Datorteknik TSEA82 + TSEA57 Fö8

Preprocessor & macro

LINKÖPINGS UNIVERSITET

1

Datorteknik Fö8 : Agenda

- Repetition avbrott
- Preprocessor & macro
- Lab 3, tips
- Tid för frågor

2

Repetition avbrott

LINKÖPINGS UNIVERSITET

3

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
```

LINKÖPINGS UNIVERSITET

4

Avbrottskällor / avbrottsvektorer

Table 9-6. Reset and Interrupt Vectors in ATmega328P

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
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

```
    .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
```

... det finns ytterligare några avbrottsvektorer

LINKÖPINGS UNIVERSITET

5

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
```

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

```
    .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
```

LINKÖPINGS UNIVERSITET

6

Avbrott : Vad behöver initieras?

```
        .org $0000
    jmp     RESET          ; Reset handler
        .org INT0addr      <-- Adresserna i vektortabellen har fördeklarerade namn, som kan användas istf konstanter.
    jmp     EXT_INT0       ; INT0 Handler
    ...
        .org INT_VECTORS_SIZE  <-- INT_VECTORS_SIZE = $34
RESET:
    ldi     r16,HIGH(RAMEND)   <-- Initiera stackpekaren SP först, den kommer att behövas ganska omgående, till subrutiner och avbrott.
    out     SPH,r16
    ldi     r16,LOW(RAMEND)
    out     SPL,r16
    ...
    call    INIT_INT0          <-- Initiera specifika avbrott.
    sei                        <-- Möjliggör avbrott globalt.
MAIN_LOOP:
    ...
    jmp     MAIN_LOOP
```

```
    .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

LINKÖPINGS UNIVERSITET

7

Preprocessor & macro

LINKÖPINGS UNIVERSITET

8

Preprocessor och kompilering

Programkod, i detta fall assembler-kod, passerar två steg innan den kan programmeras i mikrokontrollern.

1. Preprocessor-steget
   - Konstanter ersätts med sina egentliga värden
   - Lablar (symboliska adresser) beräknas
   - Enklare beräkningar utförs
   - Macron expanderas till motsvarande assembler
   - Enklare operation (HIGH, LOW, <<) utförs
   - m m
2. Kompilerings-steget
   - Med alla konstanter, definitioner och övriga beräkningar gjorda kan koden relativt enkelt direkt översättas, rad för rad, till mostvarande maskinkod.

Programkod .asm

```
     .org 0
     jmp  MAIN

     .equ N = 3
     .def tmp = r16
     .def LV = r17
     ...
MAIN:
     ldi  tmp,HIGH
     out  SPH,r16
     ldi  tmp,LOW
     out  SPL,r16
     ...
     ldi  LV,7*N
     ...
```

Maskinkod .hex

```
:1001000049726F6E
:100110006D616964
:10012000656E7275
:100130006C657A20
:
:
:
:
:
:
```


Preprocessor och kompilering

Preprocessorn förstår ett antal direktiv, som alltså inte är egentlig assemblerkod utan just direktiv för att definiera, strukturera och tolka den övriga programkoden.

Preprocessordirektiven finns för att man enklare och tydligare ska kunna skriva och strukturera sin programkod.

Dom är egentligen inte nödvändiga för att åstadkomma det som programmet ska göra, men underlättar kodandet och gör det tydligare och lättare att läsa det resulterande programmet.

Direktiv	Namn	Betydelse
`.org`	*origin*	Skriv här (adress)
`.byte`	*byte*	Reservera byte i SRAM
`.dseg`	*data segment*	Följande gäller SRAM
`.cseg`	*code segment*	Följande gäller programminnet
`.eseg`	*extra segment*	Följande gäller EEPROM
`.def`	*define*	Döp register till namn
`.equ`	*equate*	Döp konstant
`.db`	*define byte*	Skriv följande *byte* (8-bit) i minnet
`.dw`	*define word*	Skriv följande *word* (16-bit) i minnet
`.macro`	*macro*	"copy-paste" av följande
`.endmacro`	*endmacro*	...avsluta ett macro
`<< n`	*shift left*	vänsterskift n bitar
`&, |, ^`	*logical AND, OR, XOR*	bitvis OCH, ELLER, XOR
`+, -, *, /`	*arithmetic*	som förväntat
`HIGH, LOW`	*high low*	ger höga resp låga delen av följande uttryck

Kompilering med grammatik

Kompileringen utförs typiskt med två generella verktyg.

En *scanner*, som returnerar igenkänningsbara delar av programkoden, såsom instruktioner, tal, kommatecken, register m m. Dessa delar kallas vanligen för tokens.

En *parser*, som tar dessa tokens och mönstermatchar ordningen dom kommer i, mot en definierad grammatik, ofta skriven i BNF (Backus Naur Form).

Om ett matchande mönster hittas översätts delarna till motsvarande maskinkod, dvs koden genereras.

Om inget matchande mönster hittas, så förekommer ett syntax-fel och ett felmeddelande skriv ut.

BNF (Backus Naur Form)

