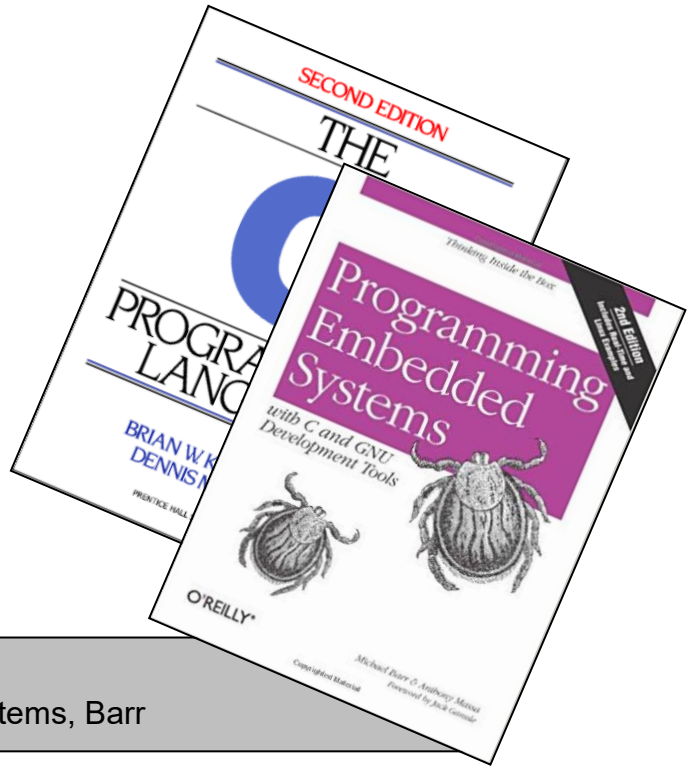
C-programmering

Programmering for innebygde datasystem

TTK4235



Kilder:

Kernighan & Ritchie (kap 1-7), Programming Embedded systems, Barr

Innhold

- Introduksjon c-programmering for innebygde datasystem
  - Hello world!
  - Nordic-øvingen
- Kommunikasjon med eksterne enheter
  - *Portmappet* og *Minnemappet* IO
  - Minnemapping via pekere og structs
  - Håndtering av IO-registre
- Håndtering av eksterne hendelser
  - Polling vs. interrupts
- Timere og *watchdog*-timere

Introduksjon

Hello world!

- Det klassiske åpningseksemplet
- Når en programmerer tilpassede datasystemer er det *lang vei* fram til «Hello world!»
- F.eks.:
  - Hvor skal meldingen vises (skjerm, serielinje, webside etc.)?
  - Hvordan formatteres meldingen for outputenheten?
  - Hvordan kommuniserer vi med den?
  - Hvordan laster vi inn programmet vårt og får det til å kjøre?
  - Har vi tilgang til de vanlige biblioteksfunksjonene som *printf()*?

Photo from www.arduino.cc

Hello world!

Du må kjenne maskinvaren du jobber med:

- Hvilken verktøykjede trengs for å bygge programmer for enheten? *(krysskompilering for hardwareplattformen)*
- Hvordan installeres programvaren på enheten? *(software/hardware for overføring av programmet til enheten)*
- Hvordan debugger vi programmet på enheten? *(debug monitor)*
- Hvilke mekanismer skal drive programmet ditt*? (polling, interrupts)*
- Hvordan adresserer du de eksterne enhetene programmet ditt skal jobbe med? *(portbasert eller minnemappet IO)*

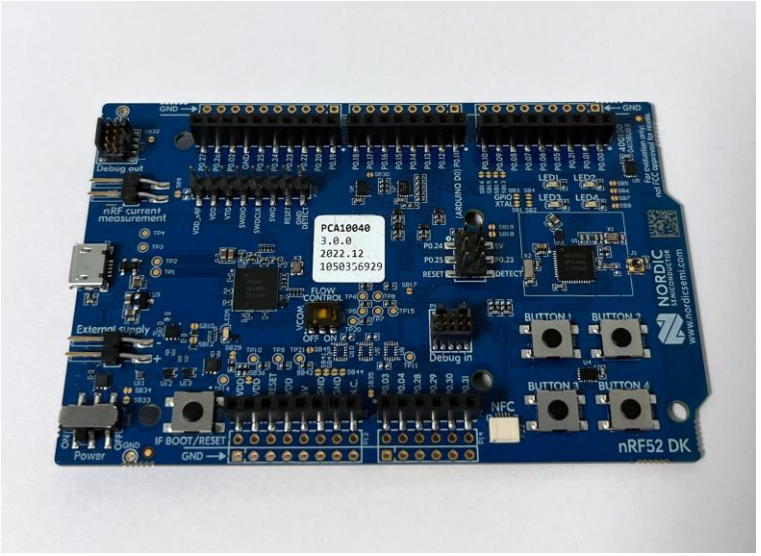
Datablad for maskinvaren er uunnværlig i dette arbeidet!

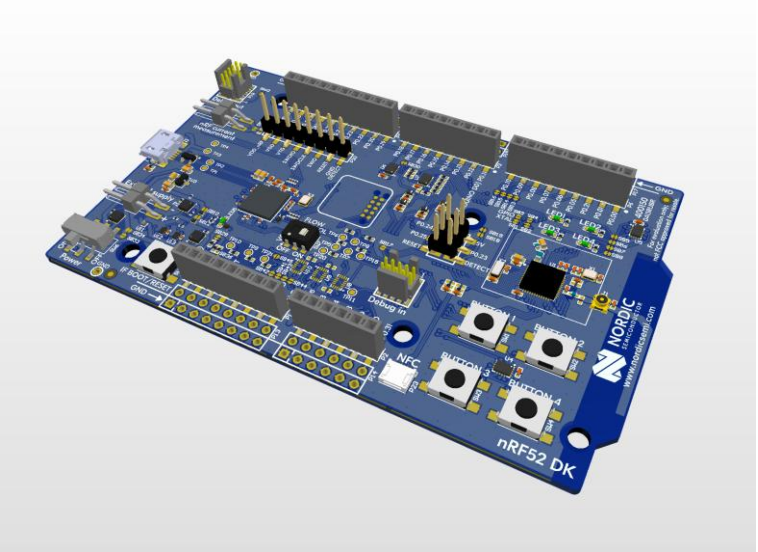
Nordic-øvingen

Nordic nRF52 DK inneholder en SoC (System on Chip) i nRF52-serien fra Nordic Semiconductor (ARM Cortex-M4).

Øvingen gir trening i programmering av mikrokontrollere, og tar i bruk flere elementer (GPIO, UART, GPIOTE, PPI, TWI).

