Pinudløst interrupt:

Dette dokument gennemgår hvordan man opsætter pin-interrupts i Arduino.

Links i dokumentet:

Hvad er et interrupt.

Det program, der normalt kører på en controller, er en række af sekventielle instruktioner udført efter hinanden i et **loop**. Dvs. at programmet kører gennem loopet, og når det når til enden af loopet, starter programmet loopet forfra igen.

Den tid, et loop tager, er derfor afhængig af hvor stort programmet er.

Hvis man så har et ret langt program, kan det tage noget tid, før programmet fx spørger til en inputknap.

Heldigvis er der en løsning. I Arduino Uno'en er der mulighed for at indstille det sådan, at to pins kan sættes til at udløse et interrupt, når de fx går høj.

Et interrupt er **et event** – eller **hændelse**, - der får en specielt programsekvens til at udføres **straks**.

Altså: Når et Interrupt opstår, afbrydes det kørende program øjeblikkeligt, og en interrupt service rutine, dvs. en subrutine, kaldes og koden heri udføres.

Interrupts udløses – og udføres et vilkårligt sted i et kørende program – og uafhængig heraf.

En Interruptrutine er en programstump, der udløses af en hændelse og kører asynkront, dvs. ikke i et loop, men som en kort selvstændig programdel, med et start - og et slut-punkt. Når ISR'en er udført, fortsætter processoren, hvor den slap.

Det vi ikke ser (i C-kode), er, at processoren automatisk gemmer (i Ram'en) adressen på næste instruktion, den skulle til at udføre. Ellers ville den jo ikke kunne vide, hvortil i programmet, - i loop, den var kommet. Det register kaldes Program Counter.

Ligeledes gemmes vitale registres indhold i RAM, i et område kaldet "stakken". Processoren kunne jo være igang med en udregning, da interruptet opstod. Det sker i baggrunden, og det er C-compileren, der genererer kode til det.

Når ISR'en er færdig, hentes de gemte værdier, som processoren var ved at arbejde med da interruptet opstod, ind i registerene igen, og loop() fortsætter som om intet er hændt.

Typisk er der gjort brug af interrupts allerede i de tidligere opgaver, der er lavet. De er bare "gemt"!

Det er fordi, Arduino´s underlæggende compiler automatisk indbygger mulighed for at bruge funktioner som, millis() , micros() og delayMicroseconds(). Disse funktioner gør brug af interrupts og bruger timere!!

PWM-funktionen analogWrite() bruger timere, ligesom tone(), noTone() og Servo library gør.

Og seriel kommunikation bruger også interrupts!!! Det sker, når der er modtaget en hel byte, og den skal lægges i modtagebufferen. Og ligeledes i sendebufferen !!

Alle interrupts får processoren til at hoppe til en bestemt adresse i program-hukommelsen.

Her er der ikke meget plads til kode, men nok plads til en "Jump" ordre.

Her ses en oversigt:

ATmega328 Interrupts

VectorNo.	Program Address(2)	Source	Interrupt Definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B

VectorNo.	Program Address(2)	Source	Interrupt Definition
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Kilde: https://courses.cs.washington.edu/courses/csep567/10wi/lectures/Lecture7.pdf

Interrupt-model

Interrupt Model

- When an interrupt event occurs:
  - Processor does an automatic procedure call
  - CALL automatically done to address for that interrupt
    - Push current PC, Jump to interrupt address
  - Each event has its own interrupt address
  - The global interrupt enable bit (in SREG) is automatically cleared
    - i.e. nested interrupts are disabled
    - SREG bit can be set to enable nested interrupts if desired
- Interrupt procedure, aka "interrupt handler"
  - Does whatever it needs to, then returns via RETI
  - The global interrupt enable bit is automatically set on RETI
  - One program instruction is always executed after RETI

Her en lille beskrivelse af hvad der sker ved et interrupt.

Interrupt-Vektorer

Her en anden oversigt der bedre forklarer assembler-koden på Interrupt Vektorerne.