```
asmprog  : asmprog asmrow
         | asmrow

asmrow   : instr
         | instr reg
         | instr reg ’,’ reg
         | instr reg ’,’ const
         | ...

instr    : ”ldi”
         | ”mov”
         | ”call”
         | ...

reg      : ”r0”
         | ...
         | ”r31”

const    : NUMBER
         | expression
         ...
```


Preprocessordirektiv

.org sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet. Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

.cseg anger att efterföljande kod ska hamna i programminnet.

.db definierar tabellvärden i programminnet (Flash)

```
    .cseg                   ; default
    .org $0000
    jmp  START
    ;
    ;    avbrottsvektorer
    ;
    .org INT_VECTORS_SIZE   ; 52
TAB: .db  1, 2, 3, 4
START:
    ;    programstart
```

Preprocessordirektiv

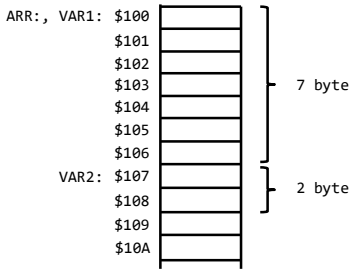
.org sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet.
Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

.dseg anger att efterföljande definitioner ska använda SRAM (dataminnet).

.byte reserverar ett antal byte för variabler, bara det. Det går inte att tilldela värden till dessa variabler här.

```
    .dseg
    .org $100     ; adress $100 i SRAM
ARR:              ; ARR=$100, handtag till struct nedan
VAR1:    .byte 7  ; VAR1, adressen till 0-te byten av dessa 7
VAR2:    .byte 2  ; VAR2, adressem till första lediga efter VAR1

    .cseg
    ; till programminnet igen
    lds  r16,VAR1
    lds  r17,VAR2
    ...
```



Preprocessordirektiv

.org sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet.
Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

Vanligt misstag. Varför blir det här galet?

```
TAB:    .org $200
        .db 2, 3, 4, 5
```

Det skapas en tabell, som hamnar på adress `$200`, men var hamnar `TAB`?
Någonstans dessförinnan, men inte så att man kan använda `TAB` för att peka på tabellen.

Dvs, gör så här:

```
        .org $200
TAB:
        .db 2, 3, 4, 5
```

Preprocessordirektiv

Följande visar på vilka möjligheter som preprocessorn ger oss när vi ska skriva kod.

Alla kodrader ger samtliga exakt samma resultat efter att preprocessorn gjort sitt.

```
ldi  r16,65
ldi  r16,$41
ldi  r16,0x41
ldi  r16,0b01000001
ldi  r16,(1<<6)|(1<<0)
ldi  r16,0xF1&0x4F
ldi  r16,'A'
ldi  r16,8*8+1
ldi  r16,HIGH($4122)
ldi  r16,LOW($2241)
ldi  r16,HIGH(16674)
ldi  r16,LOW(8769)
```

```
ldi  r16,0x41
```

Även följande ger samma resultat.