Alle grupper får utlevert en DK-enhet til låns som skal brukes i labopplegget





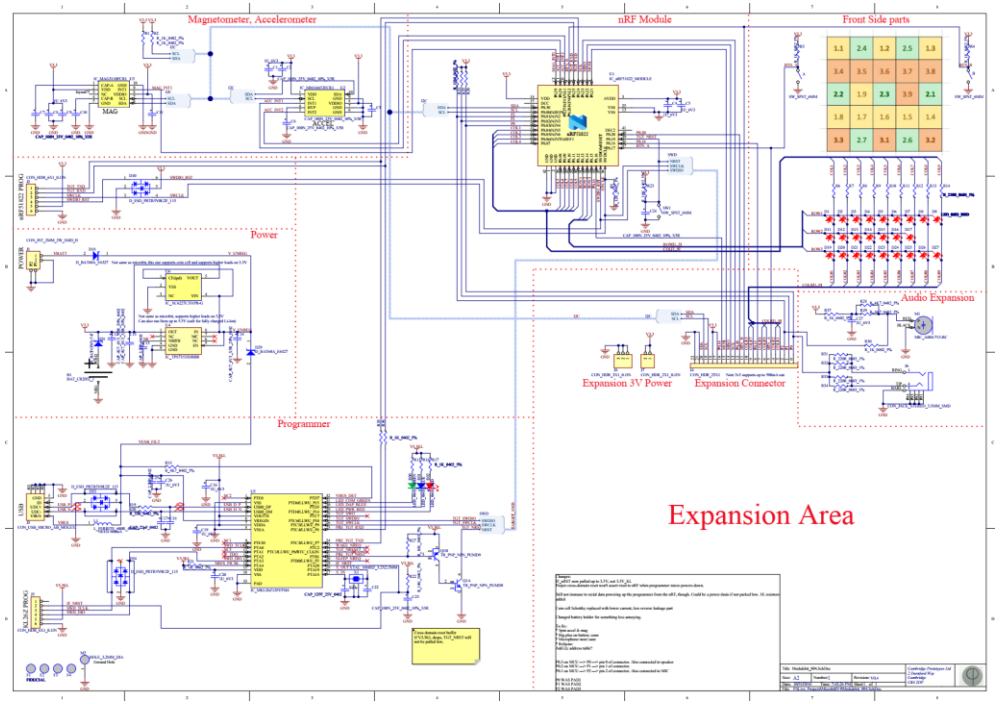
Kommunikasjon med eksterne enheter: memory mapped IO

Kommunikasjon med eksterne enheter

CPUen (og C-programmet ditt) trenger å kommunisere med eksterne enheter for:

- Konfigurasjon
- Datautveksling

Hovedalternativene er **minnemappet** (memory mapped) og **portmappet** (port mapped) IO



Koblingsskjema for micro:bit

Portmappet IO

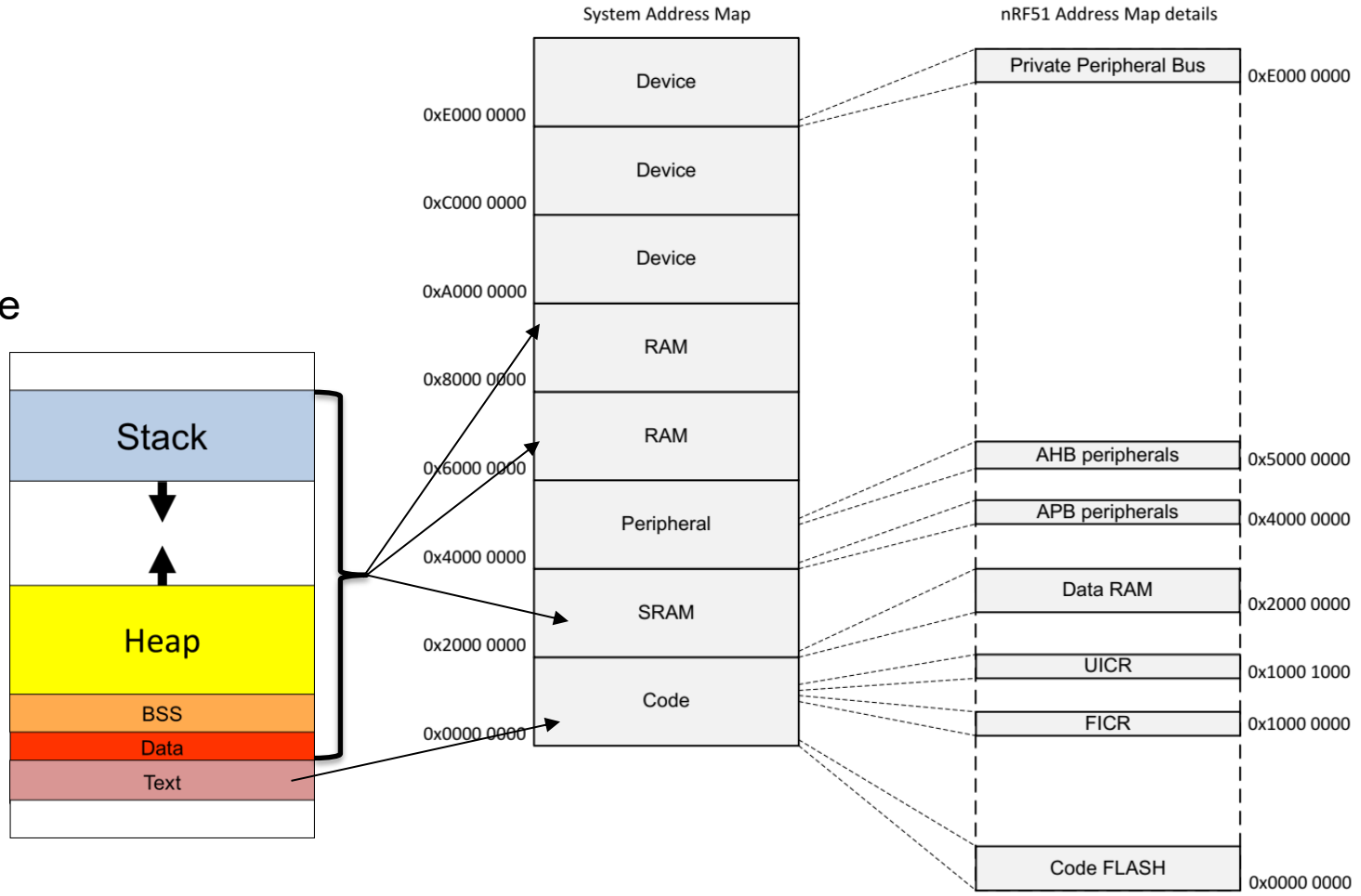
- IO-enheter adresseres i et separat adresserom fra RAM og ROM
- Egne prosessorinstruksjoner kreves for å aksessere IO-adresserommet
- Dette kan ikke gjøres via standard C-kode, og må støttes via verktøykjeden for den aktuelle plattformen
- Brukes f.eks. på x86 og x86-64, men ikke så vanlig på moderne mikrokontrollere

Minnemappet IO

Bruk av samme adresserom for å adressere minne og IO-enheter

Interne registre i IO-enheter fremstår som ordinære minneadresser som en kan lese fra eller skrive til.

