Datorteknik TSEA82 + TSEA57
Fö7

Avbrott

LINKÖPINGS UNIVERSITET

Datorteknik Fö7 : Agenda

- Repetition subrutiner
- Avbrott
- Lab 3
- Tid för frågor

Repetition subrutiner

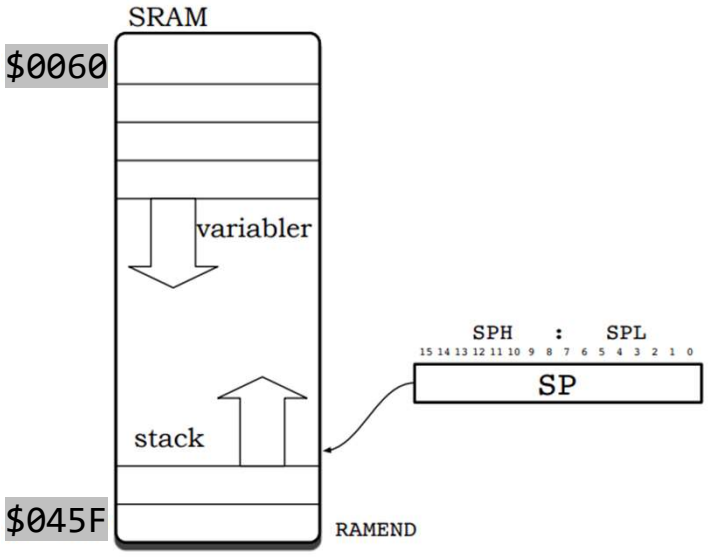
Stacken

För att anropa en subrutin används instruktionen `call` / `rcall` (Absolute Call / Relative Call). **Observera**, att man **anropar alltid** en subrutin, man **hoppar aldrig** till en subrutin.

Instruktionerna `call` / `rcall` fungerar som `jmp`, med skillnaden att återhoppsadressen sparas på en stack

Stacken är bara en del av arbetsminnet (SRAM). Vanligen så används slutet av minnet och stacken växer mot lägre adresser.

En stackpekare, SP, pekar på nästa lediga plats i stacken, och SP räknas ned / upp när något läggs dit / tas bort på stacken.



Stacken : push och pop

Instruktionerna `call` och `ret` påverkar alltså automatiskt innehållet på stacken, förändrar stackpekarens (`SP`) värde och styr programflödet genom att programräknarens (`PC`) värde läggs upp på stacken vid `call` och återhämtas vid `ret`.

Man kan även påverka stacken manuellt genom lägga dit eller hämta värden, via instruktionerna `push` och `pop`. Stackpekaren `SP` uppdateras automatiskt.

Tex:

```
push    r16     ; värdet i r16 läggs på stacken, därefter minskas SP med 1
pop     r22     ; SP ökas med 1, därefter hämtas (kopieras) värdet på stacken till r22
```


Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

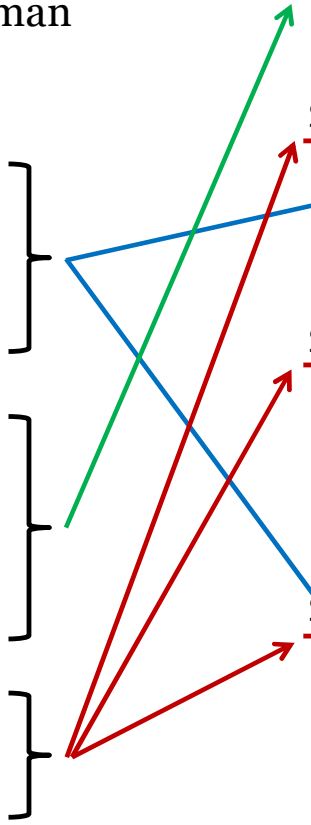
Stacken : subrutiner

Exempel: Gör en subrutin för att beräkna summan av `10` konsekutiva tal, börja med talet i `r17`. Returnera summan i `r17`.