```
.equ N = $40
.def tmp = r16

ldi  tmp,N+1
ldi  tmp,N|(1<<0)
ldi  tmp,HIGH((N+1)<<8)
ldi  tmp,LOW(N|1)
```


Macron

Macron definierar ett kodstycke som ska kopieras in i koden. Det är inte detsamma som en subrutin, utan fungerar snarare som en ”stämpel” som preprocessorn använder när den genererar kod.

```
    .macro  PUSHZ
            push ZH
            push ZL
    .endmacro

    .macro  POPZ
            pop ZL
            pop ZH
    .endmacro
    ...
SUB:
    PUSHZ
    ...
    POPZ
    ret
```

```
SUB:
    push ZH
    push ZL
    ...
    pop ZL
    pop ZH
    ret
```

Macron

Macron kan även definieras med argument, vilket gör det lite mer användbart och dynamiskt.
Man skulle t ex kunna definiera den saknade instruktionen `ADDI`:

```
.macro  ADDI        ; macro med argumenten @0 och @1
        subi @0,-@1
.endmacro

...

ADDI    r20,3       ; r20 = r20 + 3
...
subi    r20,-3      ; r20 = r20 – (-3)
```

```
subi    r20,-3
...
subi    r20,-3
```


Macron

Macron ska dock användas sparsamt och ha väl valda namn. Annars blir koden snabbt obegriplig och svårläst, eftersom man egentligen skapar nya ord i grammatiken som inte tillhör språket från början. Dvs, den oinvigde måste själv göra översättningar av alla macron för att förstå koden.
Man får dessutom inte se vad macrot gör vid simulering, utan det bara utförs.

Utan att ha alla tidigare macro-definitioner i huvudet blir det svårt att veta vad följande kod gör:

```
.macro  ...
.endmacro
...
LAST    r22
MIX     r17,r22
PUT     r17,4
BOX
LAST    Z+
...
```

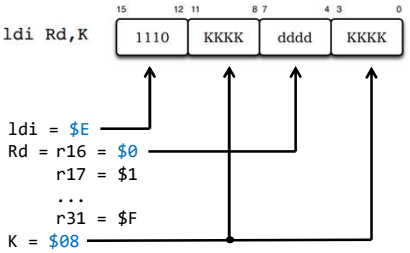
Grundregel: Undvik macron om det inte är en *jättebra* idé och har en entydig och lättolkad funktion.
Använd macron sparsamt.

Kompilering

Hela processen görs i två steg, med en s k *två-pass-assembler*:
1. Programkoden analyseras, symboliska adresser (labels) och konstanter ersätts med faktiska värden.
2. Med den informationen kan sedan assemblerinstruktioner översättas till hexadecimala tal, dvs maskinkod.

T ex programraden
```
START:  ldi r16,HIGH(RAMEND)
```

Preprocessorn ersätter RAMEND:
```
START:  ldi r16,HIGH($08FF)
```
Preprocessorn använder HIGH på $08FF:
```
START:  ldi r16,$08
```
Kompilatorn sätter adresser:
```
$0044:  ldi r16,$08
```
Kompilatorn genererar hexadecimala tal, maskinkod:
```
$0044:  $E008
```



Kompilering (överkurs)

Ett större exempel:

```
                                Adress  Hex
    .org    0x0000
    jmp     MAIN                0000    940C
                                0002    0034