Minnekartet for et system fremgår av databladet for en mikrokontroller eller SoC, samt av hva som er koblet opp mot IO-moduler



Minnekart fra datablad for nRF51-serien (Nordic Semiconductor)

Trade-off:

- *enklere for utviklere, men krever mer kompleks adressedekoder*
- *legger beslag på deler av adresserommet (mindre merkbart på 32- og 64-bits arkitekturer som har store adresserom)*

Eksempel: nRF51

Elementer under *Advanced Peripheral Bus* (APB) Peripherals:

ID	Base address	Peripheral	Instance	Description
0	0x40000000	POWER	POWER	Power Control
0	0x40000000	CLOCK	CLOCK	Clock control
0	0x40000000	MPU	MPU	Memory Protection Unit
1	0x40001000	RADIO	RADIO	2.4 GHz radio
2	0x40002000	UART	UART0	Universal Asynchronous Receiver/Transmitter
3	0x40003000	SPI	SPI0	SPI master 0
3	0x40003000	TWI	TWI0	Two-wire interface master 0
4	0x40004000	TWI	TWI1	Two-wire interface master 1
4	0x40004000	SPI	SPI1	SPI master 1
4	0x40004000	SPIS	SPIS1	SPI slave 1
6	0x40006000	GPIOTE	GPIOTE	GPIO tasks and events
7	0x40007000	ADC	ADC	Analog to digital converter
8	0x40008000	TIMER	TIMER0	Timer 0 This timer instance has 4 CC registers This timer instance supports BITMODE = 08Bit, 16Bit, 24Bit and 32Bit
9	0x40009000	TIMER	TIMER1	Timer 1 This timer instance has 4 CC registers This timer instance supports BITMODE = 08Bit and 16Bit

Eksempel: nRF51

General purpose IO Tasks and Events (GPIOTE):

15.2 Register Overview

Table 84: Instances

Base address	Peripheral	Instance	Description
0x40006000	GPIOTE	GPIOTE	GPIO Tasks and Events

Table 85: Register Overview

Register	Offset	Description
Tasks		
OUT[0]	0x000	Task for writing to pin specified in CONFIG[0].PSEL. Action on pin is configured in CONFIG[0].POLARITY.
OUT[1]	0x004	Task for writing to pin specified in CONFIG[1].PSEL. Action on pin is configured in CONFIG[1].POLARITY.
OUT[2]	0x008	Task for writing to pin specified in CONFIG[2].PSEL. Action on pin is configured in CONFIG[2].POLARITY.
OUT[3]	0x00C	Task for writing to pin specified in CONFIG[3].PSEL. Action on pin is configured in CONFIG[3].POLARITY.
Events		
IN[0]	0x100	Event generated from pin specified in CONFIG[0].PSEL
IN[1]	0x104	Event generated from pin specified in CONFIG[1].PSEL
IN[2]	0x108	Event generated from pin specified in CONFIG[2].PSEL
IN[3]	0x10C	Event generated from pin specified in CONFIG[3].PSEL
PORT	0x17C	Event generated from multiple input pins
Registers		
INTEN	0x300	Enable or disable interrupt
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
CONFIG[0]	0x510	Configuration for OUT[n] task and IN[n] event

Pekere i minnemappet IO

Pekere er mekanismen som brukes i C for å adressere spesifikke minneadresser, f.eks.:

Hvorfor *volatile*?

```
uint32_t volatile *dev_reg = (uint32_t volatile *)0x80000000;
```

`dev_reg` vil nå peke på minneadressen 0x80000000

Alternativt – for å unngå å definere en pekervariabel som tar plass i minnet:

```
#define DEV_REG ((uint32_t volatile *)0x80000000)

#define DEV_REG2 (*((uint32_t volatile *)0x80000000))
```

Hva er forskjellen på `DEV_REG` og `DEV_REG2`?

Bruk av *volatile*

Modifieren *volatile* forteller kompilatoren at verdien av en variabel kan endre seg når som helst, uavhengig av programkode som kompilatoren finner «i nærheten».

Dette hindrer optimaliseringer hvor variabelen inngår.

Deklarasjoner:

```
uint32_t volatile x; // Volatil integer
volatile uint32_t x2; // Volatil integer
uint32_t volatile * p_x; // Peker til volatil integer
uint32_t * volatile p2_x; // Volatil peker til integer
uint32_t volatile * volatile p3_x;
 // Vol. peker til vol. integer
```

Mest relevant for oss

Bruk av *volatile*

Eksempel:

```
uint32_t volatile *pGpio0Set = (uint32_t volatile *)(0x40E00018);

void gpioFunction(void)
{
     /* Set GPIO pin 0 high. */
     *pGpio0Set = 1;                /* First write. */

     delay_ms(1000);

     /* Set GPIO pin 1 high. */
     *pGpio0Set = 2;                /* Second write. */
}
```

Hvorfor er *volatile* viktig her?

Uten *volatile* vil kompilatoren anse første skriveoperasjon som overflødig, og denne kan da optimaliseres bort!

Bruk av *volatile*

Volatile brukes i hovedsak i følgende tilfeller:

1. Minnemappede registre
2. Globale variable som kan endres av en *interrupt service routine*
3. Globale variable som brukes av flere parallelle tråder

… men *volatile* i seg selv sikrer ikke mot alt som kan gå galt!

Pekere i minnemappet IO

Med pekeren til en minneadresse kan vi lese og skrive som om det gjaldt en ordinær variabel:

```c
#include <stdio.h>

uint32_t volatile *dev_reg = (uint32_t volatile *)0x80000000;

void main() {

    // Les registerverdi:
    uint32_t value = *dev_reg;

    // Sett registerverdi:
    *dev_reg = 2;

}
```

Kommunikasjon med eksterne enheter: minnemapping via struct

Minnemapping via struct

IO-enheter vi ønsker å operere mot har typisk en rekke registre med forskjellig rolle.

Vi kan bruke structs kreativt for å adressere disse på en enkel måte!

14.2 Register Overview

Table 74: Instances

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Table 75: Register Overview

Register	Offset	Description
Registers		
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register
PIN_CNF[0]	0x700	Configuration of GPIO pins
PIN_CNF[1]	0x704	Configuration of GPIO pins
PIN_CNF[2]	0x708	Configuration of GPIO pins
PIN_CNF[3]	0x70C	Configuration of GPIO pins
PIN_CNF[4]	0x710	Configuration of GPIO pins

Eksempel: noen av GPOI-registrene i nRF51

Minnemapping via struct

Structs er samlinger av navngitte variable

&a er en int-peker, så vi må bruke eksplisitt cast for å tilordne denne til **p**

Programmet skriver ut:

2

4

6

```
#include <stdio.h>

void main() {
    int a[4] = {2, 4, 6, 8};
    typedef struct {
        int a;
        int b;
        int c;
    } mystruct;
    mystruct m = {1, 2, 3};
    mystruct *p = &m;

    p = (mystruct*)&a;

    printf("a = %d\n", p->a);
    printf("b = %d\n", p->b);
    printf("c = %d\n", p->c);
}
```

Minnemapping via struct

Steg 1:
Basert på datablad, definer en *struct* som matcher registrene fra baseadressen:

```
typedef struct {
    volatile uint32_t IN;
    volatile uint32_t OUT;
    volatile uint32_t UNUSED[2];
    volatile uint32_t CTRL;
} MY_IO_REGS;
```

En slik struct vil ligge i minnet som 5 påfølgende blokker på 32 bits hver.

