter

10.2 Tre grupper av avbrottskällor

10.2.1 Interna avbrott (System exceptions)

Exempelvis systick eller att användaren försöker läsa andra processers arbetsminne.

10.2.2 Interna avbrott från periferikretsar

Avbrott från enheter på datorn utanstående processorn.

10.2.3 Externa avbrott från IO-enheter

Avbrott från olika periferienheter, såsom mus eller tangentbord.

10.3 Undantagshantering

Hur processen ska gå till väga när en exception uppstår.

10.3.1 Olika typer av exceptions

- RESET

  power on/warm reset
- FAULT

  Exekveringsfel, division by zero, skriva till ogiltig minnesadress
- TRAP

  Debuggingexception, avbrott skapat av maskininstruktion.
- INTERRUPT

  Hårdvarusignalerat avbrott, keyPress osv.

10.3.2 Detaljer

När en exception uppstår sker en Exception entry, där processorn byter från att exekvera i thread mode till att exekvera i handler mode.
När exceptionen senare har hanterats sker en Exception return då processens privilegier deeskaleras tillbaks till thread mode.

Innan processen börjar exekvera exceptionkoden? sparas en del saker undan på stacken;

xPSR

returadressen

LR

R12

R13

R3

R2

R1

R0

10.4 Relokering av vektortabellen

När vi vill lägga till interruptfunktioner måste vi ”flytta” vektortabellen.

10.5 Speciella register

APSPR, IPSR, EPSR, CONTROL, PRMASK, FAULTMASK, BASEPRI

Dessa kan användas för bl. a.

- Aktivering av FPU
- Välja stack
- ändra exekveringstillstånd
- Betjäning av avbrott

Lecture No. 11

perifera och externa avbrott

Läsanvisningar

Kap 6.5-6.∞

11.1 NVIC – kontrollerar samtliga avbrott

Innehåller fem uppsättningar av register för olika interruptfunktioner. Varje registeruppsättning har 82 bitar, en för varje interrupt.

Interrupt Set Enable Registers; NVIC_ISERx
Interrupt Clear Enable Registers; NVIC_ICERx
Interrupt Set Pending Registers; NVIC_ICPRx
Interrupt Active Bit Registers; NVIC_IABRx

11.1.1 Bestäm register x, bitnummer bit i NVIC

x är interruptindex delat med 32, dvs 0,1,2.

11.2 Externa avbrott från periferikretsar

En portpinne kan konfigureras som avbrottsingång via EXTI-modulen (External interrupt/event controller). Detta möjliggör avbrott från enheter utanför enchipsdatorn, externa avbrott.

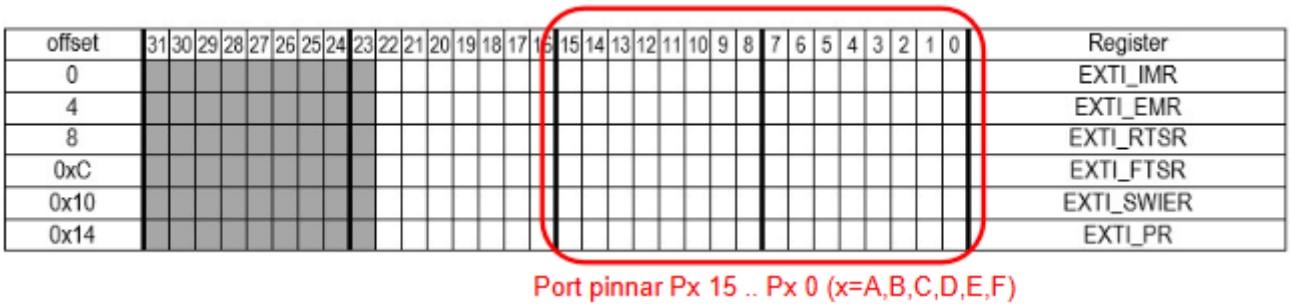
11.3 SYSCFG, Sysconfig

SYSCFG_EXTICRx är fyra kontrollregister med identisk struktur.

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
8	EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]				SYSCFG_EXTICR1
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	

11.3.1 EXTI-modul

Det finns en del kvarstående EXTI-register; IMR,EMR,RTSR,FTSR,SWIER och PR.



PR - pending request

IMR - interrupt mask register

SWIER - software interrupt event register

RTSR - rising trigger selection register

FTSR - falling trigger selection register

EMR - event mask register

Lecture No. 12

Typer och lagringsklasser

Dags att börja tentaplugga!