Spara undan kontexten för de ”lokala” register som förändras i subrutinen. Observera ordningen, Det som läggs dit sist på stacken hämtas först.

För att den som läser koden lätt ska förstå vad koden gör och snabbt och enkelt ska veta vilka in- och ut-parametrar som används, skriver man lämpligen ett ”funktionshuvud” i form av en kommentar som talar om detta.

Använd samma stam-namn på alla labels i subrutinen. Det undviker namnkonflikter och blir tydlig att dessa labels tillhör subrutinen.



```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in r17
; IN: r17, starting number
; OUT: r17, sum
SUM:
    push    r16     ; save context
    push    r18
    ldi     r16,10  ; set loop index
    clr     r18     ; clear sum
SUM_LOOP:
    add     r18,r17 ; add to sum
    inc     r17
    dec     r16     ; next in loop
    brne    SUM_LOOP
    mov     r17,r18 ; set return value
SUM_EXIT:
    pop     r18     ; restore context
    pop     r16
    ret
```

Stacken : subrutiner

För att beräkna summan av de 10 talen
4+5+6+7+8+9+10+11+12+13
kan man anropa subrutinen med 4 i `r17`:
(parameteröverföring via register)

```
    ldi     r17,4
    call    sum
    ...
```

Resultatet, summan, kommer tillbaka i `r17`,
så därför sparas inte `r17` på stacken.

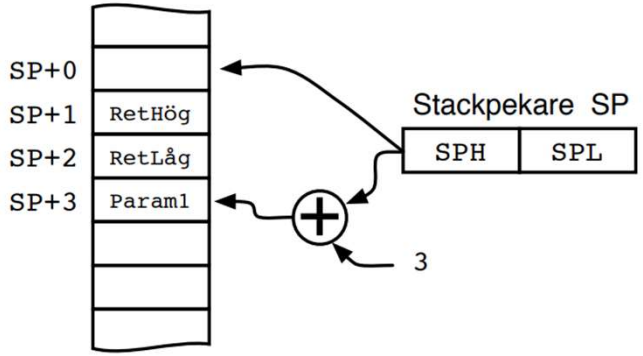
Subrutinen ska bara ha **EN** utgång, som man kan hoppa till från andra delar av subrutinen. Dvs, bara **EN** `ret`-instruktion per subrutin. Om man använder `ret`-instruktionen på flera platser för samma subrutin så måste kontexten återställas på alla dessa platser. Det blir stökigt.

```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in r17
; IN: r17, starting number
; OUT: r17, sum
SUM:
    push    r16     ; save context
    push    r18
    ldi     r16,10  ; set loop index
    clr     r18     ; clear sum
SUM_LOOP:
    add     r18,r17 ; add to sum
    inc     r17
    dec     r16     ; next in loop
    brne    SUM_LOOP
    mov     r17,r18 ; set return value
SUM_EXIT:
    pop     r18     ; restore context
    pop     r16
    ret
```

Parameter-stacken

Om man vill göra en mer generell lösning kan man använda stacken för parameteröverföring.
Här läggs parametern på toppen av stacken:

```
ldi     r25,4
push    r25
call    sum
pop     r25
...
```



Register `r25` används endast temporärt
Resultatet, summan, kommer tillbaka på toppen av stacken och kan sedan överföras till t ex `r25`.

```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in parameter stack
; IN: first pos in parameter stack
; OUT: sum in first pos in parameter stack
SUM:
    in      ZH,SPH      ; copy stack pointer
    in      ZL,SPL      ;   to Z
    push    r17         ; get parameter
    ldd     r17,Z+3     ;   from stack
    push    r16         ; save context
    push    r18
    ...
    ...         ; samma som förut
    ...
SUM_EXIT:
    pop     r18         ; restore context
    pop     r16
    std     Z+3,r17     ; store sum in
    pop     r17         ;   return stack
    ret
```

Subrutiner

Är alla lablar subrutiner?

```
    ...
TEST1:
    cpi     r16,4
    brne    TEST2
    call    SUB1
TEST2:
    cpi     r16,7
    brne    TEST3
    call    SUB2
TEST3:
    ...
```

```
; Sub SUB1
SUB1:
    ...
    ret
```