Minnemapping via struct

Steg 2:

Opprett en peker til baseadressen (f.eks. 0x80000000), og typecast denne til en peker for typen `MY_IO_REGS`:

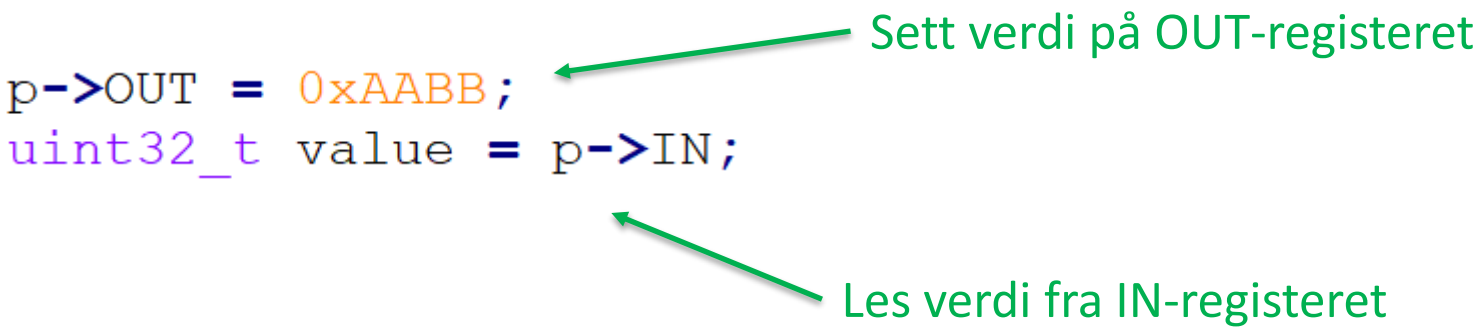
```
MY_IO_REGS *p = (MY_IO_REGS*)0x80000000;
```

Med andre ord ***later vi som*** det ligger en `MY_IO_REGS`-struct i området som starter på adressen 0x80000000

Minnemapping via struct

Steg 3:

Bruk feltene i structen etter behov:



Når vi adresserer feltene i structen vil vi faktisk adressere baseadressen 0x80000000 pluss offseten til det aktuelle feltet.

Minnemapping via struct

En trenger ikke å opprette en varig peker til structen, men kan i stedet bruke en makro:

```
#define MY_REGS ((MY_IO_REGS*)0x80000000)
...
uint32_t value = MY_REGS->IN;
```

Kommunikasjon med eksterne enheter: registeroperasjoner

Lese/skrive IO-registre

Gitt en enkel struct,
og pekeren `IOREGS`:

```
typedef struct {
    volatile uint32_t IN;
    volatile uint32_t OUT;
} REGS;
REGS *IOREGS = (REGS*)0x80000000;
```

Lese ut verdien til IN, pinne 6:

```
val = IOREGS->IN & 0x0040;
// eller
val = IOREGS->IN & 64;
```

(val > 0 betyr at bit 6 er høy)

Lese/skrive IO-registre

Slå på pinne 0 og 4 på IOREGS->OUT:

```
// Enten
IOREGS->OUT |= 1<<0 | 1<<4;
// eller
IOREGS->OUT |= 0x0011;
// eller
IOREGS->OUT |= 16 | 1;
```

… og slå av pinne 4:

```
IOREGS->OUT &= 0xEF;
// eller
IOREGS->OUT &= ~(1<<4);
// eller
IOREGS->OUT &= ~16;
```

Lese/skrive IO-registre

Vi kan gjøre koden mer leselig ved å gi relevante navn til pinnene på en inngang/utgang ved hjelp av **#define**

```
#define MOTOR_FWD 1
#define MOTOR_REV 2
#define LIGHT 4
```

Slå på MOTOR_FWD og LIGHT (pinne 0 og 2), og slå av de resterende:

```
IOREGS->OUT = MOTOR_FWD | LIGHT;
```

Bruk av IO-registre i praksis

Merk:

- Effekten av lesing/skriving av et register avhenger av registeret
- Tidligere konfigurasjon kan også ha betydning
- Hver bit eller segment/byte kan ha ulik betydning i et 32-bits register

Databladet for den aktuelle mikrokontrolleren eller tilkoblede enheten må brukes flittig!

Eksempel: nRF51

Register for konfigurasjon av GPIOTE-kanal:

Segmenter (A-D)

Verdier og betydning

Forklaring; merk forskjellig effekt avhengig av modus!

Table 89: CONFIG[n]

Bit number				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
Id															D			C	C				B	B	B	B	B							A	A
Reset				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Id	**RW**	**Field**	**Value Id**	**Value**																															
A	RW	MODE																																	
			Disabled	0																															
			Event	1																															
			Task	3																															
B	RW	PSEL		[0..31]																															
C	RW	POLARITY																																	
			None	0																															
			LoToHi	1																															
			HiToLo	2																															
			Toggle	3																															
D	RW	OUTINIT																																	
			Low	0																															
			High	1																															

Description
Mode
Disabled. Pin specified by PSEL will not be acquired by the GPIOTE module.
Event mode The pin specified by PSEL will be configured as an input and the IN[n] event will be generated if operation specified in POLARITY occurs on the pin.
Task mode The pin specified by PSEL will be configured as an output and triggering the OUT[n] task will perform the operation specified by POLARITY on the pin. When enabled as a task the GPIOTE module will acquire the pin and the pin can no longer be written as a regular output pin from the GPIO module.
Pin number associated with OUT[n] task and IN[n] event
When In task mode: Operation to be performed on output when OUT[n] task is triggered. When In event mode: Operation on input that shall trigger IN[n] event.
Task mode: No effect on pin from OUT[n] task. Event mode: no IN[n] event generated on pin activity.
Task mode: Set pin from OUT[n] task. Event mode: Generate IN[n] event when rising edge on pin.
Task mode: Clear pin from OUT[n] task. Event mode: Generate IN[n] event when falling edge on pin.
Task mode: Toggle pin from OUT[n]. Event mode: Generate IN[n] when any change on pin.
When in task mode: Initial value of the output when the GPIOTE channel is configured. When in event mode: No effect.
Task mode: Initial value of pin before task triggering is low
Task mode: Initial value of pin before task triggering is high

Eksempel: nRF51

Utdrag fra listen over GPIO-registre:

Her tilbys tre forskjellige registre for setting av utgangene:

- OUT: ordinær tilordning
- OUTSET: setting av valgte bits til 1
- OUTCLR: setting av valgte bits til 0

14.2 Register Overview

Table 74: Instances

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Table 75: Register Overview

Register	Offset	Description
Registers		
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register

F.eks., gitt en definert struct for GPIO:

```
GPIO->OUTSET = 0x01000000;
```

Vil sette en enkelt pinne (hvilken?) på GPIO til 1, og la de resterende være uendret.

Eksempel: nRF51

Dokumentasjon av OUTSET og OUTCLR:

Table 77: OUTSET

Note: Read: reads value of OUT register.
Note: Individual bits are set by writing a '1' to the bits that shall be set. Writing a '0' will have no effect.

Bit number					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
Id					AF	AE	AD	AC	AB	AA	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**																															
A	RW	PIN0			Pin 0																															
			Low	0	Read: pin driver is low																															
			High	1	Read: pin driver is high																															
			Set	1	Write: writing a '1' sets the pin high																															

Table 78: OUTCLR

Note: Read: reads value of OUT register.
Note: Individual bits are cleared by writing a '1' to the bits that shall be cleared. Writing a '0' will have no effect.

