Datorteknik
Labbserie med Arduino/KPAD

Uppdateras löpande. Se datumet:

10 maj 2021

Innehåll

0 Inledning **5**
0.1 Allmänt om laborationstillfällena 6
0.2 Labkit 7
0.3 Översikt labserie 8

1 LAB1. Miniprojekt **9**

2 LAB2. Morsesändare **13**

3 LAB3. Digitalur med LCD-display och avbrott **15**

4 LAB4. Radeditor **21**

5 Redovisning/examination **25**

0 Inledning

Labserien om fyra labbar i den här kursen är utvecklad för att till stor del kunna skötas via distansarbete. Det innebär ett ansvar på dig som student att arbeta självständigt och för egen maskin ”ta tag i saker”.

I grova drag består kursen av

- Övningsuppgifter (från hemsidan)
- Delmoment
- Laborationer

Dessutom tillkommer lektioner.

Arbetsgången är hela tiden densamma: Övningsuppgifter och mindre *delmoment* leder fram till ökad kunskap om hårdvaran och färdiga användbara rutiner/funktioner som sedan kan användas som byggblock i själva labben. Stor vikt läggs vid att dessa rutiner skall namnges på ett klart sätt och ”göra det dom heter”.

Alltså: **Övningsuppgifter → delmoment → laboration**

Längre in i kursen kommer vi lära oss hur rutinerna ska konstrueras för att verkligen kunna användas fritt utan oväntade sidoeffekter. Med den kunskapen finns det anledning att gå tillbaka och skriva om eller modifiera rutinerna så att de hela tiden blir bättre. På detta sätt har du som student en egen ”levande” kodbas som kan förväntas ändras för att hela tiden bli mer generellt användbar.

Laborationerna baseras till stor del på strukturerad programmering. En del laborationer definieras/förklaras i form av strukturdiagram enligt JSP och du anmodas använda JSP-metoden i hela labkursen.

Som exempel kommer du att bygga kod som styr en LCD-display. Den koden kommer du använda senare så det är väsentligt att den är modulärt och fristående uppbyggd, ”ortogonal” för att göra en lineär algebrajämförelse.

OBS: Föreläsningshäftet och laboration 1 och 2 (morselabben) baserar sig på vår normala hårdvara varför du få ha överseende med förekommande referenser till *Dalia*-kort med mera. Det du läser här är det som gäller.

Du skall hela tiden använda utvecklingsmiljön AtmelStudio för att genomföra övningsuppgifter, delmoment och avslutande laboration. Processorn är ATMega328p (inte ATMega16A som används i föreläsningshäftet).

Glöm inte att ständigt gå tillbaka till din tidigare kod och begrunda hur den gör. Modifiera så den blir bättre. Använd sedan den bättre versionen.

0.1 Allmänt om laborationstillfällena

Laborationer och lektioner är att betrakta som resurstillfällen för att ställa frågor för din fortsatta programmering.

Du *kan* behöva anmäla dig till labtillfällena även om de går virtuellt på distans. Detta för att hindra överlastning av resurserna vid labtillfället. Examinator meddelar vad som gäller.

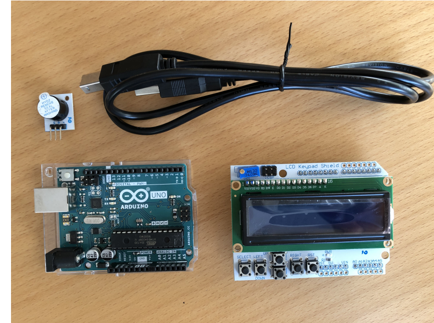
Du kommer inte bli godkänd på laborationen vid laborationstillfället. Se kapitlet *5 Redovisning/examination* för detaljer.

Mellan kurstillfällena pågår kursen kontinuerligt. Det innebär att det är ditt eget ansvar att vara aktiv och hålla takten. Till din hjälp finns övningsuppgifter och delmoment inför laborationerna.

0.2 Labkit

Du har fått labutrustning i en ziplock-påse. Labutrustningen används för Lab 2, 3 och 4. Laboration 1 utförs som en ren datorsimulering i utvecklingsmiljön AtmelStudio. Du måste ha AtmelStudio installerat.

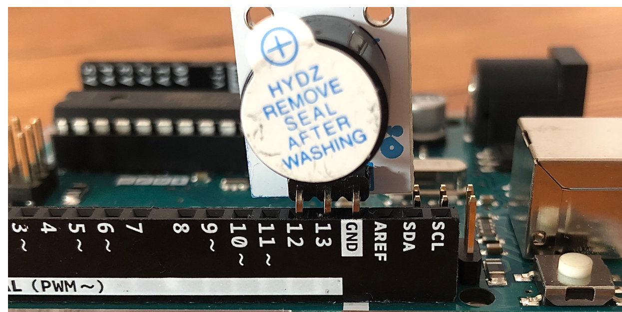
Påsen innehåller den laborationsutrustning som visas på bilden till höger



- 1 st Arduino Uno mikrokontrollerkort
- 1 st LCD Keypad Shield (KPAD)
- 1 st Summer
- 1 st USB-A/B-kabel

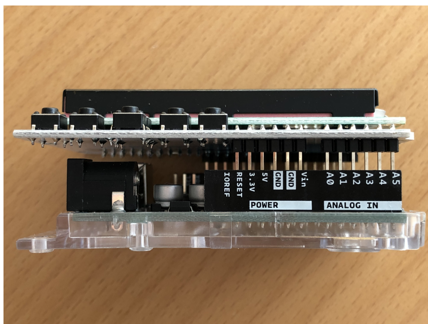
Här visas hur du ansluter delarna till varann inför varje laboration.