Interrupt Vectors

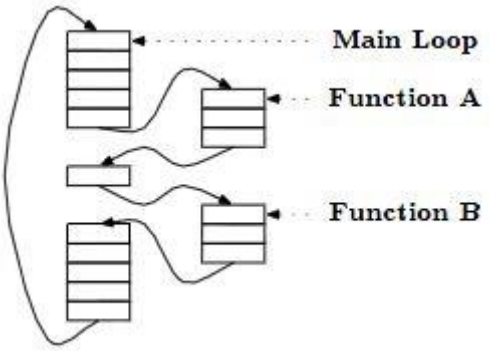
- Table in memory containing the first instruction of each interrupt handler
- Typically at program address 0

```
Address  Labels Code                 Comments
0x0000          jmp    RESET         ; Reset Handler
0x0002          jmp    EXT_INT0      ; IRQ0 Handler
0x0004          jmp    EXT_INT1      ; IRQ1 Handler
0x0006          jmp    PCINT0        ; PCINT0 Handler
0x0008          jmp    PCINT1        ; PCINT1 Handler
0x000A          jmp    PCINT2        ; PCINT2 Handler
0x000C          jmp    WDT           ; Watchdog Timer Handler
0x000E          jmp    TIM2_COMPA    ; Timer2 Compare A Handler
0x0010          jmp    TIM2_COMPB    ; Timer2 Compare B Handler
0x0012          jmp    TIM2_OVF      ; Timer2 Overflow Handler
0x0014          jmp    TIM1_CAPT     ; Timer1 Capture Handler
0x0016          jmp    TIM1_COMPA    ; Timer1 Compare A Handler
0x0018          jmp    TIM1_COMPB    ; Timer1 Compare B Handler
0x001A          jmp    TIM1_OVF      ; Timer1 Overflow Handler
0x001C          jmp    TIM0_COMPA    ; Timer0 Compare A Handler
0x001E          jmp    TIM0_COMPB    ; Timer0 Compare B Handler
```

Det skal bemærkes, at funktionerne ikke ligger ved siden af, - men ligger efter loop-koden.

Der er jo kun 1 flash-programhukommelse.

Hvis man bruger funktioner i ens program vil compileren også placere koden efter loop()-koden !!



Regler og tips for en Interrupt-rutine.

Når der bruges interrupts i et program, er der mange faldgruber og begrænsninger, der skal tages højde for.

Overholder man reglerne, kan man spares for mange unødvendige og ”hard to find” debug problemer. Her er gennemgået nogle:

En interruptsubrutine (ISR eller Funktion) kan udføres mange gange i sekundet. Fx hvis det er et timerudløst interrupt. Derfor er det vigtigt, at selve interrupt-rutinen ikke tager ret lang tid.

Pinudløste interrupts og kontaktprel:

Dvs. der bør ikke være tunge regneoperationer i en ISR. En ISR bør max tage nogle få microsekunder. ISR's skal altså holdes så korte som muligt.

Kommunikation fra en ISR til Loop()

I en ISR kan det ske, at variable, der er fælles med Loop() kun opfattes som " kopier " af de rigtige variable. Det er compileren, der på den måde vil hastighedsoptimere en ISR. Men det går jo ikke, fx hvis en ISR skal opdatere en variabel, der skal bruges i Loop().

Det forhindres ved før setup() at erklære (kvalificere) variablene som volatile (dvs. program-globale).

Eks:

```
int pin = 13;
volatile int a = 3;
volatile int Count = 0;
volatile bool state = false;
```

En anden måde at signalere fra en ISR til loop() kan ske vha. flag.

Et flag er en boolean type.

Loop skal så undersøge, om flaget er sat, - lægge det ned, - og udføre den programstump, der skal køres.

Også flag skal erklæres for **volatile**.

Opsætning af et interrupt

Før man kan benytte interrupts, skal der noget opsætning til. Ved at sætte forskellige bits i forskellige **special function registre** kan man opsætte processoren til at interrupte på forskellige events.

Interrupts skal altså enables og den tilhørende interrupt-maske skal enables, ligesom der skal skal skrives kode til det, processoren skal udføre, under et interrupt.

Pin-udløste interrupts.

Unoen kan håndtere to pinudløste interrupts. De er fast tilknyttet Arduinoens pin 2 (INT0) og pin 3, (INT1).