Undvik tarmar

```
    ...
; Sub SUB2
SUB2:
    cpi     r16,$0F
    breq    TO_DIR
SUB2_PORT:
    ori     r16,$07
    jmp     TO_PORT
SUB2_EXIT:
    ret
```

```
TO_DIR:
    out     DDRB,r16
    jmp     SUB2_PORT
```

```
TO_PORT:
    out     PORTB,r16
    jmp     SUB2_EXIT
```

Ha hela subrutinen i ett paket, med `ret`–instruktionen sist

```
    ...
; Sub SUB2
SUB2:
    cpi     r16,$0F
    brne    SUB2_PORT
    out     DDRB,r16
SUB2_PORT:
    ori     r16,$07
    out     PORTB,r16
SUB2_EXIT:
    ret
```

Lämna **alltid** subrutinen via `ret`-instruktionen.

Avbrott

Pollning

Huvuduppgiften för en dator är att ta indata, på något sätt bearbeta indata och producera utdata.

Ett sätt att göra det är att huvudloopen i ett program har som uppgift att om och om igen kontrollera indata/insignaler, värden i register, status för flaggor eller annan information för att sedan agera, producera utdata/utsignaler.

Principen kallas för att polla, eller pollning.

Nackdelar med pollning är att dels blir huvudloopen blir ganska processorintensiv, dels kan det ta för lång tid innan rätt sak kontrolleras, dvs reaktionstiden blir för lång, eller ta olika lång tid mellan varven i huvudloopen.

```
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
MAIN_T2:
    lds     r16,$210
    cpi     r16,7
    brne    MAIN_T3
    call    TASK2
MAIN_T3:
    sbic    PINB,3
    call    TASK3
    ...
    ...
    rjmp    MAIN_LOOP
```

Avbrott

Avbrott är ett sätt att, förstås, avbryta det som pågår och istället göra något annat. Det sker via en *avbrottsbegäran*, vilket tvingar processorn att hoppa till en särskild rutin, en *avbrottsrutin*.

När avbrottet är färdigt, återgår exekveringen till det som processorn gjorde innan avbrottet kom.

Ur huvudprogrammets synvinkel kan ett avbrott komma precis när som helst, som en blixt från klar himmel.

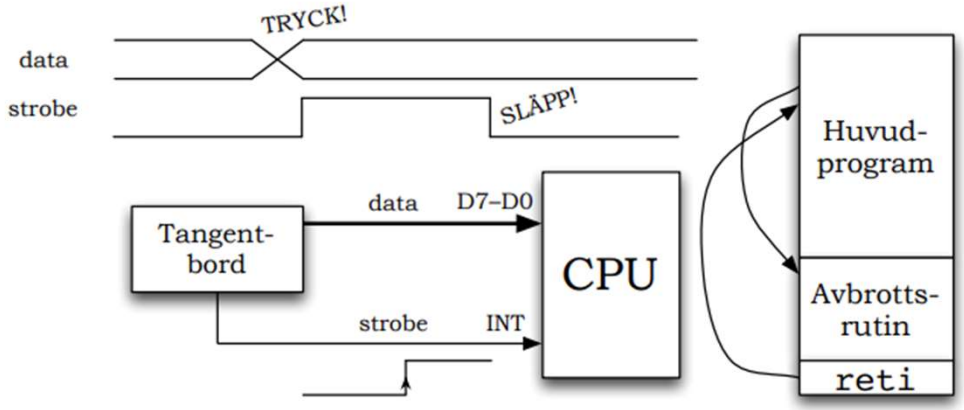
Det medför att avbrott behöver hanteras något annorlunda jämfört med subrutiner. Subrutiner är något som programmet har kontroll över när dom händer, men det gäller inte avbrott.

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Avbrott

Det finns ett antal olika händelser som kan orsaka avbrott, så kallade *avbrottskällor*. T ex en flanken på en yttre signal.