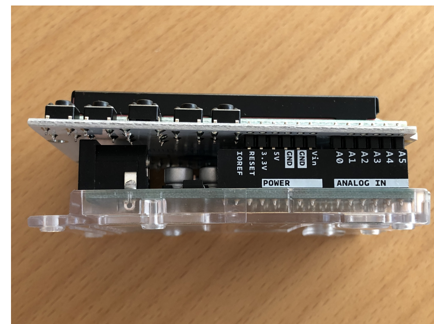
För Lab 2 behöver du Arduinokortet, summern och USB-sladden.



Summern ansluts mellan benen GND och 12 och måste vändas rätt, annars är den tyst. Arduinons pinne 12 heter PB4 och nås med `set`/`clr`-instruktionerna '`sbi PORTB,4`' respektive '`cbi PORTB,4`' från programmet. Låt skyddstejpen vara kvar, den låter högt nog ändå...

För Lab 3 och 4 behöver du Arduinokortet, KPAD och USB-sladden. Avlägsna summern.





Placera KPAD över Arduinokortet så att stiften stämmer överens med hylslisterna. Sedan trycker du försiktigt ihop de två delarna. Nya kontakter kan kärva något.

Funktionstest För ett inledande funktionstest ansluter du USB-kabeln till kortet. En grön lysdiod med fast sken indikerar att matningspänning finns och en gul blinkande lysdiod är processorns förprogrammerade inbyggda program som kör. Om du ser blinket är kortet OK. OBS: På vissa kort är strömbegränsningsmotstånden till lysdioderna av fel värde och blinket är därför mycket svagt!

0.3 Översikt labserie

Innan den egentliga labserien börjar är ett *Installationstillfälle* schemalagt där du kan få hjälp att installera utvecklingsmiljön.

Labserien består av fyra laborationer enligt nedan:

LAB1 (kräver AtmelStudio)

Laborationen utgörs av kodskrivning och simulering i utvecklingsmiljön. Ingen ytterligare hårdvara behövs. Här skall du lära dig hur man hanterar processorns register, gör villkorliga hopp, konstruerar vänteloopar, läser in en yttre signal och skickar signaler signaler till yttervärlden. All signalering sker i simulatorn, både simulera yttre tryckknapp och läsa av utskickat värde på mikrokontrollerns pinnar.

Lab1 utgör de absoluta grunderna för att kunna tillgodogöra dig resten av kursen. Det är helt rimligt att, vid behov, göra denna laboration flera gånger för att innehållet ska fastna ordentligt.

LAB2 (kräver Arduinokort och en högtalare)

Här skall du tillverka ett program som skickar ut morsekod enligt det internationella morsealfabetet. I ett delmoment tillverkar du rutiner som kan pipa de ”korta” och ”långa” tonerna. I ett annat lär du dig hur tabelluppslagning görs. Slutligen skall ditt program tuta ut en text lagrad i minnet till högtalaren.

LAB3 (kräver *shield*en KPAD)

Här tillverkar du en klocka som visar tiden på en LCD-display. Du börjar med att konfigurera och testa displayen så att du kan skriva ASCII-tecken till den. De rutiner du skriver här kommer användas även i nästa laboration.

Tiden skall lagras BCD-kodad i minnet och ett delmoment är att skriva kod som räknar upp tiden ett steg (sekund) för varje gång koden körs. Tiden mellan två uppräkningar kan initialt bestämmas av en vänteloop men skall slutligen styras med avbrott via en intern *timer*.

LAB4 (kräver *shield*en KPAD)

Denna labb använder analog/digital-omvandlare för att läsa in tangentnedtryckningar som i sin tur används för att styra en radeditor på displayen. Med knapparna skall man första kunna välja position på displayen och sedan välja tecken som skall vara på den positionen.

Ladda ner AtmelStudio enligt `0_KursInformation.pdf` på hemsidan. När du gjort övningsexemplen och delmomenten är du redo för den efterföljande labuppgiften.

Slutligen ett råd: Ingen skriver vacker, eller ens fungerande, kod direkt. Gå fram stegvis i små steg. Stora kodstycken är notoriskt svåra att felsöka. Undvik dem. När du väl fått koden att fungera går du tillbaka och snyggar till den.

1 LAB1. Miniprojekt

Syfte: I den första laborationen skall du ska bekanta dig med utvecklingsmiljön genom att programmera och modifiera en given kod. Hela laborationen genomförs som en **simulering** i AtmelStudio.

Tips: Aktivera radnummer i AtmelStudio genom "Tools/Options/Text Editor/Assembler", klicka i rutan "Display/Line Numbers"

Övningsuppgifter: Inför denna labb är grundläggande kunskap av processorn tillräcklig. Lämpliga övningsuppgifter är minst 1 – 16.

Delmoment: Skriv in och simulera kod som genomför följande. Testa i simulatorn så noggrant som möjligt. Se till att du förstår statusregistrets, `SREG`, funktion för koden.

1. Gör en vänteloop som tar en byte som argument. Hur lång tid tar den som kortast och längst att utföra?
2. Använd instruktionen `adiw` för en 16-bitars loop, dvs en som räknar så långt den kan med 16 bitar (65536). Notera hur lång den är i millisekunder. Den längden är en lämplig byggsten för senare tidsfördröjningar.
3. Lägg till instruktioner i looparna som bryter dem vid knappnedtryckning på `PINC` bit 0.

Labuppgift: Du återfinner hela laborationen som *Appendix E, Miniprojekt* i föreläsningshäftet. Uppgiften är densamma här ***men utförs*** som en ren simulering i denna laboration med följande modifieringar:

- Koden i föreläsningshäftet är för processorn ATMega16A. Den har `PORTA` vilket *inte* ATMega328p har, så antingen skriver du av koden i *Appendix E* och byter processor till ATMega16A eller så väljer du ATMega328p och följande kod (ändrade rader markerade med '`<---`').
- Detta betyder också att vi inte använder en 7-segments LED-modul som utenhet. De fyra utbitarna `B3-B0` avläses istället i simulatorns I/O-register (Välj "Debug/Windows/I/O view").

```
1       ; r16-r19 free to use
2       .def     num = r20 ; number 0-9
3       .def     key = r21 ; key pressed yes/no
4
5       ; set stack
6       ldi      r16,HIGH(RAMEND)
7       out      SPH,r16
8       ldi      r16,LOW(RAMEND)
9       out      SPL,r16
10
11      call     INIT
12      clr      num
```

```
13  FOREVER:
14      call    GET_KEY     ; get keypress in boolean 'key'
15  LOOP:
16      cpi     key,0
17      breq    FOREVER     ; until key
18      out     PORTB,num   ; print digit
19      call    DELAY
20      inc     num         ; num++
21      cpi     num,10      ; num==10?
22      brne    NOT_10      ; no, so jump
23      clr     num         ; was 10
24  NOT_10:
25      call    GET_KEY
26      jmp     LOOP
27
28      ;
29      ; --- GET_KEY. Returns key != 0 if key pressed
30  GET_KEY:
31      clr     key
32      sbic    PINC,0      ; <---- skip over if not pressed
33      dec     key         ; key=$FF
34      ret
35
36      ;
37      ; --- Init. Pinnar on C in, B3-B0 out
38  INIT:
39      clr     r16
40      out     DDRC,r16    ; <----
41      ldi     r16,$0F
42      out     DDRB,r16
43      ret
44
45      ;
46      ; --- DELAY. Wait a lot!
47  DELAY:
48      ldi     r18,3
49  D_3:
50      ldi     r17,0
51  D_2:
52      ldi     r16,0
53  D_1:
54      dec     r16
55      brne    D_1
56      dec     r17
57      brne    D_2
58      dec     r18
59      brne    D_3
60      ret
```

Obs! Kopiera aldrig koden ur pdf:en. Det följer ofta med osynliga tecken som AtmelStudio fastnar på, och som är lögn att hitta. Skriv av koden! Om du vill använda koden utan subrutinanrop (`call`) får du själv genomföra ändringar ovan.

För att bli smidig i simulatorn använder du speciellt följande snabbkommandon:

- Använd F11 för att enkelstega genom kod.
- Använd F10 för att enkelstega över rutiner.
- Använd Ctrl+F10 för simulera fram till cursorn.
- Använd F5 för att köra i full fart.
- Använd F9 för att sätta en brytpunkt på raden du är. F9 igen tar bort brytpunkten. Alternativt kan du klicka längst ut till vänster på raden
- Använd Ctrl+Shift+F5 för att avbryta debuggningen.
- Använd Ctrl+K, Ctrl+C för kommentera ett markerat block.
- Använd Ctrl+K, Ctrl+U för avkommentera ett markerat block.

Du kan modifiera din kod på olika sätt. Till exempel så den:

- känner av en annan insignal (annan pinne, annan port)
- skickar ut data på övre porthalvan (`B7-B4`) istället
- direkt utför additionen på `r20`:s höga nibble
- skickar ut data på en annan port
- räknar till annat slutvärde
- räknar baklänges

Kommentar:

- I programmet har registren döpts om till `num` och `key`. Det är strängt taget onödigt och kanske till och med förvillande. Programmet har med detta som exempel på att det *går* att göra så, inte att det nödvändigtvis är bra. Det går utmärkt att skriva `r20` och `r21` direkt i stället.
- Programmet är avsett för en klockfrekvens på 1 MHz, motsvarande ungefär 1 instruktion per mikrosekund[1] (10^{-6} s). På grund av detta är vänteloopen `DELAY` inte mindre än tre nästlade loopar, annars skulle den ta slut alldeles för fort. För vår simulerings skull behövs inte mer en nivå, modifiera koden till detta.

  Hur många klockcykler tar tre nästlade loopar som mest? Hur många varv blir det? Vad motsvarar det för tid per instruktion? Allt detta kan du simulera fram.
- Notera hur rutinen `GET_KEY` gör vad den heter. En del tycker att de tre programraderna är för få för att ges ett eget namn men programmets tydlighet och läsbarhet tjänar på det. *I valet mellan läsbarhet och kodstorlek vinner alltid läsbarhet.*
- Se till att du, åtminstone på ett översiktligt plan, förstår det (i föreläsningshäftet *Appendix E*) givna strukturdiagrammet. Vi får tillfälle att återkomma till sådana i senare labbar.
- Efter laborationen skall du kunna mata in kod, kompilera, enkelstega, sätta brytpunkter, analysera innehåll i register och I/O-register samt känna igen programdelarna sekvens, iteration och selektion Du känner dig dessutom bekväm med utvecklingsmiljön. Om inte: Repetera. Resten av kursen hänger på detta!

[1]De enklaste instruktionerna utförs på en enda klockcykel.

2 LAB2. Morsesändare

Syfte: I den här labben skall du tillverka en utsignal som hörs i en summer. Dessutom skall du använda ytterligare adresseringsmoder samt konstant-tabeller i FLASH-minnet för textsträngen som skall sändas. Slutresultatet är en sändare som piper morsekod.

Tips: Det finns mobilappar som tolkar morsepipen till text![1]

Övningsuppgifter: Lämpliga uppgifter är 14–21 dvs minnesåtkomst av SRAM, tabeller i FLASH och ASCII-kodning. Uppgifterna 37 och 38 för konfiguration av portarna. Quiz-frågorna 40–45 a)–g).

Delmoment: Bland annat dessa rutiner behöver du säkert i det färdiga programmet.

1. Summern. Summern är sådan att den piper när den får en digital etta (5V) på sig och tystnar när signalen blir låg (0V) igen. Enklast är att använda 'sbi PORTB,4' respektive 'cbi PORTB,4' för detta.

   a) Skriv en rutin BEEP som sänder ett pip med en längd som anges i r16. Med r16=1 sänds ett pip ut och med r16=10, tio gånger längre pip osv. Pipen skall kunna varieras mellan *cirka*[2] 20 och 250 ms.[3]

   b) Skriv en rutin NOBEEP som sänder tystnad av **exakt** samma längd som BEEP med samma argument. Med **exakt** menas så exakt du kan.

   c) Genom upprepade anrop av BEEP och NOBEEP i en evig loop ska du kunna höra ”pip-pip-pip-pip-pip-. . . ”. Ändra på r16 och lyssna på resultatet.

2. Tabeller. (Enbart simulering i detta delmoment.)

   En sträng i programminnet (.cseg) avslutas med en tom, NUL, byte:

```
TEXT:
    .db "HELLO WORLD",$00
```

   a) Skriv en rutin som använder Z-pekaren för att successivt returnera tecken för tecken ända till strängen är slut.

   b) Skriv en rutin LOOKUP som som returnerar binärkodningen för ASCII-tecknet i r16. Binärkodningen framgår av labhäftet för morselabben på hemsidan. Du behöver inte ta hänsyn till andra tecken än A–Z. Rutinen får inte påverka Z-pekaren.

[1] På Chrome *kanske* du kan ha glädje av https://morsecode.world/international/decoder/audio-decoder-adaptive.html

[2] Verkligen *cirka* dvs 10–200 ms duger också

[3] Använd adiw-delayen från förra laborationen.

Labuppgifter: **Uppgift 1** Läs och förstå ursprungslabhäftet (`3_Lab_Morse.pdf`) på hemsidan. Labben är samma som tidigare, enda skillnaden är att en högtalare monteras på Arduinokortet enligt bild i avsnittet *1.2 Labkit* tidigare i detta dokument. Detta medför att du inte själv behöver skapa vågformen som utgör ljudet.

Du skall använda de rutiner som skickar ut pipljud till högtalaren som du skrev i Delmomentet. Timingen är viktig, var noggrann där.

Använd strukturdiagrammet och pseudokoden i labhäftet som grund.

Inför **Uppgift 2**, tänk på att du vill avgöra specialfall så tidigt som möjligt i din kod. `LOOKUP` ska inte behöva hantera specialfall, bara göra en tabelluppslagning. Tips: Du kan använda simulatorn för att säkerställa att mellanslaget blir 7 tidsenheter långt.

Gör **Uppgift 3** i mån av intresse.

I den examinerande videon skall framgå att programmet kan sända ditt LiU-ID, tre gånger med ett mellanslag mellan varje; `MICJO` som exempel:

```
MICJO␣MICJO␣MICJO
```

Kommentar: Denna labb var en rejäl duvning i assemblerprogrammering eller hur? Det är många delar som måste fungera oklanderligt var för sig innan de kopplas ihop till ett fungerande program. Du har också märkt att assemblerprogrammering inte är en speciellt förlåtande verksamhet. Därför ägnar vi mycket tid åt ett strukturerat angreppssätt i problemlösningen.

Du har nu genomfört många små men väsentliga moment, som

- uppdela kod i subrutiner
- tabelluppslagning av konstanter i FLASH-minnet
- tillverka fördröjnings/timing-loopar med in-argument
- skicka ut en digital signal bit-vis.
- använda ASCII-kodade tecken
- avsluta strängar med `NUL`-tecknet (`$00`)

Dessutom har du antagligen hört mer morsekod än någonsin tidigare`:)`

3 LAB3. Digitalur med LCD-display och avbrott

Syfte: Här skall du konstruera en digital klocka som visar tiden på LCD-displayen: `00:00:00` till `23:59:59`.

Du skall slutligen använda *avbrotts*-mekanismen för att stega fram tiden sekunderna med intern timer.

Den slutliga koden skall dessutom utformas så den inte har oönskade sidoeffekter om den används som en del i ett större programvaruprojekt. Detta gäller speciellt avbrottsrutinen. Se dokumentet om *Kodstil*.

Övningsuppgifter: Lämpliga uppgifter är 23 (samband mellan BCD och ASCII), 62 för BCD-uppräkning, 65 för pekarbegränsningar.

Delmoment: I delmomenten skriver du kodstycken som behövs för att initiera och skriva ut text på displayen, räkna upp tid och slutligen använda en intern *timer* som tidbas. Använd de föreslagna rutinnamnen i din kod för att underlätta kodgranskning för labassistenterna.

Nu är det också dags att designa ditt program inte bara strukturerat, utan också i övrigt enligt god programmeringssed (***Appendix K Erfarenhetslista***) i föreläsningsmaterialet.

1. **Vänta.** Skriv en rutin `WAIT` om minst cirka 10 ms, denna kommer du använda som fördröjning i hela denna laboration.
2. **Backlight**. Skriv rutinerna `BACKLIGHT_ON` och `BACKLIGHT_OFF` som tänder respektive släcker displayens backlight. Funktionstesta sedan backlighten med kod av typen

```
AGAIN:
    call    BACKLIGHT_ON
    ; wait a while
    call    BACKLIGHT_OFF
    ; wait a while
    jmp     AGAIN
```

3. **Initiera LCD-displayen.** Läs hårdvarubeskrivningen (`2_Hardvara.pdf`) på kurshemsidan, speciellt databladets delar om initiering till fyra-bitarsmod. All skrivning av data till displayen i fyrabitarsmod sker på `D`-portens övre halva (`D7-D4`).

   Notera speciellt att:

   - Backlighten måste vara på för att något skall synas.
   - Det är viktigt med en väntetid efter spänningspåslag innan man adresserar displayen, annars pratar man med en yrvaken display och den bryr sig inte. Vänta minst 10 ms.

- Vi kan inte läsa *Busy Flag* (och på så sätt ta reda på när displayen är redo för en ytterligare skrivning) utan måste själva vänta så att displayen hinner göra klart mellan skrivningarna. Vänta minst 10 ms efter varje skrivning. (Forts nästa sida...)
- Varje skrivning till displayen måste utföras med en komplett sekvens enligt databladet. Speciellt räcker det **inte** att bara lägga ut data till den, signalen `E`:s fallande flank är verkställande så när den faller (hög→låg) måste displaydata vara giltigt.

Skriv sedan de angivna rutinerna a)–c) nedan innan du startar displayen enligt d).

a) `LCD_WRITE4:` som genomför en skrivning av `r16`:s övre nibble till displayen. `E` måste vara hög under fyra instruktioner, exvis `nop`. Vänta sedan så att skrivningen hinner behandlas internt i displayen.

b) `LCD_WRITE8:` som genomför en skrivning av hela `r16` till displayen, först övre nibble, sedan lägre.

c) `LCD_ASCII:` och `LCD_COMMAND:`, som skriver ut en byte i `r16` som ASCII-tecken eller displaykommando. Använd rutinerna ovan som byggblock.

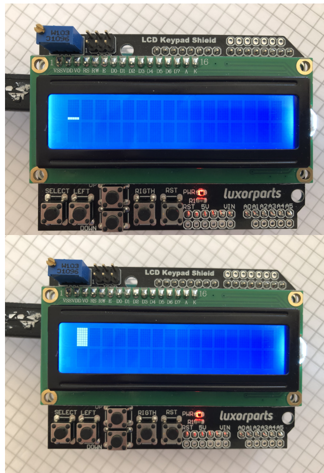
d) Komplettera kodskelettet, '`***`'-markeringarna, enligt databladet och med rutinerna ovan:

```
    .equ    FN_SET  = $***
    .equ    DISP_ON = $***
    .equ    LCD_CLR = $***
    .equ    E_MODE  = $***
LCD_INIT:

    ; --- turn backlight on
    ***
    ; --- wait for LCD ready
    ***

    ;
    ; --- First initiate 4-bit mode
    ;

    ldi     r16,$30
    call    LCD_WRITE4
    call    LCD_WRITE4
    call    LCD_WRITE4
    ldi     r16,$20
    call    LCD_WRITE4

    ;
    ; --- Now configure display
    ;

    ; --- Function set: 4-bit mode, 2 line, 5x8 font
    ldi     r16,FN_SET
    call    LCD_COMMAND

    ; --- Display on, cursor on, cursor blink
    ldi     r16,DISP_ON
    call    LCD_COMMAND

    ; --- Clear display
    ldi     r16,LCD_CLR
    call    LCD_COMMAND

    ; --- Entry mode: Increment cursor, no shift
    ldi     r16,E_MODE
    call    LCD_COMMAND
    ret
```

Glöm inte att sätta korrekta portriktningar[1] på de bitar du använder innan testkörning!

En korrekt initiering ger en display med blinkande cursor:

`BLINK:`



`jmp BLINK`

e) Du kommer även behöva rutiner för att börja skriva i kolumn 0, `LCD_HOME:` samt tömma skärmen, `LCD_ERASE:`. Skriv och testa även dessa. (Notera att rutinerna kan dela på mycket kod! Strukturera!)

4. **Utskrift.** Skriv en rutin, `LCD_PRINT`, som skriver ut den NUL-terminerade ASCII-strängen `Z` pekar på från SRAM. Detta kommer vara din enda utskriftsrutin i denna och nästa laboration.[2]

   Vill du skriva ut strängen i `LINE` blir det till exempel:

```
LINE_PRINT:
     call    LCD_HOME          ; column zero of display
     ldi     ZH,HIGH(LINE)     ; start of string
     ldi     ZL,LOW(LINE)
     call    LCD_PRINT         ; print it
     ret
```

5. **Tidsuppräkning**. Tiden skall återfinnas BCD-kodad[3] i sex minnesceller med början i adressen `TIME:` i SRAM.

   Endast den lägsta nibblen, de fyra lägsta bitarna, i varje byte skall innehålla information om vilken *decimal* siffra byten representerar. [4]

   Tiden ”18:04:53” anges i sex bytes i minnet som

01	08	00	04	05	03

   och skall räknas upp enligt nedan

[1] I en egen subrutin naturligtvis. `LCD_PORT_INIT:` kan vara ett rimligt namn.

[2] Vill du — alltså inte nödvändigt för denna labb — även skriva ut fast text från FLASH måste du skriva en egen rutin som tar tecken med `lpm` som i förra laborationen. Den kan då heta `LCD_FLASH_PRINT:`.

[3] Fyra bitar kan rymma siffrorna `0`–`F`, med BCD-kodning tillåter vi dock enbart siffrorna `0`–`9`.

[4] Använd BCD. Somliga försöker addera direkt i ASCII. Gör inte det. ASCII är inte en datatyp lämplig för aritmetik.

Tid	TIME+5	TIME+4	TIME+3	TIME+2	TIME+1	TIME+0
00:00:00	0	0	0	0	0	0
:	:	:	:	:	:	:
00:00:09	0	0	0	0	0	9
00:00:10	0	0	0	0	1	0
00:00:11	0	0	0	0	1	1
:	:	:	:	:	:	:
00:00:19	0	0	0	0	1	9
00:00:20	0	0	0	0	2	0
00:00:21	0	0	0	0	2	1
:	:	:	:	:	:	:
00:00:59	0	0	0	0	5	9
00:01:00	0	0	0	1	0	0
00:01:01	0	0	0	1	0	1
:	:	:	:	:	:	:
00:59:59	0	0	5	9	5	9
01:00:00	0	1	0	0	0	0
:	:	:	:	:	:	:
23:59:59	2	3	5	9	5	9
00:00:00	0	0	0	0	0	0

Det visar sig enklare för uppräkningen om minnesadressen `TIME+0` pekar på sekundsiffran till höger. `TIME+1` pekar på tiosekundsiffran osv. Tiden ligger alltså i själva verket baklänges i minnet.

Skriv en rutin `TIME_TICK:` som räknar upp den BCD-kodade tiden enligt ovan. Rutinen skall vara sådan att **varje anrop ökar tiden ett steg (”sekund”)**.

Simulera `TIME_TICK:` och läs av dess effekt direkt i minnet SRAM. Testkoden kan vara denna, med en brytpunkt på raden efter anropet:

```
TIME_TEST:
    call    TIME_TICK
    jmp     TIME_TEST   ; <--- break point here
```

6. **Tidsutskrift.** Skriv rutinen `TIME_FORMAT:` som formatterar tiden i `TIME` till en sträng färdig för utskrift i `LINE` enligt formatet `Hh:Mm:Ss`.

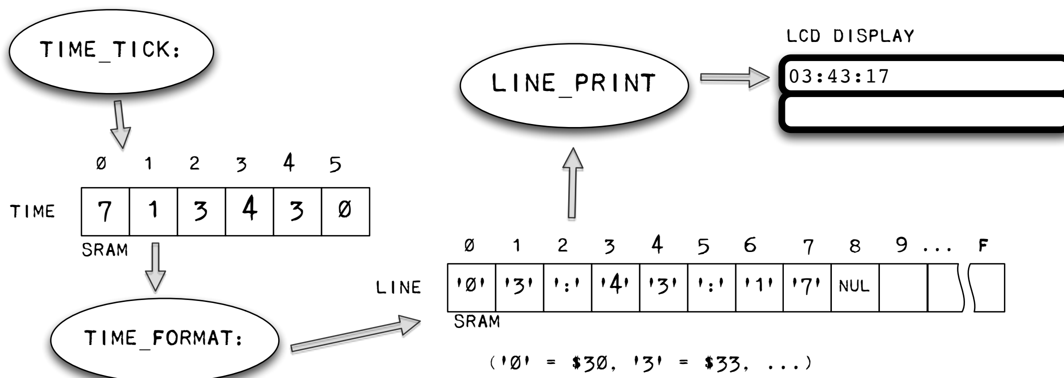
7. **Avbrott.** Koden nedan använder mikrokontrollerns interna 16-bitars timer för att generera avbrott varje sekund:

```
    .equ    SECOND_TICKS = 62500 - 1    ; @ 16/256 MHz
TIMER1_INIT:
    ldi     r16,(1<<WGM12)|(1<<CS12)    ; CTC, prescale 256
    sts     TCCR1B,r16
    ldi     r16,HIGH(SECOND_TICKS)
    sts     OCR1AH,r16
    ldi     r16,LOW(SECOND_TICKS)
    sts     OCR1AL,r16
    ldi     r16,(1<<OCIE1A)             ; allow to interrupt
    sts     TIMSK1,r16
    ret
```

Använd koden för att fånga upp avbrottet via avbrottsvektorn `OC1Aaddr` (motsvarande adress `$0016`).

För att veta att avbrottet tagits kan din kod ge ett pip eller tända/toggla backlighten på KPAD. Glöm inte att slå på avbrott globalt (`sei`) för att få det att hända.

Labuppgift: Använd rutinerna i delmomenten ovan för att realisera klockan med ett dataflöde enligt figuren:



`TIME_TICK:` *räknar upp tiden i* `TIME`, `TIME_FORMAT` *konverterar tiden till en skrivbar* ASCII*-sträng i* `LINE`. `LINE_PRINT` *skriver slutligen ut strängen på displayen.*

1. I en första version kan du använda en sekundlång loop, `WAIT_1SEC`, bara för att konstatera att klockan räknar upp som den skall på displayen, dvs kod ungefär som denna:

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

2. I slutversionen använder du naturligtvis ett mycket mer noggrant timeravbrott för att få klockan att stega upp exakt en gång per sekund.

   Om avbrottsrutinen håller ordning på tiden helt autonomt kan huvudprogrammet ägna sig åt att skriva ut tiden:

```
MAIN:
    call    TIME_FORMAT
    call    LINE_PRINT
    jmp     MAIN
```

   `MAIN:` körs i full fart och dess rutiner kommer regelbundet bli avbrutna av timerabvrottet. Det är därför viktigt att avbrottsrutinen utformas så den inte har några sidoeffekter (påverkar register osv).

Labuppgiften skall, förutom reflektionsdokument och kod, redovisas i en video där du visar att klockan kan ticka från `23:59:45` till `00:00:15`. Du behöver inte visa i videon, men kontrollera själv, att övergången `13:59:59` till `14:00:00` fungerar.

Kommentar: Notera hur vi successivt, i små steg, omformat siffrorna till tid, ASCII-kodad utskrift och slutligen skrivit ut tiden. Varje rutin är var för sig begränsad och överblickbar.

Avbrottsmekanismen gör att klockan går rätt oberoende av vad processorn håller på med i övrigt.

Om du skrivit koden för `BCD:` sekvensiellt, så märker du att det nästan är samma sak som görs flera gånger. Prova att göra `BCD:` mindre genom att huvudsakligen använda en loop.

4 LAB4. Radeditor

Syfte: Här skall du först tolka knapparna på KPAD genom analog/digital omvandling. Knapparna skall sedan användas för att navigera på displayens första rad och skriva in valfri text. Den här laborationen sammanfattar i princip allt i tidigare laborationer förutom avbrott. Det nya är AD-omvandlaren.

Övningsuppgifter: Då denna laboration blir ganska mycket kod är det viktigt att du strukturerar din kod från början. Studera övningsuppgifterna 50, 52 och möjligen 53–55.

Delmoment:

1. **Hexadecimal utskrift**. För att visa AD-omvandlingens resultat behöver du först konstruera utskriftsrutinen `LCD_PRINT_HEX:` som skriver ut `r16`:s värde på displayen i hexadecimal form. Med `r16`=159 skrivs `9F` ut och så vidare.[1]

   Prova din rutin enligt detta mönster med känt indata:

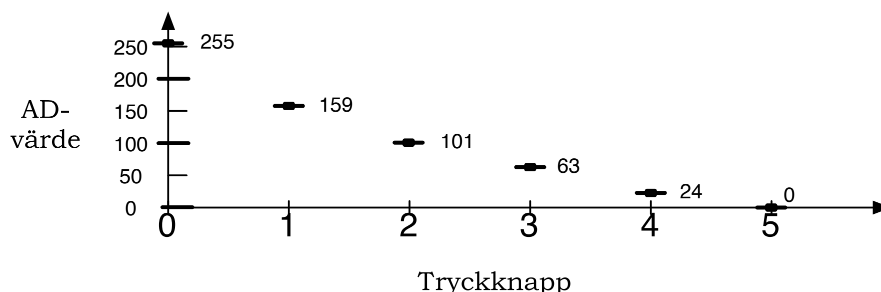
   ```
       ldi     r16,$9F
       call    LCD_PRINT_HEX
   STOP:
       jmp     STOP
   ```

   som ska skriva `9F` på displayen och så vidare. Förslag på `LCD_PRINT_HEX:` finns sist i denna laboration.

2. **AD-omvandlare**. Läs av ett 8-bits värde med rutinen `ADC_READ8:` där `r16` är den önskade kanalen, dvs `r16=0` för `PC0`.
   Se den givna koden i föreläsningsunderlaget, men notera att I/O-registren för AD-omvandlaren inte kan nås med de vanliga `in`- och `out`-instruktionerna. AD-omvandlarens I/O-adresser ligger utanför det område `in`- och `out`-instruktionerna kan nå.[2] De ligger i minnet och kan nås enligt ”`sts ADMUX,r16`” för skrivning och ”`lds r16,ADCH`” för läsning. Ett *prescaler*-värde på 128 är lämpligt. Spänningsreferensen skall sättas till AV_{cc}.

3. **Tryckknappar**. De olika knapparna på KPAD sitter i en resistorstege och varje knapp ger en viss utspänning. Bestäm vilka AD-omvandlarvärden dina knappar ger genom att skriva ut värdet med `LCD_PRINT_HEX:` till displayen.

   Med faktiska uppmätta AD-omvandlarvärden (255, 159, 101, 63, 24, 0) kan man konstruera diagrammet nedan. I detta fall ser man att AD-värden < 12 motsvarar knappen `5`, därefter < 42 är knappen `4` osv.



[1] Med en loop kan du göra en uttömmande test på *alla* möjliga indata.

[2] Se m328pdef.inc i din AtmelStudio-installation.

4. **Knappavkodning**.

   Skriv en rutin KEY: som *omedelbart* returnerar vilken knapp som är nedtryckt. Returvärdet i r16 skall kodas 0–5 enligt nedanstående tabell. Lämpliga intervall för de tidigare uppmätta AD-omvandlarvärdena är angivna. Använd dina egna uppmätta värden då de kan avvika från KPAD till KPAD.

Knapp	Ingen	SELECT	LEFT	DOWN	UP	RIGHT
Returvärde	0	1	2	3	4	5
Intervall	255–207	206–130	129–82	81–43	42–12	12–0

   Använd sedan KEY: för rutinen KEY_READ: som väntar att tidigare knapp först släppts. KEY_READ: *hänger* alltså och garanterar en ny knappnedtryckning.

```
KEY_READ:
    call    KEY
    tst     r16
    brne    KEY_READ            ; old key still pressed
KEY_WAIT_FOR_PRESS:
    call    KEY
    tst     r16
    breq    KEY_WAIT_FOR_PRESS  ; no key pressed
    ; new key value available
    ret
```

5. **Cursor**. Läs i LCD:ns datablad om hur man adresserar DDRAM. Cursorn placeras på den adress som anges, även om inget skrivs till den adressen. Skriv sedan en rutin LCD_COL: som placerar cursorn i den kolumn 0–15 som anges i r16. Värdet, 0–15, skall lagras i SRAM på adressen CUR_POS: och kan inte vara ett globalt register.

   Nu kan du kan testa om dina knapprutiner fungerar genom att sätta cursorn i den kolumn 0–5 som KEY_READ: returnerar.

```
MAIN:
    call    KEY_READ
    call    LCD_COL
    jmp     MAIN
```

Labuppgift: Konstruera ett program som kan skriva text med bokstäverna A–Z på displayen.

1. Med knapparna RIGHT och LEFT skall cursorn kunna styras till olika kolumner på displayens övre rad. Knapparna UP och DOWN skall bläddra fram ett tecken A–Z i vald kolumn på den raden. På detta sätt skall kan man skriva in och editera text på displayen.

   Med SELECT skall man kunna tända och släcka (*toggla*) displayens backlight.

   Markören skall inte kunna gå utanför raden på displayen.

2. (Detta är en valfri uppgift!) Ändra i programmet så man kan skriva in även mellanslag förutom *enbart* A–Z.

Labuppgiften skall, förutom reflektionsdokument och kod, redovisas i en video där du visar:

- att cursorn inte kan gå utanför raden,
- att du kan knappa in ditt Liu-ID på en ursprungligen tom display och
- att backlighten går att styra med SELECT.

Kommentar: Precis som i lab3 är det lämpligt att skriva ut hela raden för varje teckens ändring, dvs återanvänd `LINE_PRINT:`. Ett annat tillvägagångssätt är att bara skriva ut förändringen för berörd kolumnposition, detta bedöms ge mer komplicerad kod.

När du fått grundläggande funktion hos displayen kan du sannolikt minska den använda tidsfördröjningen ner till ett par millisekunder. Prova att successivt minska tidsfördröjningen, det är trevligare med snabb respons på knappnedtryckningarna.

Om du tycker att du får flera steg för varje knapptryckning kan det vara på grund av korta kontaktstudsar. Gör då så att du väntar några millisekunder innan du gör en andra, den egentliga, AD-omvandlingen.

Som förslag på `LCD_PRINT_HEX:` kan du använda nedanstående kodstycke:

```
    ; Write hexadecimal
    ; In:  r16, value
    ; Out: -
LCD_PRINT_HEX:
    call    NIB2HEX
NIB2HEX:
    swap    r16
    push    r16
    andi    r16,$0F
    ori     r16,$30
    cpi     r16,':'
    brlo    NOT_AF
    subi    r16,-$07
NOT_AF:
    call    LCD_ASCII
    pop     r16
    ret
```

5 Redovisning/examination

Varje laboration kräver ett *reflektionsdokument* i pdf. I Laboration 2, 3 och 4 skall du dessutom, till kursens sida på Lisam, skicka in

- en kort video, där du demonstrerar funktionen, samt
- laborationens kompletta kod.

1. Reflektionsdokumentet skall innehålla dina reflektioner över till exempel några av dessa frågor, du får naturligtvis gärna skriva annat också. En omfattning på en halv till en A4-sida med normalt radavstånd kan vara rimlig.

- Vad har jag lärt mig under denna laboration?
- Hur kan jag använda erfarenheterna i senare laborationer?
- Hur kan jag använda erfarenheterna i annan programmering?
- Vad var svårast/krångligast/mest tidskrävande?
- Sammanfattningsvis tyckte jag detta om labben: ...

2. Video Gör en video på högst en minut som visar och beskriver ditt resultat av laborationen. I videon måste funktionen tydligt framgå.

3. Kod Den kod som gav funktionen enligt videon skall redovisas. Vi vill se snyggt formatterad, lättläst kod med relevanta kommentarer. Koden skall vara körbar och **rensa bort**, inte bara kommentera bort, irrelevanta delar. Skicka in assemblerkodsfilen, **inte** hela projektet.

- För koden är det bara assembler-filen som ska skickas in, inte hela projektet. Filen ska alltså sluta på `.asm` och inte `.atsln` eller något sånt. Det är bara `.asm`-filen vi är intresserad av.
- För reflektionsdokumentet är det en PDF vi vill ha, **inte** .dot, .odt, .pages osv. PDF-en ser likadan ut oavsett plattform.
- För video finns flera format och vad jag hört så har ingen ännu skickat in något som vi inte kunna titta på. (.mp4, .mov är de vanligaste)
- Låt ditt Liu-ID framgå av filen så underlättar det också för oss. Filerna kan till exempel namnges enligt mallen:

  `micjo37_lab2.asm`   `micjo37_lab2.pdf`   `micjo37_lab2.mp4`

Samtliga dessa filer skall laddas upp till ditt kurskonto på Lisam om inget annat anges. Slutdatum för uppladdning meddelas på annat sätt (men är typiskt drygt en vecka efter labtillfället).