Og der kan vælges, om det signal der føres til en pin for at udløse et interrupt skal være low level, en rising edge, en falling edge, eller pinchange.

Et eksempel, ” the Arduino Way”:

Pin - interrupts kan sættes op ” the arduino way ” – eller man kan selv ”pille bit”!!

Her et eksempel på Arduino-sprogets opsætnings-syntax:

```
void setup()
{
    pinMode(13, OUTPUT);      // Pin 13 is output to which an LED is connected
    digitalWrite(13, LOW);    // Make pin 13 low, switch LED off
    pinMode(2, INPUT);      // Pin 2 is input to which a switch is connected =
INT0
    pinMode(3, INPUT);      // Pin 3 is input to which a switch is connected =
INT1
    attachInterrupt(0, blink1, RISING);
    attachInterrupt(1, blink2, RISING);
                              // attachInterrupt(0, INT0_handler, CHANGE);  //
}

void loop() {
    /* Nothing to do: the program jumps automatically to Interrupt Service
Routine "blink" in case of a hardware interrupt on Ardiono pin 2 = INT0 */
}

void blink1(){                // Interrupt service routine
    digitalWrite(13, HIGH);
}

void blink2(){                // Interrupt service routine
    digitalWrite(13, LOW);
}
```

Har man behov for i et programforløb at disable muligheden for interrupt, kan det ske med instruktionen;

```
detachInterrupt(0); // stopper interrupt 0!
detachInterrupt(1);
```

Arduino-funktionerne *attachInterrupt*() og *detachInterrupt*() bruges kun til pin-udløste interrupts.

Et eksempel mere:

Her et eksempel på at Interruptfunktionen ”kun” sætter et flag, en bool, og så vil loop() tjekke om dette flag er sat !!

```
volatile bool Motor_on = false;

void setup() {

  attachInterrupt(0, triggerRunMotor, RISING); // define pin-Interrupt
}
void loop() {
  if (Motor_on) {
    Motor_on = false;
    startMotor();
  }
}

void triggerRunMotor() {
  Motor_on = true;

}
void startMotor() {
  // this function may contain code that
  // requires heavy computation, or takes
  // a long time to execute
}
```

Selvgjort pin-interrupt-opsætning

Opsætningen af eksterne interrupts kan også styres ved at man selv sætter nogle bits i SFR-registre.

De SFR´s der bruges hertil er:

I External Interrupt Control Register A indstilles måden, man ønsker et interrupt udløst.

```
Eks: EICRA = 0b00000011;
Eller
bitSet (EICRA, 0 );
bitSet (EICRA, 1 );
```

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00

External Interrupt Control Register A

ISCx1	ISCx0	DESCRIPTION
0	0	Low level of INTx generates an interrupt request
0	1	Any logic change on INTx generates an interrupt request
1	0	The falling edge of INTx generates an interrupt request
1	1	The rising edge of INTx generates an interrupt request

ISC Bit Settings

Et Extern Interrupt skal enables. Det sker i External Interrupt Mask Register, kaldet EIMSK