Bit number					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
Id					AF	AE	AD	AC	AB	AA	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**																															
A	RW	PIN0			Pin 0																															
			Low	0	Read: pin driver is low																															
			High	1	Read: pin driver is high																															
			Clear	1	Write: writing a '1' sets the pin low																															

Eksamen 2019, Oppgave 2

Oppgave 2: Minnemappet I/O (21%)

Styringssystemet i robotgressklipperen skal baseres på en nRF51-chip. Figur 2 viser et utdrag fra databladet til enheten hvor du kan se deler av registeroversikten for GPIO-enheten (General-Purpose Input/Output). Du programmerer systemet i C.

14.2 Register Overview

Table 74: Instances

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Table 75: Register Overview

Register	Offset	Description
Registers		
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register
PIN_CNF[0]	0x700	Configuration of GPIO pins
PIN_CNF[1]	0x704	Configuration of GPIO pins

Figur 2: Utdrag fra datablad for mikrokontroller

a)

Noen datasystemer bruker «*memory mapped IO*», mens andre bruker «*port mapped IO*». Beskriv kort forskjellen på disse systemene med tanke på hvordan du kan kommunisere med eksterne enheter via C-kode?

b)

nRF51 bruker memory mapped IO. Hvordan går du fram i C dersom du skal sette høy verdi på pinnene 0 og 2 i registeret OUT uten å endre verdien på de resterende pinnene? Gi svaret i form av C kode (bare de relevante linjene, du trenger ikke å skrive et fullverdig program).

c)

Du kan bruke en *struct* for å adressere alle registrene tilhørende GPIO-enheten. Beskriv fremgangsmåten ved hjelp av C-kode (bare de relevante linjene, du trenger ikke å skrive et fullverdig program). Bruk bare de første registrene som eksempel. Hvordan håndterer du sprang i adressene (som f.eks. mellom DIRCLR og PIN_CNF[0]) når du setter opp structen?

Vis hvordan du bruker structen til å skrive en verdi til et av registrene i GPIO-enheten.

I tilfelle du skulle ha nytte av det har vi laget en konverteringstabell for de hexadesimale tallene i utdraget fra databladet:

Hex	504	508	50C	510	514	518	51C	700	704
Dec	1284	1288	1292	1296	1300	1304	1308	1792	1796

Håndtering av eksterne hendelser: interrupts og polling

Håndtere eksterne hendelser

Eksterne hendelser er signaler utenfra som kan utløse en aksjon i prosessoren.

Slike hendelser kan f.eks. være:

- Data som kommer inn fra en ekstern enhet
- En timer som signaliserer at tiden er ute
- At brukeren har trykket på en knapp

En kan typisk velge å håndtere disse enten via **polling** eller **interrupts**

Polling vs interrupts

Polling:

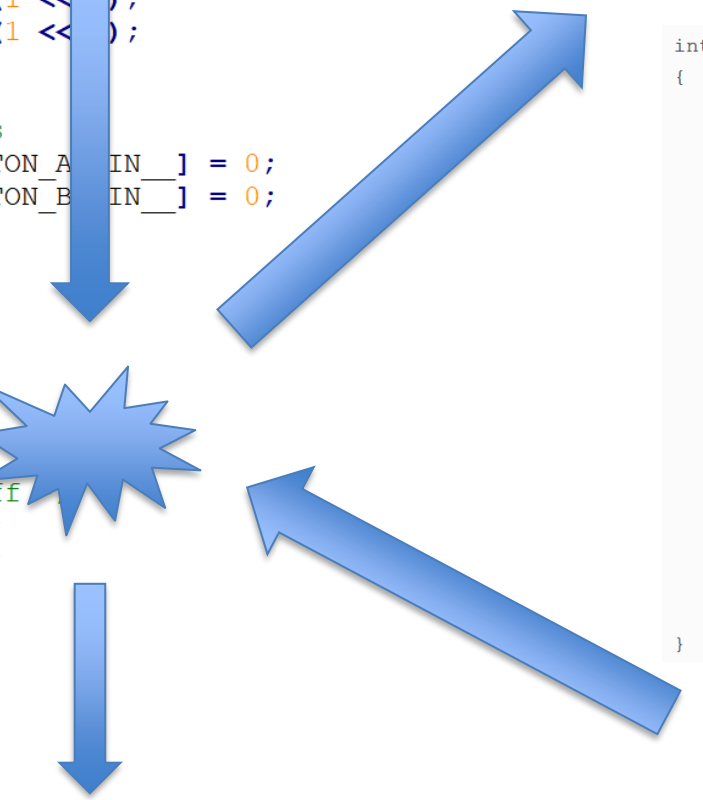
- Programmet sjekker status på eksterne enheter regelmessig og håndterer evt. hendelser
- Enkelt!
- Kan være ineffektivt, da programmet stadig er opptatt med pollingen, og det kan ta tid før programmet oppdager hendelser

Interrupts:

- Eksterne hendelser fører til at programmet automatisk avbrytes for å håndtere dem
- Hovedprogrammet kan utføre andre aktiviteter i periodene mellom hendelser
- Generelt mer effektivt og oversiktlig, men gir også utfordringer!

Interrupts

```
int main(){
    // Configure LED Matrix
    for(int i = 4; i <= 15; i++){
        GPIO->DIRSET = (1 << i);
        GPIO->OUTCLR = (1 << i);
    }

    // Configure buttons
    GPIO->PIN_CNF[__BUTTON_A_PIN__] = 0;
    GPIO->PIN_CNF[__BUTTON_B_PIN__] = 0;

    int sleep = 0;
    while(1){

        /* Do stuff */
        doStuff1();
        doStuff2();

        /* Do other stuff */
        doOtherStuff1();
        doOtherStuff2();

        sleep = 10000;
        while(--sleep);
    }
    return 0;
}
```



```
interrupt_void interruptServiceRoutine(void)
{
    uint8_t intStatus;

    /* Determine which interrupts have occurred. */
    intStatus = *pIntStatusReg;

    /* Acknowledge the interrupt. */
    *pIntStatusReg = intStatus;

    if (intStatus & INTERRUPT_SOURCE_1)
    {
        /* Do interrupt processing. */
    }

    if (intStatus & INTERRUPT_SOURCE_2)
    {
        /* Do interrupt processing. */
    }
}
```

Interrupts

En liste over nummererte interrupts kan finnes i dokumentasjonen til en prosessor eller mikrokontroller

Interruptene kan ha ulik prioritet, som får betydning hvis flere kommer samtidig

Table 8-1. Partial interrupt list for PXA255 processor

Interrupt number	Interrupt source
8	GPIO Pin 0
9	GPIO Pin 1
11	USB
26	Timer 0
27	Timer 1
28	Timer 2

Eksempel på interruptliste fra boka

Interrupts

Noen interrupts kan aktiveres/deaktiveres i software (*maskerbare* interrupts)

Dette gjøres f.eks. via bits i et register i interrupt controlleren:

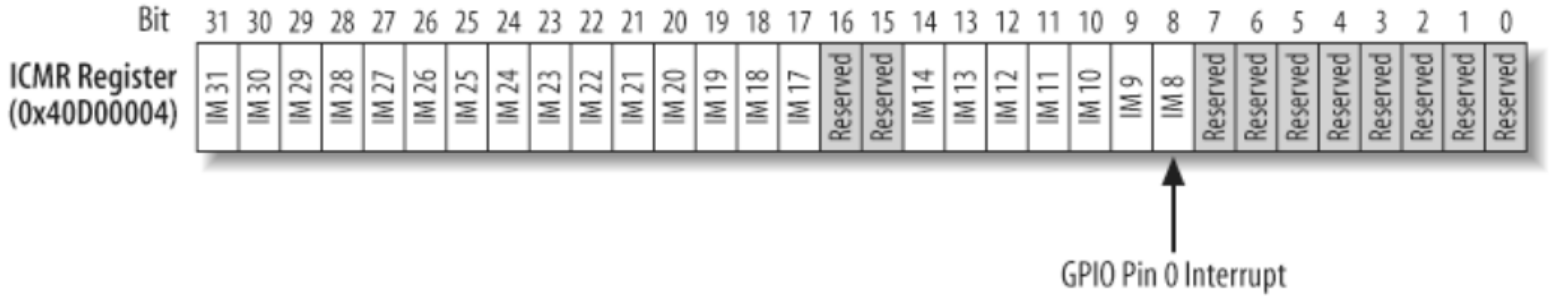


Figure 8-3. PXA255 Interrupt Controller Mask Register

Interrupt Service Routine (ISR)

Når en interrupt skal håndteres vil prosessoren bruke en *Interrupt Vector Table* for å finne adressen til dennes ISR

På adressen i tabellen er det bare plass til en peker til en funksjon et annet sted i minnet. Denne kan peke til vår ISR

Table 8-3. ARM interrupt vector table

Exception/interrupt source	Address
Reset	0x00000000
Undefined instruction	0x00000004
Software interrupt	0x00000008
Prefetch abort	0x0000000C
Data abort	0x00000010
IRQ	0x00000018
FIQ	0x0000001C

Interrupt Service Routine (ISR)

Hvordan definerer vi en ISR?

- Siden språket C prøver å holde seg unna plattformavhengige detaljer er det opp til kompilatorene å støtte notasjon
- Dette løser de på forskjellige måter

Sjekk dokumentasjonen for din plattform og verktøykjede!

```
void __attribute__((interrupt("IRQ"))) do_irq()
{
    //your irq service code goes here
}
```

ARM (gcc)

```
#include <avr/interrupt.h>

ISR(ADC_vect)
{
    // user code here
}
```

AVR (gcc)

```
14  void init_adc(unsigned char inngang)
15  {
16    ADMUX = (1 << REFS0) |
17            (1 << ADLAR) |
18            (inngang & 0x07);
19
20    ADCSR = (1 << ADEN) |
21            (1 << ADSC) |
22            (1 << ADFR) |
23            (1 << ADIE) ;
24  }
25  #pragma vector=ADC_vect
26  __interrupt void speilegg(void)
27  {
28      dac_write(0, ADCH);
29  }
30
31  void main( void )
32  {
33      //TCCR0 = 4;
34
```

AVR (IAR Systems)

Interrupt Service Routine (ISR)

En ISR har følgende oppgaver:

- Lagre prosessorens tilstand
  - Noen prosessorer gjør det automatisk
  - Med riktige direktiv vil kompilatoren autogenerere koden
- Bekrefte interrupten
- Utføre nødvendige aksjoner
- Gjenopprette prosessorens tilstand

En ISR bør ha kort kjøretid. Større operasjoner må gjøres utenfor
(*f.eks. ved å sette et flagg slik at hovedløkka i programmet ser at operasjonen skal gjøres*)

Interrupt Service Routine (ISR)

Eksempel fra boka:

Indikerer for kompilatoren at dette er en ISR

Les ut hvilke interrupts som har inntruffet (hw-avhengig)

Acknowledge ved å skrive verdien tilbake (hw-avhengig)

Utfør aksjoner for de interruptene som har inntruffet

```
interrupt void interruptServiceRoutine(void)
{
    uint8_t intStatus;

    /* Determine which interrupts have occurred. */
    intStatus = *pIntStatusReg;

    /* Acknowledge the interrupt. */
    *pIntStatusReg = intStatus;

    if (intStatus & INTERRUPT_SOURCE_1)
    {
        /* Do interrupt processing. */
    }

    if (intStatus & INTERRUPT_SOURCE_2)
    {
        /* Do interrupt processing. */
    }
}
```

Delte data og race conditions

Fordi en ISR kan bryte inn når som helst under kjøringen av hovedtråden i programmet, og operere på samme data, kan det oppstå *race conditions* som fører til feil.

Race condition: *En situasjon der utfallet i et system avhenger av sekvensen eller timingen av uforutsigbare hendelser*

Delte data og race conditions

Eksempel:

Hva er problemet
med denne koden?

```
int gIndex = 0;

interrupt void serialReceiveIsr(void)
{
    /* Store receive character in memory buffer. */

    gIndex++;
}

int main(void)
{
    while (1)
    {
        if (gIndex)
        {
            /* Process receive character in memory buffer. */

            gIndex--;
        }
    }
}
```

Delte data og race conditions

Problemet:

- Både hovedprogrammet og ISRen opererer på *gIndex*
- Operasjonen *gIndex--* er *ikke atomisk*

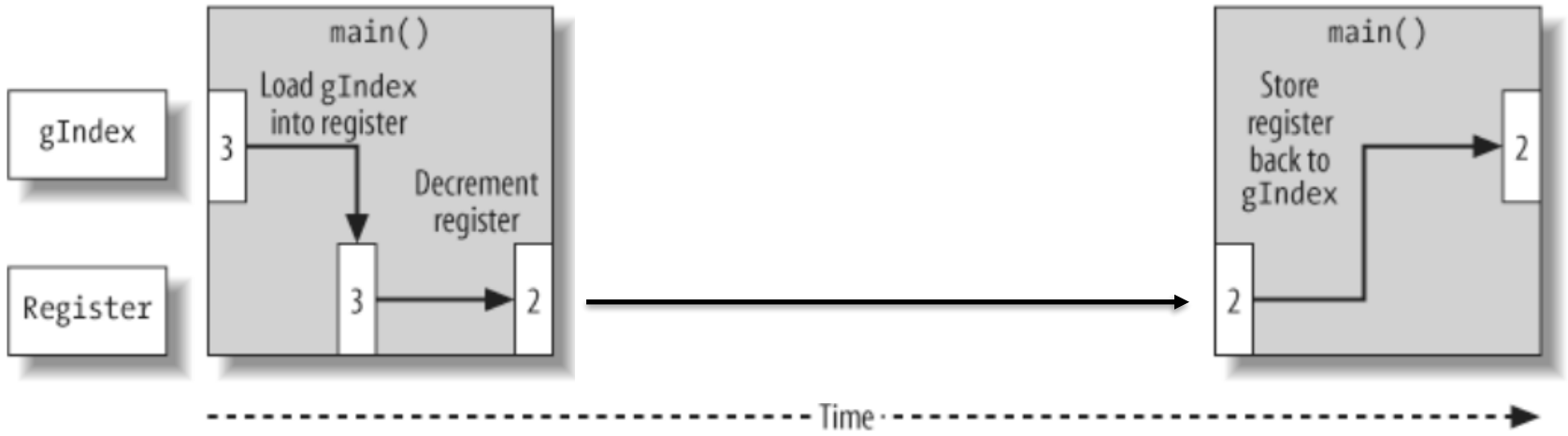


Figure 8-5. Example race condition

Delte data og race conditions

Problemet:

- Både hovedprogrammet og ISRen opererer på *gIndex*
- Operasjonen *gIndex--* er *ikke atomisk*

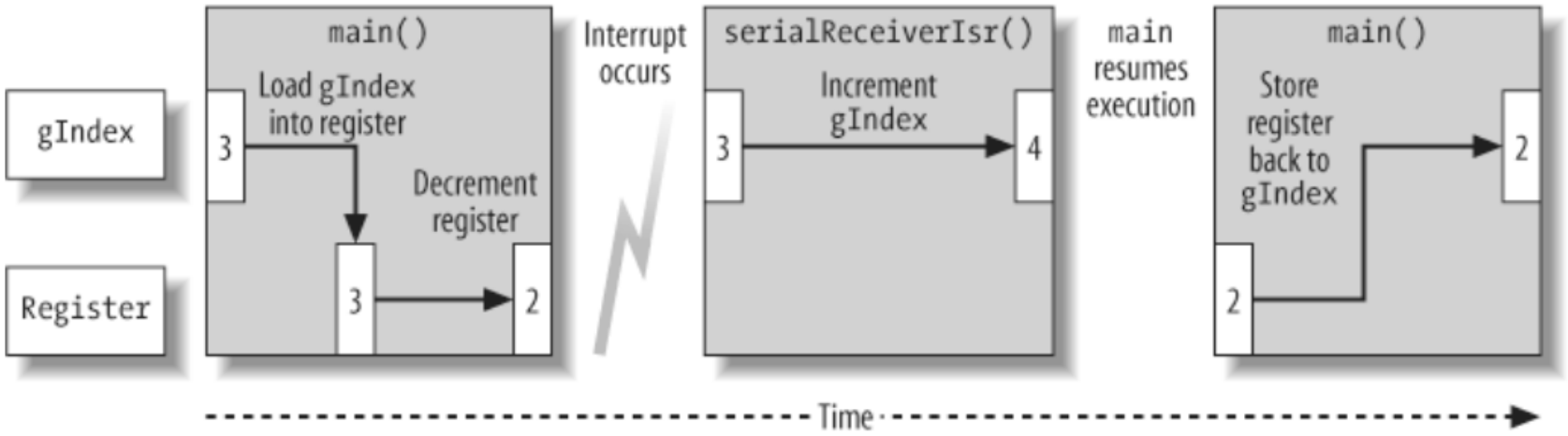


Figure 8-5. Example race condition

Delte data og race conditions

Løsningen:

Deaktiver interrupts i den kritiske fasen når hovedprogrammet jobber med *gIndex*:

Kan du se et potensielt problem med denne løsningen?

Interrupts må bare slås på her dersom de var aktive i utgangspunktet!

```
int main(void)
{
    while (1)
    {
        interruptDisable();

        if (gIndex)
        {
            /* Process receive character in memory buffer. */

            gIndex--;
        }

        interruptEnable();
    }
}
```

Timere og watchdog-timere

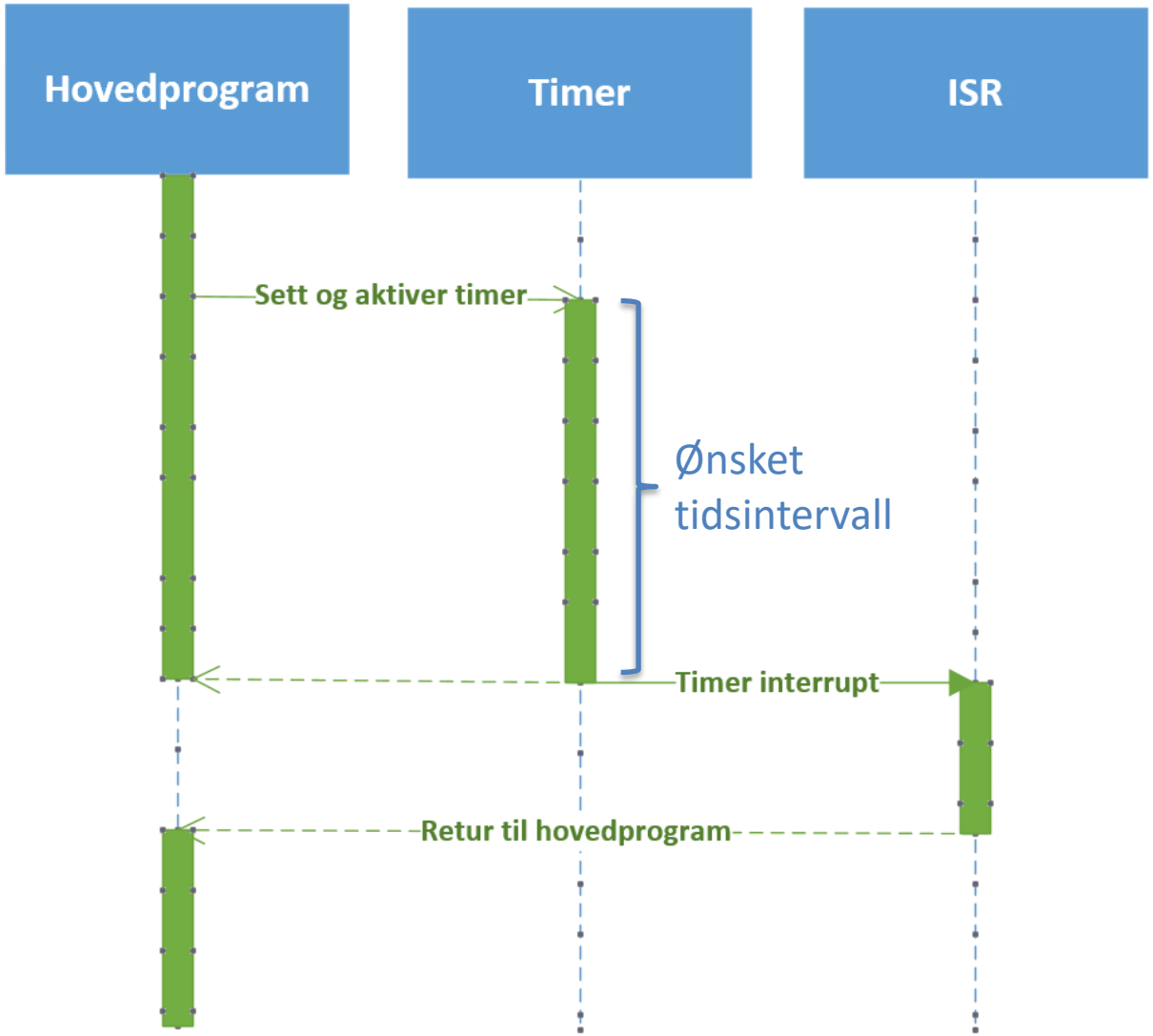
Timere

Mikrokontrollere er utstyrt med *timer*-enheter.

- Enheter som kan settes opp til å generere interrupts med gitte tidsintervaller
- Disse kan bruke til å håndtere tidsavhengige eller regelmessige operasjoner (f.eks. den blinkende LEDen i boka)
- Mer nøyaktig enn *busy wait (som er likt polling)*
- Frigjør hovedtråden i programmet til å gjøre andre ting

Timere

Eksempel på sekvens ved bruk av timer:



Timere

Oppsett av timere er avhengig av hardware, men semi-generelt:

- Sett verdi i timer-register (basert på nåverdi og intervall)
- Aktiver timer-interrupt i timer-enheten
- Aktiver timerens interrupt i interrupt controller
- Lag ISR som bekrefter, tar aksjon og evt. oppdaterer timer-register

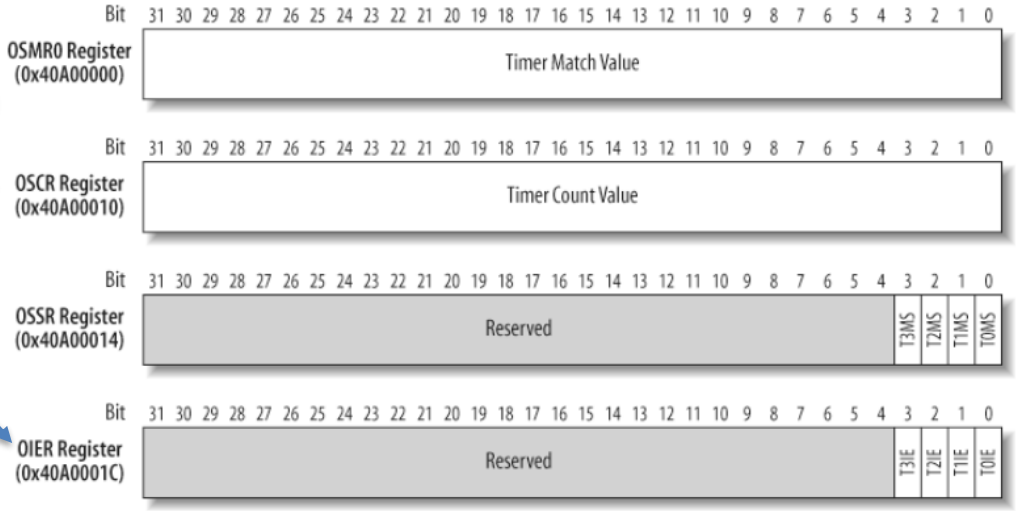


Figure 8-6. PXA255 processor timer 0 registers

Watchdog-timere

Tilpassede datasystemer skal ofte fungere autonomt over lang tid. Noen kan bare fjernstyres (f.eks. instrumentering på en satellitt).

Hvordan sikrer vi at kræsj eller heng i programvaren ikke setter systemet ut av spill?

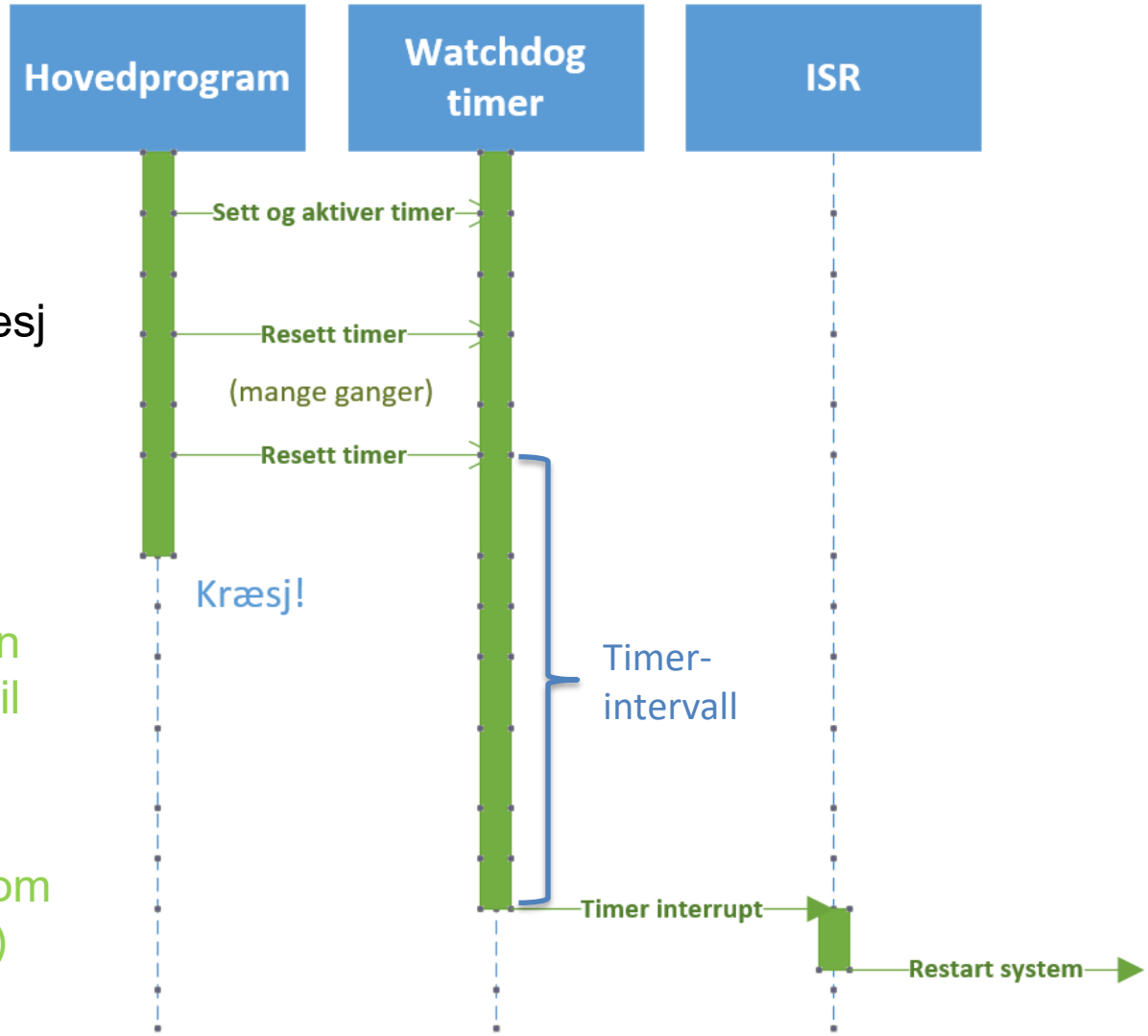
Illustrasjon fra NTNU SmallSat

Watchdog-timere

- Timere som (hvis aktivert) må resettes av programmet innen en viss tid for ikke å generere en interrupt
- Hovedprogrammet lages slik at det regelmessig resetter watchdog-timeren – lenge før timeout!
- ISRen for watchdog-timeren kan ha i oppgave å gi systemet en hard reset
- Dersom programmet kræsjer eller henger vil resettingen utebli, og watchdog-timeren genererer en interrupt når tiden går ut

Watchdog-timere

Eksempel på sekvens der systemet restartes etter kræsj i hovedprogrammet:



Er det en god idé å bruke en timer med en ISR satt opp til å resette watchdog'en?

(Nei – da oppdager vi ikke om hovedprogrammet kræsjer!)