

Datorteknik TSEA82 + TSEA57

Fö8

Preprocessor & macro

Datorteknik Fö8 : Agenda

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

Repetition avbrott

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 blixtnärvaro 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
```

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

```

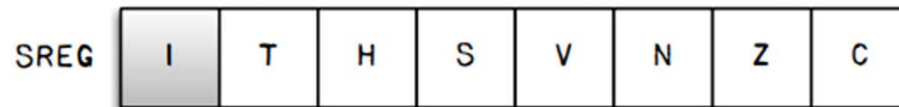
```

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

```

Avbrott : 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
```

Avbrott : Vad behöver initieras?

```

.org $0000
jmp     RESET      ; Reset handler
.org INT0addr      ← ; Adresserna i vektortabellen har
jmp     EXT_INT0    ; INT0 Handler fördeklarerade namn, som kan
...              användas istf konstanter.
.org INT_VECTORS_SIZE ← INT_VECTORS_SIZE = $34

RESET:
ldi     r16,HIGH(RAMEND) ← Initiera stackpekaren SP först, den
out     SPH,r16          kommer att behövas ganska
ldi     r16,LOW(RAMEND)  omgående, till subrutiner och
out     SPL,r16          avbrott.
...
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

```

Preprocessor & macro

Preprocessor och kompilering

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

1. Preprocessor-steget

- Konstantsymboler 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 motsvarande maskinkod.

Programkod .asm

```
.org 0
jmp MAIN

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



Maskinkod .hex

```
0000:
    jmp 0004

0004:
    ldi r16,$04
    out $3E,r16
    ldi r16,$5F
    out $3D,r16
    ...
    ldi r17,21
    ...
```

```
: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
<code>.org</code>	<i>origin</i>	Skriv här (adress)
<code>.byte</code>	<i>byte</i>	Reservera byte i SRAM
<code>.dseg</code>	<i>data segment</i>	Följande gäller SRAM
<code>.cseg</code>	<i>code segment</i>	Följande gäller programminnet
<code>.eseg</code>	<i>extra segment</i>	Följande gäller EEPROM
<code>.def</code>	<i>define</i>	Döp register till namn
<code>.equ</code>	<i>equate</i>	Döp konstant
<code>.db</code>	<i>define byte</i>	Skriv följande <i>byte</i> (8-bit) i minnet
<code>.dw</code>	<i>define word</i>	Skriv följande <i>word</i> (16-bit) i minnet
<code>.macro</code>	<i>macro</i>	"copy-paste" av följande
<code>.endmacro</code>	<i>endmacro</i>	...avsluta ett macro
<code><< n</code>	<i>shift left</i>	vänsterskift <i>n</i> bitar
<code>&, , ^</code>	<i>logical AND, OR, XOR</i>	bitvis OCH, ELLER, XOR
<code>+, -, *, /</code>	<i>arithmetic</i>	som förväntat
<code>HIGH, LOW</code>	<i>high low</i>	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
.org ADCCaddr        ; 0x1C
jmp AD_INT

.org INT_VECTORS_SIZE ; 42
TAB: .db 1, 2, 3, 4
START:
;    programstart
```

Preprocessordirektiv

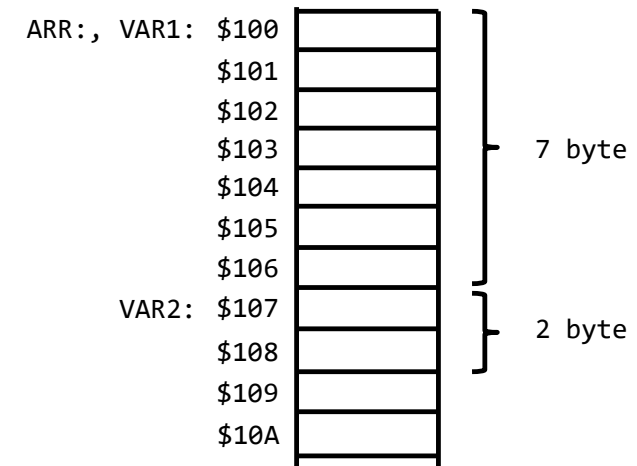
.org sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet. Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