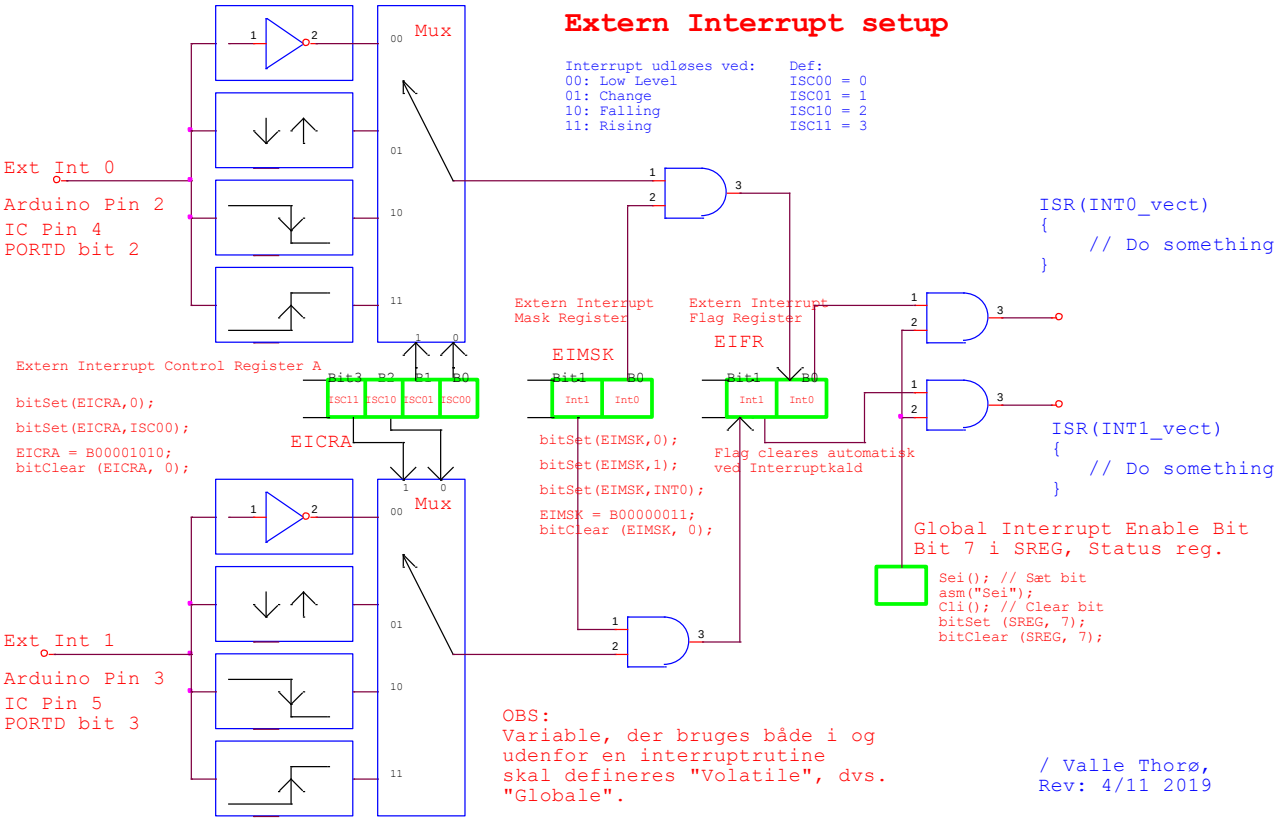
	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
EIMSK	-	-	-	-	-	-	INT1	INT0

External Interrupt Mask Register

`bitSet (EIMSK, 0 );`	
Og når et INTF flag bliver høj, udløses et interrupt. Flaget resettes automatisk når Interrupt Service Rutinen udføres.	<table><tr><th></th><th>7 bit</th><th>6 bit</th><th>5 bit</th><th>4 bit</th><th>3 bit</th><th>2 bit</th><th>1 bit</th><th>0 bit</th></tr><tr><td>EIFR</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>INTF1</td><td>INTF0</td></tr></table>*External Interrupt Flag Register*

Kilde: https://sites.google.com/site/qeewiki/books/avr-guide/external-interrupts-on-the-atmega328

Her forsøgt vist grafisk – og med eksempler – at vise, hvordan man sætter bit:



Kodeeksempler:

```
void setup(void)
{
    pinMode(2, INPUT);
    pinMode(13, OUTPUT);
```

```
    digitalWrite(2, HIGH);     // Enable pullup resistor
    sei();                     // Enable global interrupts

    bitSet(EIMSK, 0); // Enable external interrupt INT0
    bitSet(EICRA, 1); // Trigger INT0 on falling edge
                      // fra:  bitSet(x, bitPosition)
}

void loop(void)
{
 // Do something
}
                                //
ISR(INT0_vect)  // Interrupt Service Routine attached to INT0 vector

{
    digitalWrite(13, !digitalRead(13));    // Toggle LED on pin 13
}
```

Opgave !!

Registre indstilles
Der er to pin-interrupt.
Typer, falling, rising ??
Int0 og Int1 pin ??
Ej bruge delay mm i interrupt-funktioners. Udveksl data med attributten volatile

ISR, interrupt service routine. (servicefunktion)