Det är ett så kallat *externt avbrott*, dvs det har sin källa utanför processorn. Det finns även *interna avbrott*, dvs någon händelse inuti processorn, som kan orsaka avbrott.

```
     .org 0x0000
     jmp      MAIN
     .org 0x0002
     jmp      EXT_INT0
MAIN:
     ...
MAIN_LOOP:
     cpi      r16,4
     brne     MAIN_T2
     call     TASK1
     ...
     ...
     rjmp     MAIN_LOOP
```

```
EXT_INT0:
     ...      ; save context
     ...
     ...      ; restore context
     reti
```

Avbrottskällor / avbrottsvektorer

Table 11-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Avbrott : Vad händer?

1. När avbrottet triggats kommer den globala avbrottsflaggan I i statusregistret SREG att nollställas, för att förhindra ytterligare avbrott.

2. Återhoppsadressen, dvs programräknaren PC's nuvarande värde, sparas på stacken.

3. Därefter styrs exekveringen till avbrottets avbrottsvektor i vektortabellen, dvs PC=avbrottsvektor. Detta är hårdkodat, inbyggt i hårdvaran, och kan inte ändras.

4. Från avbrottsvektorn görs ett vanligt hopp till själva avbrottsrutinen.

```
     .org 0x0000
     jmp     MAIN
     .org 0x0002
     jmp     EXT_INT0
MAIN:
     ...
MAIN_LOOP:
     cpi     r16,4
     brne    MAIN_T2
     call    TASK1
     ...
     ...
     rjmp    MAIN_LOOP
```

```
EXT_INT0:
     ...     ; save context
     ...
     ...     ; restore context
     reti
```

Avbrott : Återhopp, vad händer?

När avbrottsrutinen gjort sitt arbete görs återhopp med instruktionen `reti`. Den instruktionen 1-ställer den globala avbrottsflaggan `I` i statusregistret `SREG`, vilket möjliggör nya avbrott.

Samtidigt sker själva återhoppet, genom att programräknaren PC återtar den tidigare sparade återhoppsadressen från stacken.

Återhopp från en avbrottsrutin måste alltså göras med `reti` (inte `ret`, som för subrutiner). Annars 1-ställs inte den globala avbrottsflaggan `I`, och det kan inte ske fler avbrott.

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Instruktion	Innebörd
reti	SP:=SP+2, PC:=M(SP), 1->I

Avbrott : Spara inre tillstånd

Eftersom avbrottet kan komma när som helst, så kan det tänkas att processorn har information i statusregistret, t ex från en jämförelse innan avbrottet, som man inte vill förlora.

Detta inre tillstånd, dvs statusregistret, behöver sålunda sparas, tills efter avbrottet.

```
EXT_INT0:
    push    r16         ; save
    in      r16,SREG    ; .. inner
    push    r16         ; .. context
    ...
    pop     r16         ; restore
    out     SREG,r16    ; .. inner
    pop     r16         ; .. context
    reti
```

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Instruktion	Innebörd
reti	SP:=SP+2, PC:=M(SP), 1->I

Avbrott : Vad behöver initieras?

```
        .org $0000
    jmp     RESET           ; Reset handler
        .org INT0addr
    jmp     EXT_INT0        ; INT0 Handler
    ...
        .org INT_VECTORS_SIZE
RESET:
    ldi     r16,HIGH(RAMEND)
    out     SPH,r16
    ldi     r16,LOW(RAMEND)
    out     SPL,r16
    ...
    call    INIT_INT0
    sei
MAIN_LOOP:
    ...
    jmp     MAIN_LOOP
```

Adresserna i vektortabellen har fördeklarerade namn, som kan användas istf konstanter.

INT_VECTORS_SIZE = $2A

Initiera stackpekaren SP först, den kommer att behövas ganska omgående, till subrutiner och avbrott.

Initiera specifika avbrott.
Möjliggör avbrott globalt.