    .org    INT_VECTORS_SIZE
MAIN:
    ldi     r16, HIGH(RAMEND)   0068    E008
    out     SPH, r16            006A    BF0E
    ldi     r16, LOW(RAMEND)    006C    EF0F
    out     SPL, r16            006E    BF0D
    rcall   INIT                0070    D001
END:
    rjmp    END                 0072    CFFF
INIT:
    ldi     r18,0xFF            0074    EF2F
    out     DDRB,r18            0076    B924
    ret                         0078    9508
```

Intel-HEX
```
:0200000200000FC
:0400000000C9434002 8
:1000680008E00EBF0FEF0DBF01D0FFCF2FEF24B96F
:0200780000895E9
:00000001FF
```

Det färdigkompilerade resultatet sparas i en s k Intel-Hex-fil, som följer en viss standard för hur informationen ska lagras.

Det gröna är adressangivelser.

Det blå är ”nyttolasten”, dvs själva maskinkoden.

Labb 3

tips

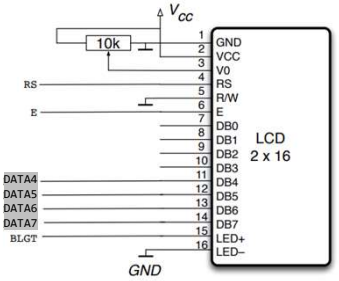
LINKÖPINGS UNIVERSITET

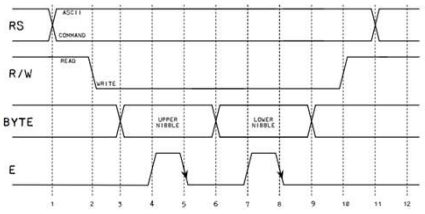
21

Lab3 : tips, LCD

```
; Sub LCD_WRITE8
; Write 8 bits to LCD
; IN: r16, 8 bits
LCD_WRITE8:
     * anropa LCD_WRITE4
     * byt plats [7..4]<->[3..0]
     * anropa LCD_WRITE4
     * byt plats [7..4]<->[3..0]
     * delay
     ret
```

```
; Sub LCD_WRITE4
; Write 4 bits to LCD
; IN: r16, 8 bits (4 high bits used)
LCD_WRITE8:
     * r16 -> PORT
     * 1 -> E
     * 4 st nop
     * 0 -> E
     ret
```






```
; Sub LCD_ASCII
; Send ASCII char to LCD
; IN: r16, ASCII char
LCD_ASCII:
     * 1 -> RS
     * anropa LCD_WRITE8
     ret
```

```
; Sub LCD_COMMAND
; Send command to LCD
; IN: r16, command
LCD_COMMAND:
     * 0 -> RS
     * anropa LCD_WRITE8
     ret
```

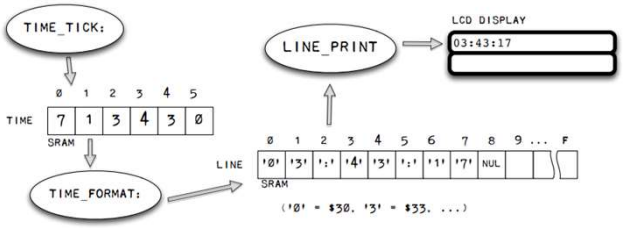
LINKÖPINGS UNIVERSITET

22

Lab3 : tips, TIME

En 23:59:59-klocka med särbehandling av timmar

```
; Sub TIME_TICK
; Increase time by one second
TIME_TICK:
     * entals-sekunder++
     * om entals-sekunder < 10 : TIME_TICK_EXIT
     * nollställ entals-sekunder
     * tiotals-sekunder++
     * om tiotals-sekunder < 6 : TIME_TICK_EXIT
     * nollställ tiotals-sekunder
     * entals-minuter++
     ...
     * entals-timmar++
     * om entals-timmar = 4 och tiotal-timmar = 2 :
       nollställ båda : TIME_TICK_EXIT
     * om entals-timmar < 10 : TIME_TICK_EXIT
     * nollställ entals-timmar
     * tiotals-timmar++
TIME_TICK_EXIT:
     ret
```



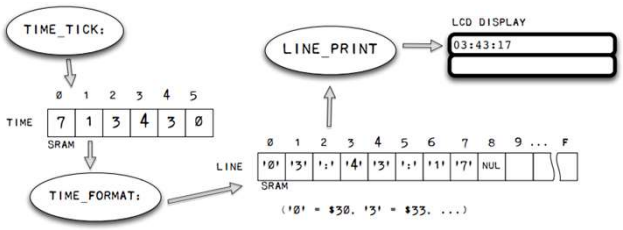
Övergångarna 13:59:59 -> 14:00:00 samt 23:59:59 -> 00:00:00 måste särbehandlas

LINKÖPINGS UNIVERSITET

Lab3 : tips, TIME

En 59:59:59-klocka med efterjustering av timmar