.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, adressen till första lediga efter VAR1

.cseg
; till programminnet igen
lds r16,VAR1 ; VAR1=$100
lds r17,VAR2 ; VAR2=$107
...
```



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

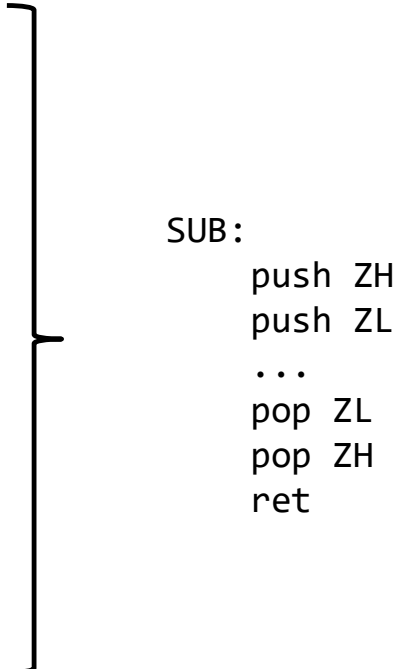
ldi tmp,N+1
ldi tmp,N|(1<<0)
ldi tmp,HIGH((N+1)<<8)
ldi tmp,LOW(N|1)
```

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
```

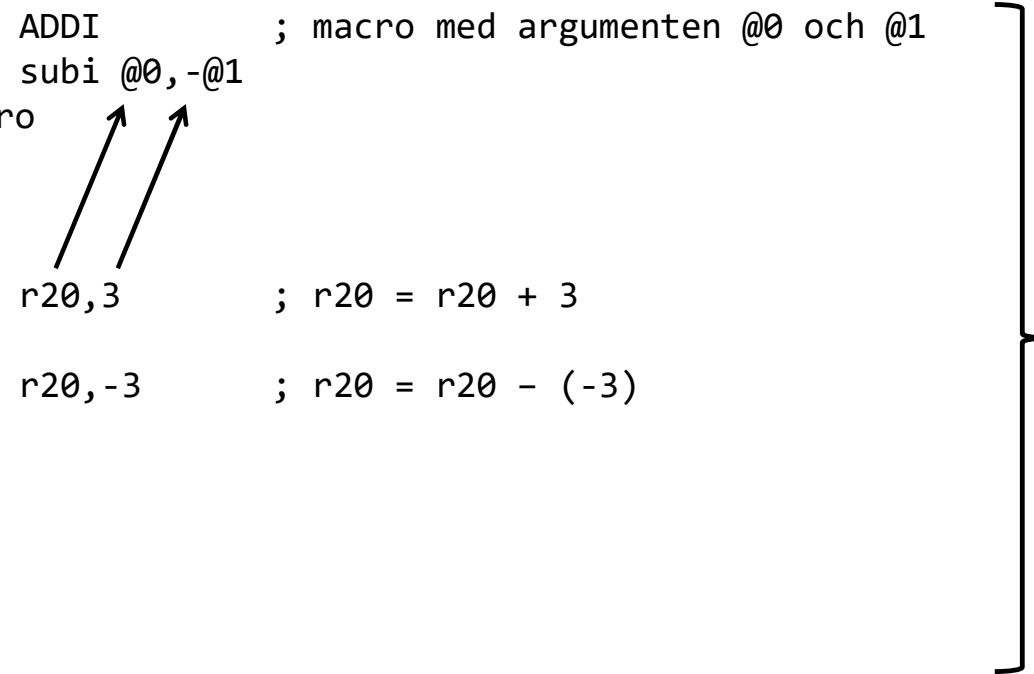

Macron

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.

Text programraden

```
START: ldi r16,HIGH(RAMEND)
```

Preprocessorn ersätter RAMEND:

```
START: ldi r16,HIGH($045F)
```

Preprocessorn använder HIGH på \$045F:

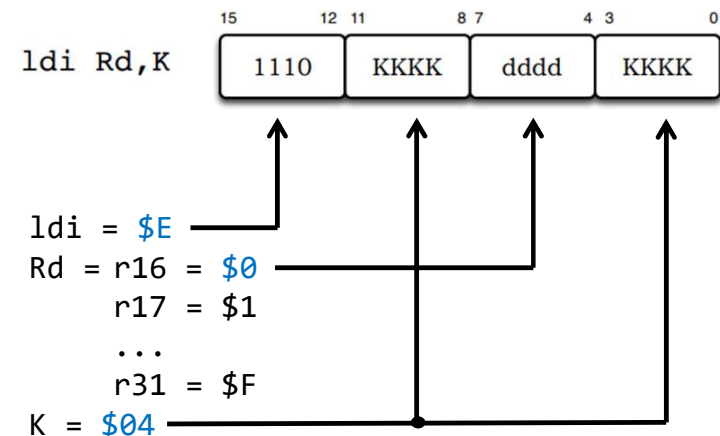
```
START: ldi r16,$04
```

Kompilatorn sätter adresser:

```
$0044: ldi r16,$04
```

Kompilatorn genererar hexadecimala tal, maskinkod:

```
$0044: $E004
```



Kompilering (översikt)

Ett större exempel:

		Adress	Hex
<code>.org</code>	<code>0x0000</code>		
<code>jmp</code>	MAIN	0000	940C
		0002	0034
<code>.org</code>	<code>INT_VECTORS_SIZE</code>		
MAIN:			
<code>ldi</code>	<code>r16, HIGH(RAMEND)</code>	0068	E008
<code>out</code>	<code>SPH, r16</code>	006A	BF0E
<code>ldi</code>	<code>r16, LOW(RAMEND)</code>	006C	EF0F
<code>out</code>	<code>SPL, r16</code>	006E	BF0D
<code>rcall</code>	INIT	0070	D001
END:			
<code>rjmp</code>	END	0072	CFFF
INIT:			
<code>ldi</code>	<code>r18, 0xFF</code>	0074	EF2F
<code>out</code>	<code>DDRB, r18</code>	0076	B924
<code>ret</code>		0078	9508

Intel-HEX

```
:0200000020000FC
:040000000C94340028
:1000680008E00EBF0FEF0DBF01D0FFCF2FEF24B96F
:020078000895E9
:00000001FF
```

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

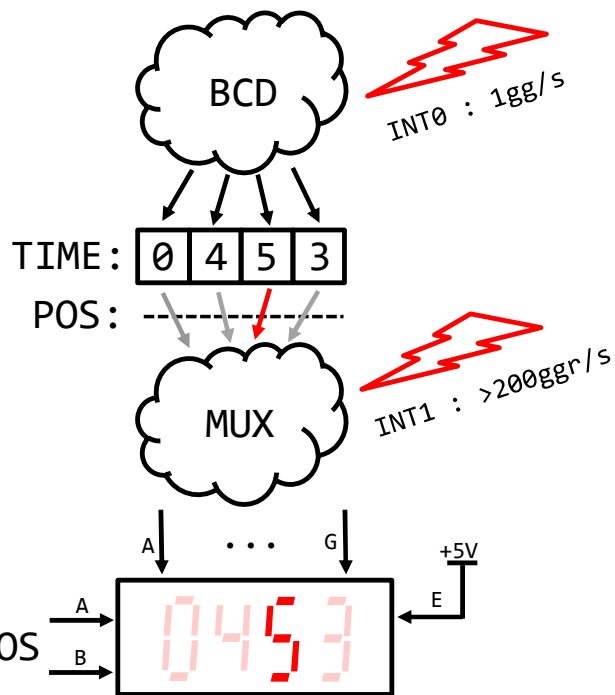
Det gröna är adressangivelser.

Det blå är ”nyttolasten”, dvs själva maskinkoden.

Labb 3

...

Labb3 : tips/"krav"



Datastruktur

```
; --- Memory layout in SRAM
        .dseg
        .org    SRAM_START
TIME:    .byte   4           ; time in BCD format
POS:     .byte   1           ; 7-seg position

; --- Table layout in FLASH
        .cseg
SEGTAB:  .db 0x3F, ...
```

Använd pekare!!

Z-pekaren för att läsa ur tabeller i programminnet.

X- el. Y-pekaren för att läsa ur tabeller (TIME) i SRAM (arbetsminnet).

MUX ska bara tända upp nästa (dvs en) siffra vid varje avbrott. Därav POS för att hålla reda på vilken siffra som ska tändas.

Lagra INTE POS i ett register, utan i SRAM.

BCD görs lämpligen som en while-loop, eftersom samma princip gäller för varje siffra men med olika gränser:
Räkna upp, för stor? Nej->färdig, Ja->nollställ gå till nästa.

Lab3 : BCD, tre varianter

En 59:59-klocka (2 loop-varv)

```
; ISR BCD
; Increase time by one second
BCD:
    * spara kontext
    * initiera pekare
BCD_LOOP: "while fler siffror"
    * entals-del++
    * om entals-del < 10 : BCD_EXIT
    * nollställ entals-del
    * tiotal-del++
    * om tiotal-del < 6 : BCD_EXIT
    * nollställ tiotal-del
    rjmp BCD_LOOP
BCD_EXIT:
    * återställ kontext
    * reti
```

En 59:59-klocka (4 loop-varv)

```
; ISR BCD
; Increase time by one second
BCD:
    * spara kontext
    * initiera pekare
BCD_LOOP: "while fler siffror"
    * siffr++
    * läs TAL från TAB
    * om siffr < TAL : BCD_EXIT
    * nollställ siffr
    rjmp BCD_LOOP
BCD_EXIT:
    * återställ kontext
    * reti

TAB: .db 10,6,10,6
```

En 99:99-klocka med efterjustering (4 loop-varv)

```
; ISR BCD
; Increase time by one second
BCD:
    * spara kontext
    * initiera pekare
BCD_LOOP: "while fler siffror"
    * siffr++
    * om siffr < 10 : BCD_ADJUST1
    rjmp BCD_LOOP
BCD_ADJUST1:
    * tiotal-sekunder < 6 : BCD_ADJUST2
    * nollställ tiotal-sekunder
    * rjmp BCD_LOOP
BCD_ADJUST2:
    * tiotal-minuter < 6 : BCD_EXIT
    * nollställ tiotal-minuter
BCD_EXIT:
    * återställ kontext
    * reti
```

Lab3 : MUX

```
; ISR MUX
; Display next digit
MUX:
    * spara kontext
    * hämta pos
    * pos++
    * pos < 4 : MUX_NEXT
    * nollställ pos
MUX_NEXT:
    * spara pos
    * hämta siffra [TIME+POS]
    * hämta utseende [7SEGTAB+SIFFRA]
    * nollställ tidigare siffra
    * byt till nästa position
    * tänd ny siffra
MUX_EXIT:
    * återställ kontext
    * reti
```

Observera!

Endast en siffra kommer att vara tänd åt gången.

Dvs, MUX ska bara släcka nuvarande siffra och tända nästa siffra.

Före MUX:

0	4	5	3
---	---	---	---

Efter MUX:

0	4	5	3
---	---	---	---

LAX

...

LAX

- LAX:en går 24-25/5
- LAX:en är 90 minuter
- Anmälan i Lisam
- LAX:en görs enskilt, dvs inget samarbete, utan hjälp
- Det finns övnings-LAX:ar på kurshemsidan
- Labbar + LAX = godkänd kurs

Extra-labb

...

Extra-labb (bara för Labb1 och Labb2)

- Extra1 : 27/4 kl 08:15-10:00
- Extra2 : 28/4 kl 08:15-10:00
- Anmälan i Lisam

Tid för Frågor

Anders Nilsson

www.liu.se