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Instruktion	Innebörd
sei	1->I
cli	0->I

Avbrott : Initiera det specifika avbrottet

Beroende på vilket/vilka avbrott man vill använda så måste den respektive specifika avbrottsfunktionaliteten konfigureras/initieras.
T ex för det externa avbrottet `INT0`:

Avbrottet möjliggörs via Global Interrupt Control Register:

7	6	5	4	3	2	1	0	
INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
R/W	R/W	R/W	R	R	R	R/W	R/W	

Abrottets flank konfigureras i MCU Control Register:

7	6	5	4	3	2	1	0	
SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

När avbrottet inträffar noteras *avbrottsbegäran* i registret General Interrupt Flag register:

7	6	5	4	3	2	1	0	
INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
R/W	R/W	R/W	R	R	R	R	R	

Exakt vad som behöver initieras, beror på vilket avbrott som ska användas. Kontrollera i processorns manual.

```
     .org 0x0000
     jmp       RESET
     .org INT0addr
     jmp       EXT_INT0
RESET:
     ...

INIT_INT0:
     ldi  r16,(1<<INT0)
     out  GICR,r16
     ldi  r16,(1<<ISC01)|(1<<ISC00)
     out  MCUCR,r16
     ret
```

```
EXT_INT0:
     ...       ; save context
     ...
     ...       ; restore context
     reti
```

Avbrott : Vad händer *egentligen*?

1. När avbrottshändelsen inträffat noteras en *avbrottsbegäran* i det specifika avbrottets avbrottsflagga
2. Om den globala avbrottsflaggan `I` (i statusregistret `SREG`) är `1`-ställd:
   2.1 Om det specifika avbrottets funktion är aktiverat:
   - `0`-ställ den globala avbrottsflaggan `I` i `SREG`
   - `0`-ställ avbrottsbegäran för det specifika avbrottet
   - spara återhoppsadressen på stacken
   - styr exekveringen till det specifika avbrottets avbrottsvektor
   - utför ett hopp till avbrottsrutinen

   2.2 Om det specifika avbrottets funktion är inaktiverat:
   - fortsätt som vanligt, dvs det blir inget avbrott
3. Om den globala avbrottsflagga `I` (i statusregistret `SREG`) är `0`-ställd:
   - fortsätt som vanligt, dvs det blir inget avbrott
4. Efter färdig avbrottsrutin, sker återhopp med `reti`, varpå:
   - den globala avbrottsflaggan `1`-ställs och exekveringen fortsätter vid återhoppsadressen
5. Om det inkommit avbrottsbegäran under tiden som avbrottsrutinen kör, dvs det finns `1`-ställda avbrottsflaggor för något/några avbrott (dvs *avbrottsbegäran*), så kommer dessa avbrott nu att utföras enligt ovan.

```
    .org 0x0000
    jmp     MAIN
    .org 0x0002
    jmp     EXT_INT0
MAIN:
    ...
MAIN_LOOP:
    cpi     r16,4
    brne    MAIN_T2
    call    TASK1
    ...
    ...
    rjmp    MAIN_LOOP
```

```
EXT_INT0:
    ...     ; save context
    ...
    ...     ; restore context
    reti
```

Avbrott : Exempel för INT0 och INT1

Konfigurera de externa avbrotten `INT0` och `INT1` för negativ flank.

```
        .org $0000
    jmp     MAIN            ; Reset handler
        .org INT0addr
    jmp     EXT_INT0        ; INT0 Handler
        .org INT1addr
    jmp     EXT_INT1        ; INT1 Handler
        .org  INT_VECTORS_SIZE

MAIN:
    ldi   r16,HIGH(RAMEND)      ; init SP
    out   SPH,r16
    ldi   r16,LOW(RAMEND)
    out   SPL,r16
    ; Configure flanks
    ldi   r16,(1<<ISC01)|(0<<ISC00)|
              (1<<ISC11)|(0<<ISC10)
    out   MCUCR,r16
    ; Enable specific interrupts
    ldi   r16,(1<<INT1)|(1<<INT0)
    out   GICR,r16
    ; Enable interrupts globally
    sei
MAIN_WAIT:
    jmp   MAIN_WAIT
```

```
    ; ---- ISR Interrupt Service Routines

EXT_INT0:
    push  r16
    in    r16,SREG
    push  r16
    ...
    pop   r16
    out   SREG,r16
    pop   r16
    reti

EXT_INT1:
    push  r16
    in    r16,SREG
    push  r16
    ...
    pop   r16
    out   SREG,r16
    pop   r16
    reti
```

Interna avbrott

Table 11-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

Externa avbrott, dvs en yttre signal via flank eller nivå orsakar ett avbrott i processorn.