Övning

Konvertera följande kod till assembly:

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

12.1 Tillfällig lagring - i registren eller på stacken

Lokala variabler.

12.2 Permanent lagring

Vi allokerar plats i minnet redan innan programmet börjar exekvera. Saker som globala variabler.

12.2.1 Keyword static i C

Har två betydelser, beroende på om den statiska är deklarerad som global eller lokal.
Global:
Används för att ha samma variabel över flera filer.
Lokal:
Används för att visa kompilatorn att variabeln ska allokeras minne vid startup, innebär att variabeln behåller värde mellan flera funktionsanrop.

12.3 Synlighet

En funktion blir synlig först efter raden den är deklarerad på, kan lösas genom att börja programmet med s.k. prototyper.

12.4 Varaktighet

- automatic
- static
- allocated
- thread

12.5 linkage

12.5.1 ingen bindning

Lokala variabler, syns enbart i sitt deklarerade scope.

12.5.2 extern bindning

Som standard samtliga globala variabler. Kan ses från samtliga c-filer.

12.5.3 intern bindning

När en global variabel deklareras som *static*

12.6 lagringsklasser

- auto
  tillfällig bindning
- register
  visar att passar bra i register
- static
  skapar permanent intern bindning
- extern
  skapar permanent extern bindning

12.7 union

```
//Same syntax as structs
typedef union{
   float v[2];
   struct { float x,y; };
}Vec2f,*PVec2f;
```

12.8 Endianness

Vilken ordning bytes skickas i.

12.8.1 Big vs little endian

Big endian

Med big endian skickas mest signifikanta byten först.

Little endian

Med little endian skickas mest signifikanta byten sist.

12.8.2 Kod för att testa endianness

```c
#include <stdio.h>
union {
   int a;
   char c[4];
} x;
int main() {
   x.a = 0x11223344;
   printf("%d: %d %d %d %d\n",
   x.a, x.c[0], x.c[1], x.c[2], x.c[3]);
}
```

12.9 Enums

```c
#include <stdio.h>

typedef enum  {  monday, tuesday, wednesday,
                 thursday, friday, saturday, sunday
              } daysofweek;
char *daysname[] = { "Mon", "Tue", "Wed", "Thu",
                     "Fri", "Sat", "Sun" };

int main() {
  for(int i = monday; i <= wednesday; i++)
    printf( "Day = %s\n" , daysname[i]);
}
```

12.10 Bit field

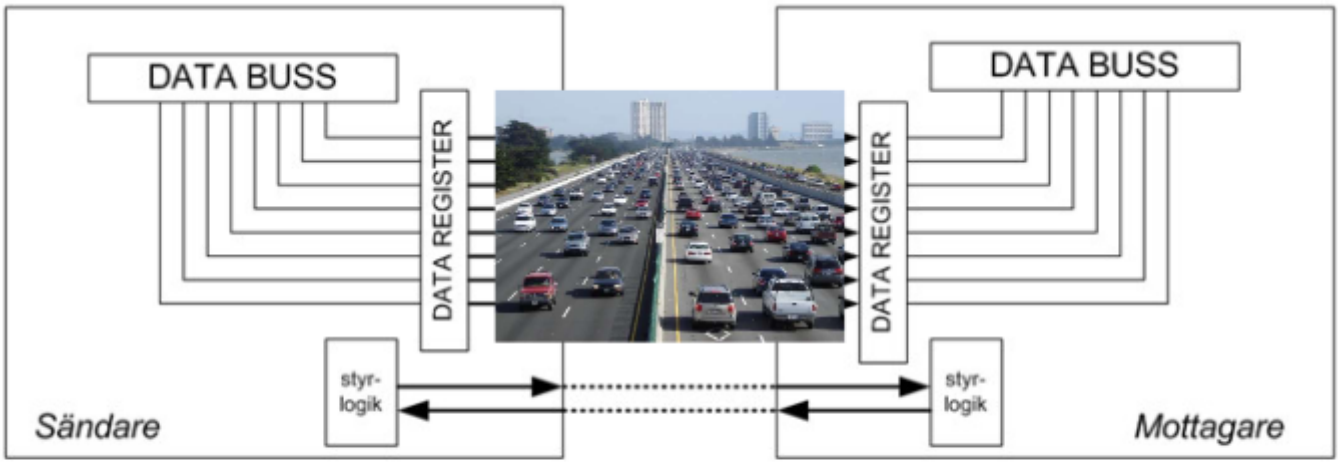
Kan dela upp variabler och ge olika delar av den som egna variabler.

```
struct S {
   unsigned int b1 : 5;
   unsigned int :0; // Start new unsigned int
   unsigned int b2 : 6;
   unsigned int b3 : 15;
};
```

Lecture No. 13

Seriell kommunikation

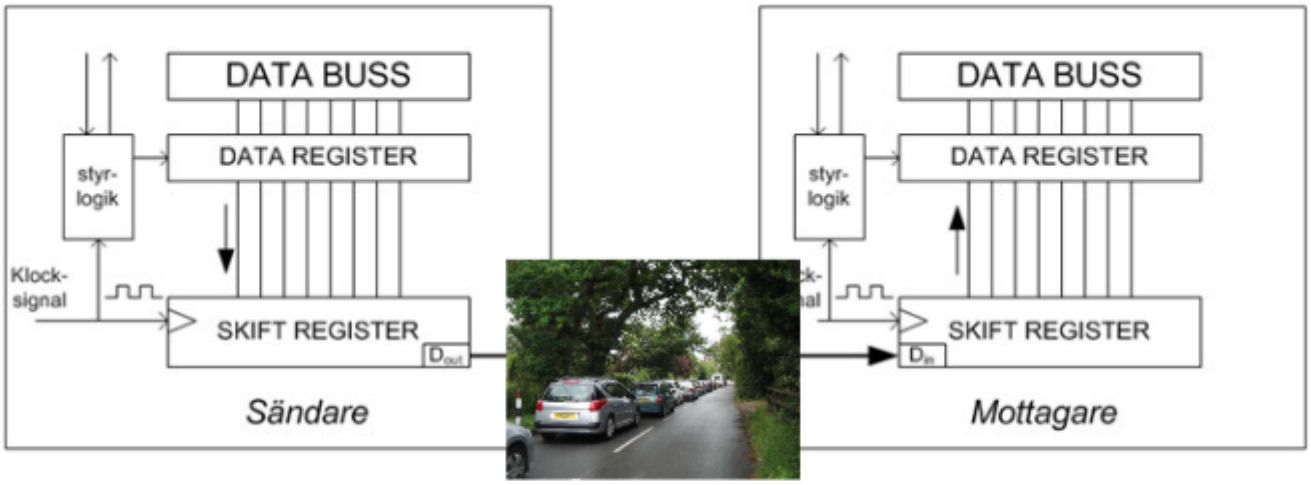
13.1 Vad är parallell överföring?



+ god bandbredd
- många ledare ger dyrare överföringsmedia
mycket snabbt vid korta avstånd

Väldigt dyrt pga exempelvis data loss.

13.2 Vad är seriell överföring?



+ få ledare ger enklare (billigare) överföringsmedia
- sämre bandbredd
överföringshastighet efter prestandakrav

13.3 Nätverkstopologi

13.3.1 point to point

En enhet kommunicerar till en annan

13.3.2 point to point full duplex

Två enheter pratar parallellt med varandra

13.3.3 Mask

Alla enheter kan vara kopplade till vilka andra som helst

13.3.4 Ring

Pratar i en ring

13.3.5 Star

Alla pratar via en central nod

13.3.6 Tree

Enheter kopplade i en trädlik struktur, endast en väg till samtliga andra

13.3.7 Bus

13.4 Nätverksprotokoll

13.4.1 Accessmetoder

De vanligaste är

- Master/slave
- CSMA/CD - Carrier sense multiple access collision detect
- CSMA/CR - Carrier sense multiple access Multiple access
- TDMA - Time division multiple access
- Token-ring

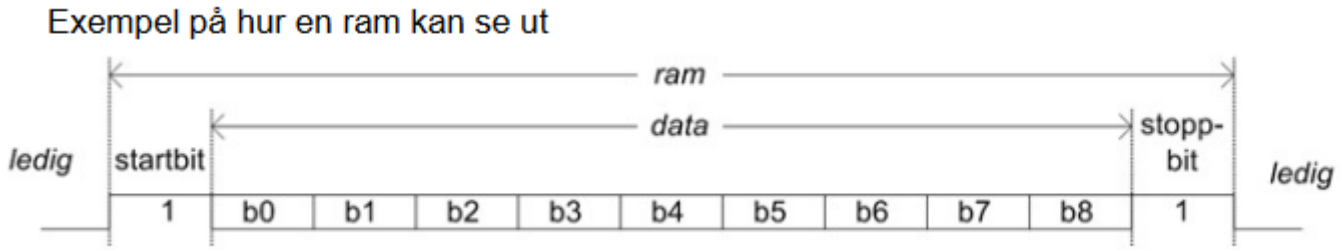
Ethernet använder CSMA/CD.

13.5 Asynkron överföring

Akta så de inte hamnar ur synk.

13.6 Klocksynkronisering

Ramar in varje byte i en frame med start- och slut-signal samt en paritetsbit för säkerhets skull.



13.7 USART

Universal Synchronous/Asynchronous Receiver/Transmitter
Ett fysiskt gränssnitt vi kan använda oss av för att skicka data mellan två eller flera enheter.
Värt att definiera USART:en som en strukt.

```
typedef struct tag_usart{
   volatile unsigned short sr;
   volatile unsigned short Unused0;
   volatile unsigned short dr;
   volatile unsigned short Unused1;
   volatile unsigned short brr;
   volatile unsigned short Unused2;
   volatile unsigned short cr1;
   volatile unsigned short Unused3;
   volatile unsigned short cr2;
   volatile unsigned short Unused4;
   volatile unsigned short cr3;
   volatile unsigned short Unused5;
   volatile unsigned short gtpr;
} USART,*PUSART;
#define USART1 ((USART *) 0x40011000)
#define USART2 ((USART *) 0x40004400)
#define USART3 ((USART *) 0x40004800)
#define USART4 ((USART *) 0x40004C00)
```

13.7.1 Joinkade metoder

```
void _outchar( char c ){
   while (( USART1->sr & (1<<7) )==0);
   USART1->dr = (unsigned short) c;
}
char _tstchar( void ){
   if( (USART1->sr & (1<<5) )==0)
   return 0;
   return (char) USART1->dr;
}
char _inchar( void ){
   char c;
   while ( (c=_tstchar() )==0);
   return c;
}
```

```
static char inbuf, outbuf;
void usart_init( void )
{
*((void (**)(void) ) USART1_IRQVEC ) = usart_irq_routine;
inbuf= 0;
*((unsigned int *) NVIC_USART1_ISER) |= NVIC_USART1_IRQ_BPOS;
USART1->brr = 0x2D9;
USART1->cr3 = 0;
USART1->cr2 = 0;
USART1->cr1 = UE | RXNEIE | TE | RE;
}
char usart_tstchar ( void ){
   char c = inbuf;
   inbuf = 0;
   return c;
}
void usart_outchar( char c ){
   outbuf = c;
   USART1->cr1 |= TXEIE;
}
void usart_irq_routine( void ){
   if(( USART1->cr1 & TXEIE) && (USART1->sr & TXE)){
      USART1->dr = (unsigned short) outbuf;
      USART1->cr1 &= ~TXEIE;
   }if( USART1->sr & RXNE){
      inbuf = (char) USART1->dr;
   }
}
```

Lecture No. 14

Standarbdbiblioteket och Runtimebiblioteket

14.1 Standardbiblioteket

libc/C standardbibliotek ären samling filer innehållande de absolut nödvändigaste c-funktionerna, saker som strcmp(),memcpy(),free(), osv.

14.2 Dynamisk minneshantering

Hittills i kursen har vi enbart sparat saker, statiskt i koden, om vi istället behöver kunna göra saker som ladda in filer, eller hantera databaser kommer vi behöva en variabel minnesrymd, för att programmet inte alltid ska ta upp lika stor minnesrymd om den teoretiskt kan så finns två funktioner för att kunna allokera ch deallokera minnesrymder; malloc() och free().

```
#include <stdlib.h>
#define TEXT_BUFFER_SIZE 1000

char * p;
p = (char *) malloc(TEXT_BUFFER_SIZE);
do_textbuffer_job( p );
free(p);
```

Eftersom dessa funktioner inte finns någonstans i simulatorn måste vi explicit säga till länkaren att dessa ska med för att koden ska kunna köras.

Lecture No. 15

Programbibliotek

Finns inte så mycket mer att säga om programbibliotek, stdio och stdlib är de vanligaste men det finns ett otömbart antal och de fungerar alla ungefär likadant.