```
; Sub TIME_TICK
; Increase time by one second
TIME_TICK:
     * initiera pekare
TIME_TICK_LOOP:
     * entals-del++
     * om entals-del < 10 : TIME_TICK_ADJUST
     * nollställ entals-del
     * tiotals-del++
     * om tiotals-del < 6 : TIME_TICK_ADJUST
     * nollställ tiotals-del
     rjmp TIME_TICK_LOOP
TIME_TICK_ADJUST:
     * om entals-timmar = 4 och tiotal-timmar = 2 :
       nollställ båda : TIME_TICK_EXIT
TIME_TICK_EXIT:
     ret
```



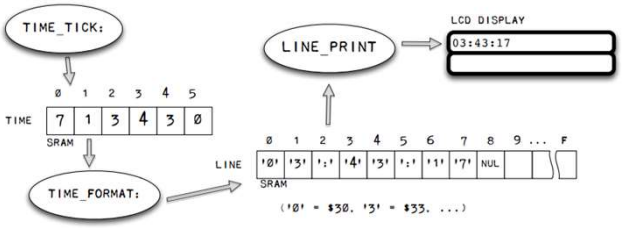
Teoretisk kan är klockan gjord för att räkna upp till 59:59:59, men kommer aldrig så långt då den nollställs när den blir 24:00:00 -> 00:00:00

LINKÖPINGS UNIVERSITET

Lab3 : tips, TIME

Gör om BCD-data i TIME till ASCII i LINE

```
; Sub TIME_FORMAT
; Time data to ASCII with ':'
TIME_FORMAT:
     * initiera pekare
TIME_FORMAT_LOOP: ; 3 varv
     * läs --TIME
     * gör om till ASCII
     * skriv LINE++
     * läs --TIME
     * gör om till ASCII
     * skriv LINE++
     * skriv ':' till LINE++
     brne TIME_FORMAT_LOOP
     * skriv NULL-tecken efter sista siffran i LINE
TIME_TICK_EXIT:
     ret
```



Pekare med pre-dekrement och post-inkrement är **MYCKET** användbart i den här labben.

Lab3 : tips, MAIN

```
LINE_PRINT:
     * skriv ut tecken för tecken från LINE
       mha tidigare LCD-funktioner
```

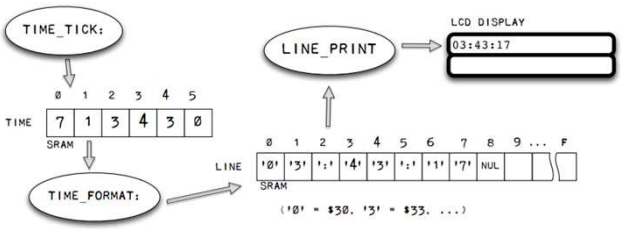
Testa först med bara ett huvudprogram:

```
MAIN:
     call TIME_TICK     ; increment time
     call TIME_FORMAT
     call LINE_PRINT
     call WAIT_1SEC     ; wait one second
     jmp  MAIN
```

Inför sedan avbrott och huvudprogram enligt:

```
ISR:
     call TIME_TICK
     reti

MAIN:
     call TIME_FORMAT
     call LINE_PRINT
     jmp  MAIN
```



Pekare med pre-dekrement och post-inkrement är **MYCKET** användbart i den här labben.

Tid för Frågor

LINKÖPINGS UNIVERSITET

27

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET

28