Interna avbrott. Någon händelse inne i processorn orsakar avbrott. T ex en timer har uppnått ett visst värde, en seriell överföring är slutför, en A/D-omvandling är klar m m.

Timer-avbrott

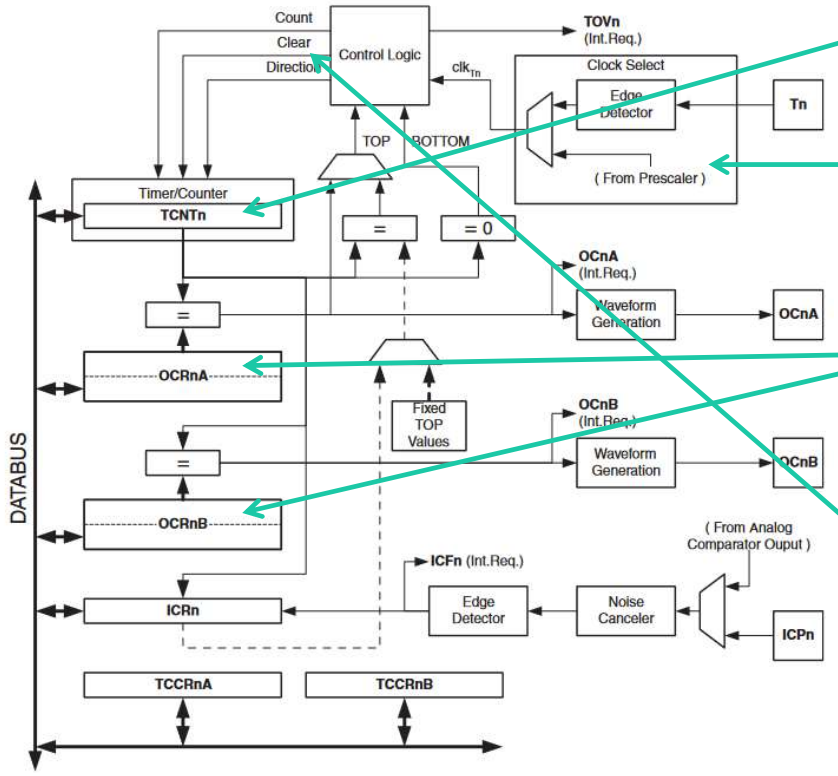
Processorn har tre olika hårdvaru-timers som kan arbeta parallellt och oberoende av varandra, samtidigt som den övriga programkoden gör nåt annat.

Med en timer kan man mäta tidsperioder / pulslängder och få avbrott vid pulsens flanker. Man kan även generera pulslängder /tidsperioder och få avbrott när det gått en viss tid.

En timer är alltså ett bra sätt att uppnå god timing, jämfört med att använda vänteloopar i ett program, eftersom programflödet kan ta olika lång tid i en huvudloop beroende på vad som händer i programmet.

Timer-avbrott

Figure 16-1. 16-bit Timer/Counter Block Diagram[1]



Timern är egentligen bara en räknare (`TCNTn`) vars värde räknas upp/ned för varje räknarpuls.

Räknarpulsen utgår (vanligen) från processorn klocka och kan skalas ned med en prescaler.

Räknarens värde kan jämföras med innehållet i jämförelseregister (`OCRnA`, `OCRnB`), och när räknaren uppnått samma värde kan t ex ett avbrott genereras.

Samtidigt som räknaren uppnått jämförelsevärdet nollställs räknaren och den börjar om att räkna på nytt.

Allting sker parallellt med att det övriga programmet löper på som vanligt.

Labb 3

...

Tid för Frågor

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET