Laboration 5C
C-programmering på AVR
TSEA57 Datorteknik I

Anders Nilsson

2021-06-05
version 1.2

Innehåll

1 Introduktion

1.1 Syfte

Avsikten med den här laborationen är att du ska bekanta dig med C-programmering för AVR-processorn, och använda tabeller i minnet, avbrott, interna timers och A/D-omvandling. Den här versionen av laborationen är specifikt anpassad till processorn ATmega328. Förekommande namn på register, bitar och avbrottsvektorer kan för andra varianter av AVR-processorn behöva ändras.

1.2 Förkunskaper

För att på ett bra sätt tillgodogöra sig laborationen behöver man ha essensen av dom tidigare laborationerna i datorteknikkursen.

1.3 Material

Allt material som behövs till laborationen ingår i det utlånade lab-kitet.

1.4 Förberedelser

Som förberedelse behöver du ha läst igenom denna laboration. Det är också bra att bekanta sig med programspråket C ur ett generellt perspektiv. En bra källa för detta kan vara följande web-resurs: https://www.tutorialspoint.com/cprogramming/index.htm

2 Programspråket C

C är egentligen ett ganska litet språk, med avseende på mängden grundläggande konstruktionsprimitiver, och vid C-programmering för AVR så använder man typiskt begränsade delar av själva C-språket. Dock finns vissa specifika programbibliotek, som man behöver känna till, avsedda enbart för just C-programmering med AVR.

Man bör även vara medveten om att AVR-processorn har en tämligen begränsad mängd med arbetsminne (RAM) varför vissa språkkonstruktioner såsom rekursivitet eller dynamisk minnesallokering bör undvikas då det blir svårt att ha koll på mängden arbetsminne som då går åt.

AVR-processorn är en enkel 8-bitars processor, som jobbar med relativt låg klockfrekvens, och saknar hårdvarustöd för flyttal. Detta gör att den inte är någon mästare på tyngre beräkningar. Man kan dock göra flyttalsberäkningar på den, men dessa kommer att implementeras i mjukvara och tar då förstås relativt lång tid att utföra.

2.1 Programbibliotek

Här visas endast programbibliotek specifika för denna laboration.

2.1.1 <avr/io.h>

Programbiblioteket `io.h` innehåller definitioner av processorns alla I/O-register och namn på tillhörande bitar i dessa register. Detta bibliotek kan inkluderas i programmet med följande programrad:

```
#include <avr/io.h>
```

Det medför att man i sitt program direkt kan använda sig av namn på register och dess bitar, och dessa namn stämmer överens med de namn på I/O-register som finns beskrivna i AVR-processorns manual.

Programexempel:

```
#include <avr/io.h>

void main()
  {
    DDRB = (1<<PB0);     // PB0 as output, PB1-PB7 as input
    while(1)
    {
      PORTB = PIND;      // Copy PIND to PORTB
    }
  }
```

2.1.2 <util/delay.h>

Programbiblioteket `delay.h` deklarerar funktioner för så kallade *busy-wait*-loopar, dvs tomma programloopar som går runt-runt bara för att det ska förflyta en viss definierad tid. Dessa funktioner är:

```
void _delay_ms(double __ms)
void _delay_us(double __us)
```

Funktionen `_delay_ms(`*x*`)` väntar (loopar runt) programmet i *x milli*sekunder, och med `_delay_us(`*y*`)` väntar programmet i *y mikro*sekunder. Eftersom funktionerna använder sig av programloopar där varje resulterande assemblerinstruktion tar en viss tid att utföra så måste C-kompilatorn känna till vilken klockfrekvens som processorn använder, för att kunna konstruera en delay-funktion som väntar den angivna tiden. Dvs, klockfrekvensen måste deklareras som en konstant på följande sätt:

```
#define F_CPU 16000000UL        // 16 MHz clock
```

Programexempel:

```
#define F_CPU 16000000UL  // Processor is clocked with 16 MHz
#include <avr/io.h>
#include <util/delay.h>

void main()
{
  DDRB = (1<<PB2);    // PB2 as output, PB0-PB1, PB3-PB7 as input
  while(1)
  {
    PORTB = (1<<PB2); // Generate an output ...
    _delay_ms(500);   // ... on PB2 with ...
    PORTB = 0;        // ... a frequency of ...
    _delay_ms(500);   // ... 1 Hz
  }
}
```

2.1.3 <avr/interrupt.h>

Programbiblioteket `interrupt.h` deklarerar funktionalitet för avbrott. Bland annat så definieras namn på alla avbrottsvektorer. Dessa är:

Vektornamn	Betydelse
INT0_vect	External Interrupt Request 0
INT1_vect	External Interrupt Request 1
PCINT0_vect	Pin Change Interrupt Request 0
PCINT1_vect	Pin Change Interrupt Request 1
PCINT2_vect	Pin Change Interrupt Request 2
WDT_vect	Watch-dog Timeout Interrupt
TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
TIMER2_OVF_vect	Timer/Counter2 Overflow
TIMER1_CAPT_vect	Timer/Counter1 Capture Event
TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
TIMER1_OVF_vect	Timer/Counter1 Overflow
TIMER0_COMPA_vect	Timer/Counter0 Compare Match A
TIMER0_COMPB_vect	Timer/Counter0 Compare Match B
TIMER0_OVF_vect	Timer/Counter0 Overflow
SPI_STC_vect	SPI Serial Transfer Complete
USART_RX_vect	USART RX Complete
USART_UDRE_vect	USART, Data Register Empty
USART_TX_vect	USART TX Complete
ADC_vect	ADC Conversion Complete
EE_READY_vect	EEPROM Ready
USART_RX_vect	USART RX Complete
ANALOG_COMP_vect	Analog Comparator
TWI_vect	Two-wire Serial Interface
SPM_READY_vect	Store Program Memory Ready

Dessa avbrottvektorer används sedan för att deklarera avbrottsrutiner med funktionen `ISR` (Interrupt Service Routine).

Programexempel:

```
#include <avr/interrupt.h>

ISR(ADC_vect)     // Execute when ADC complete and interrupts enabled
{
  // code to read A/D converted value
}

void main()
{
  // code for initialising A/D converter
  // code for activating interrupts;
  // code for starting A/D conversion
  while(1)
  {
    // code for waiting for interrupt/conversion result
  }
}
```

Dessutom deklareras funktioner för att 1:ställa respektive 0:ställa I-flaggan i processorns statusregister `SREG`. Dvs, aktivera respektive inaktivera möjligheten för avbrott globalt i processorn.

```
sei();     // Activate interrupts globally
cli();     // Deactivate interrupts globally
```

2.1.4 <avr/pgmspace.h>

Programbiblioteket `pgmspace.h` deklarerar ett antal funktioner för att läsa data från programminnet (FLASH-minnet). T ex följande funktioner:

```
uint8_t pgm_read_byte(address)    // Read byte at address in FLASH-memory
uint16_t pgm_read_word(address)   // Read word at address in FLASH-memory
```

Det blir också möjligt att lägga in tabeller i programminnet med en konstant-deklaration tillsammans med ordet `PROGMEM` på följande sätt:

```
const uint8_t TABLE[] PROGMEM = {0, 1, 4, 9, 16, 25, 36};   // TABLE data
```

Programexempel:

```
include <avr/pgmspace.h>

const uint8_t SQUARE[] PROGMEM = {0, 1, 4, 9, 16, 25, 36};  // Square data

void main()
{
  uint8_t result;
  uint8_t data = 5;

  result = pgm_read_byte(&SQUARE[data]);    // result will be 25

  while(1);
}
```

2.2 Datatyper

Följande datatyper är relevanta vid C-programmering för AVR.

Datatyp	**Storlek**	**Talområde AVR**	**Likvärdig**
`char`	8 bitar	[0, 255]	`uint8_t`
`signed char`	8 bitar	[-128, 127]	`int8_t`
`unsigned int`	16 bitar	[0, 65535]	`uint16_t`
`int`	16 bitar	[-32768, 32767]	`int16_t`
`unsigned long`	32 bitar	[0, 2^{32}-1]	`uint32_t`
`long`	32 bitar	[-2^{31}, 2^{31}-1]	`int32_t`
`unsigned long long`	64 bitar	[0, 2^{64}-1]	`uint64_t`
`long long`	64 bitar	[-2^{63}, 2^{63}-1]	`int64_t`
`float`	32 bitar	[1.175494e-38, 3.402823e+38]	
`double`	32 bitar	[1.175494e-38, 3.402823e+38]	`float`

Datatypernas namn är standardiserade i C. Däremot är deras motsvarande storlek och talområde implementations- och plattformsberoende. Dvs, t ex en `int` på en AVR är inte lika stor som en `int` på en PC. Ett flyttal (en `float`) är samma sak som ett flyttal med dubbel precision (en `double`) på en AVR, medan dessa skiljer sig åt på en PC.

Skälen till att storleken på olika datatyper skiljer sig åt är att datatypen anpassats till vad som lämpar sig bäst för plattformen och dess typiska användningsområde. En AVR-processor är relativt simpel, har låg klockfrekvens och saknar hårdvarustöd för flyttal. Den är helt enkelt inte gjord för att utföra tunga beräkningar med precision. Därav finns egentligen inte heller behovet av flyttal med dubbel precision (`double`), men av kompatibilitetsskäl kan man ändå använda `double` fast det kommer att fungera precis som en `float`.

Som programmerare behöver man känna till storleken på olika datatyper för att veta vilka begränsningar som gäller. I en AVR kan det av det skälet vara tydligare att använda sig av den likvärdiga benämningen för heltal, t ex att skriva `uint8_t` istället för `char` eller `int16_t` istället för `int`. Med t ex `uint8_t` framgår det att det är ett heltal (int) utan tecken (u för unsigned) med 8 bitars storlek.

Utöver datatyperna i tabellen ovan förekommer vanligen datatypen `void` som är en obestämd datatyp, vilket också kan innebära ingen datatyp. Den används när en funktion inte returnerar eller tar något argument, eller då argumentet är av obestämd typ vilken kan bestämmas senare.

2.2.1 Type casting

Variabler i ett program har ju vanligen olika datatyper, utefter vad som lämpar sig bäst för variabeln i fråga. Vid beräkningar blir ofta flera variabler (ibland med olika datatyp) inblandade i samma uttryck. För att beräkningsresultatet ska bli korrekt kan man vara tvungen att göra s k *type casting*. Det innebär att en variabel just för tillfället, i själva beräkningen, görs om till en annan datatyp. Antag följande situation:

```
uint8_t inches;
float centimeters;

centimeters = (float)inches*254/100;
```

Variabeln `inches` är ett 8-bitars heltal (`uint8_t`), men beräkningen involverar operationer som kan resultera i ett decimaltal (dvs ett flyttal), och för att resultatet ska kunna bli det behöver man utföra *type casting* genom att skriva den önskade datatypen inom parentes framför uttrycket. Resultatet kommer förvisso annars att bli ett flyttal (i situationen ovan), dock ”utan decimaler” (dvs med nollor i decimalerna).

2.2.2 Heltalsberäkningar med division

Beräkningar i C med de fyra räknesätten (addition, subtraktion, multiplikation och division) fungerar på samma sätt som vid vanlig matematik, men med heltalsaritmetik blir det viktigt att uppmärksamma vad som händer vid division. Alla heltalsdivisioner ger ju bara heltal som resultat, dvs decimaldelen har förlorats. Det medför att ordningen på operationerna kan vara avgörande. Följande programkod:

```
uint8_t a = 7;
uint8_t b = a * 10 / 3;  // b = 23
```

ger inte samma resultat som:

```
uint8_t a = 7;
uint8_t b = a / 3 * 10;  // b = 20
```

I första fallet beräknas 7*10 (=70) delat med 3 blir 23,3333... dvs som heltal 23.

I andra fallet beräknas 7/3 (=2,3333... dvs som heltal 2) multiplicerat med 10 blir 20.

Eftersom vi kan förlora precision vid heltalsdivision så blir ordningen på operationerna alltså av betydelse. Man skulle tillfälligt under beräkningen kunna förändra datatypen till `float` (dvs göra type casting) för att behålla precisionen och göra sig (på sätt och vis) oberoende av ordningen, så här:

```
uint8_t a = 7;
uint8_t b = (unint8_t)((float)(a) / 3 * 10);  // b = 23
```

Då blir `a` ett flyttal och resultatet av divisionen också ett flyttal, och för att sedan spara det som ett heltal görs ytterligare type casting till `uint8_t`, **men** dels så blandar man in flyttal (vilket är krävande för en AVR-processor) och det blir stökigare kod.

Vid heltalsdivision är det dock möjligt att ta reda på den så kallade resten, den som egentligen orsakar precisionsförlusten och som gör att resultatet inte går jämnt upp. Istället för divisionetecken (`/`) används då procenttecken (`%`), så här:

```
uint8_t a = 70;
uint8_t b = a / 3;  // b = 23
uint8_t c = a % 3;  // c = 1 (70 - 3*23 = 1)
```

Ett exempel till:

```
uint8_t a = 170;
uint8_t b = a / 100;  // b = 1
uint8_t c = a % 100;  // c = 70 (170 - 1*100 = 70)
```

2.3 Generella register och RAM

AVR-processorns uppsättning med generella register (`R0` till `R31`) används inte vid C-programmering. Eller rättare sagt, de används inte av programmeraren, men likväl av C-kompilatorn för att sköta flytt av data mellan minnen och I/O-register m m. Dvs allt som behöver vara variabler får istället deklareras med en datatyp och kommer sålunda att beredas plats i processorns RAM-minne.

Programexempel:

```
include <avr/io.h>

uint8_t data;
uint16_t result;
```

```
PORTB = data;
...
```

Vilket i assembler skulle kunna motsvaras av:

```
  .dseg
data:
  .byte 1
result:
  .byte 2

  .cseg
  lds r16,data
  out PORTB,r16
...
```

Som C-programmerare behöver man alltså inte bekymra sig om hur vilka generella register som kommer att användas eller om och när de behöver sparas på stacken. Detta sköter C-kompilatorn.

2.4 I/O-register

Alla register som inte är generella register (generella register är R0-R31) är s k I/O-register. Detta även om de inte har någon direkt koppling till en I/O-port. Processorn har en stor mängd sådana I/O-register, alla med en specifik funktion. Exakt vad respektive register heter och vilken funktion det har finns angivet i processor-manalen. I assembler behöver man ofta använda generella register för att överföra information eller konstanter till ett I/O-register.

För att t ex sätta datariktningen för vissa bitar i ett I/O-register skriver man i assembler:

```
  ldi r16,(1<<PB2)|(1<<PB1)
  out DDRB,r16
```

Detta skulle i ett C-program motsvaras av:

```
  DDRB = (1<<PB2)|(1<<PB1);
```

C-kompilatorn kommer förstås att översätta C-koden till assembler som då använder något generellt register, men som C-programmerare behöver man inte bry sig om vilket generellt register det blir.

Två andra vanliga bitmanipulationer är vanligt förekommande. Man vill kunna 0-ställa eller 1-ställa vissa bitar i ett I/O-register utan att påverka övriga bitar i registret.

Skillnaden mellan assembler och C är inte så stor. I C gör man som i assembler, dvs man or:ar in 1:or och and:ar in 0:or. Antag att man vill 0-ställa `PB3` i `DDRB` utan att påverka övriga bitar. I ett C-program kan man då skriva:

```
  DDRB = DDRB & ~(1<<PB3);
```

Dvs, and-operationen (ampersand-tecknet &) utförs mellan DDRB och inversen (tilde-tecknet ~) av en 1:a på bitplats PB3 (dvs binärt blir det 11110111), och resultatet av det tilldelas sedan till DDRB. Detta kan också skrivas i en kortare variant enligt:

```
DDRB &= ~(1<<PB3);  // in assembler cbi DDRB,PB3
```

Antag att man vill 1-ställa PB2 i DDRB utan att påverka övriga bitar. I ett C-program kan man då skriva:

```
DDRB = DDRB | (1<<PB2);
```

Dvs, or-operationen (pipe-tecknet |) utförs mellan DDRB och det datamönster som bildas av (1<<PB1) (dvs binärt 00000100), och resultatet tilldelas sedan till DDRB. Kortvarianten av detta blir då:

```
DDRB |= (1<<PB2);  // in assembler sbi DDRB,PB2
```

2.5 Tabeller och pekare

Det är vanligt förekommande att man använder tabeller för att lagra information. I C-programmering, liksom i assembler, är pekare en vanlig metod för att indexera i tabeller. Antag följande assembler-kod:

```
  .dseg
TEXT:
  .byte 5
  .cseg

  ldi XH,HIGH(TEXT)
  ldi XL,LOW(TEXT)

  ld  r16,X+
  out PORTB,r16
```

Detta skulle i C-programering kunna motsvaras av:

```
char TEXT[5];

PORTB = *TEXT;
TEXT++;
```

I C-programmet ovan deklareras tabellen (eller arrayen) TEXT till att vara 5 byte stor. TEXT blir på så sätt också en pekare (alltså en adress) i C. Man kan komma åt adressens innehåll och ”peka” med TEXT genom att skriva en asterisk (*) framför pekaren. Därefter räknas pekaren upp till nästa position med TEXT++.

Antag att man vill ha en funktion som skriver ut alla tecken i TEXT på PORTB fram till dess att NUL-tecknet (en 0:a) påträffas. Det skulle kunna göra på följande sätt i C:

```
char TEXT[5];
```

```
void print_str(char* str) // argument to function is a pointer
{
  while (*str)            // as long as *str is not zero
    {
      PORTB = *str;       // set PORTB to *str
      str++;              // increase pointer
    }
}

...

print_str(TEXT);          // call function with TEXT pointer
```

I funktionshuvudet för `print_str` deklareras alltså en pekare (`str`) av typen `char`. Det sker genom att datatypen `char` efterföljs av en asterisk (`*`). Dock blev `TEXT` också en pekare i deklarationen `char TEXT[5]` utan att `char` efterföljdes av en asterisk (`*`). Det beror i det fallet på att `TEXT` består av flera stycken `char` efter varandra, och enda sättet att komma åt alla blir då via en pekare, så därför väljer C-kompilatorn att då göra en pekare av `TEXT`.

Hade man däremot deklarerat `TEXT` så att den bara består av en stycken av den aktuella datatypen (dvs `char`) så här:

```
char TEXT;
```

Då blir `TEXT` inte en pekare, utan helt enkelt bara en `char`.

Man kan dock ta reda på vilken adress `TEXT` finns på i det fallet, dvs man kan ta fram pekaren för `TEXT`, t ex på följande sätt:

```
char TEXT;
char* t_pointer = &TEXT;
```

I C-koden ovan deklareras `TEXT` som en `char`. Adressen (dvs pekaren) till `TEXT` får man fram genom ampersand-tecknet (`&`) framför `TEXT` (på nästa rad). Därefter tilldelas `char`-pekaren `t_pointer` den adressen (pekaren).

Nu skulle man alltså kunna tilldela ett tecken till `TEXT` via pekaren `t_pointer`, på följande sätt:

```
char TEXT;
char* t_pointer = &TEXT;

*t_pointer = 'A';      // now TEXT = 'A'
```

Detta med pekare i C kan vara lite svårt att ta till sig vid en första genomläsning. Dock brukar det underlätta om man tidigare har använt pekare i assembler, och att man har förstått det vill säga.

Pekare i C är ett dock ett smidigt sätt att åstadkomma effektiv programmering. Mer information om pekar kan man hitta bland annat vid `https://www.tutorialspoint.com/cprogramming/c_pointers.htm`

2.6 Kommentarer

Kommentarer i C-kod kan skrivas på två sätt. Dels som ett stycke då kommentaren inleds med `/*` och avslutas med `*/`, och dels som avslutning på en rad då kommentaren inleds med `//` och fortsätter till slutet av raden.

Programexempel:

```
/* The following two lines of code will set data direction
   PB3-0 as outputs and the value 12 to PORTB */
   DDRB = 0x0F;
   PORTB = 12;

   DDRA = 0xFF;   // Data direction for PORTA is all outputs
   PORTA = 0xA3;  // Value to PORTA is 0xA3
```

2.7 Timers

2.7.1 Pre-scaler och jämförelseregister

Processorn ATmega328 har tre inbyggda timers (timer 0, 1 och 2), vilka egentligen bara är räknare som kan fås att fungera på lite olika sätt. Dessa timers kan räkna tiden (egentligen klockcykler) i olika takt via något som kallas pre-scalers. En pre-scaler tar processorns grundfrekvens och delar den med ett av flera fasta värden (t ex 8, 64, 256 eller 1024) och låter den resulterande frekvensen klocka själva timern. På så sätt kan en timer fås att gå i en viss takt, och alla timers kan dessutom gå i olika takt.

Timerns räknarvärde kan jämföras med innehållet i ett visst jämförelseregister och när de är lika så kan man få timern att starta om vid 0 och samtidigt orsaka ett avbrott. Detta kallas för att timern jobbar i CTC mode (Clear Timer on Compare match). För att få detta att hända måste en timers relevanta I/O-register konfigureras på rätt sätt. För mer information om detta hänvisas till processorns datablad, avsnitten ”Timer/Counter0”, ”Timer/Counter1” samt ”Timer/Counter2”.

Timer 0 och Timer 2 är 8 bitar stora, dvs de kan räkna mellan 0 - 255. Timer 1 är 16 bitar stor och kan således räkna mellan 0 - 65535.

Antag följande scenario. Processorn klockas med en frekvens av 16 MHz, och vi vill få någon timer att orsaka ett avbrott 100 gånger per sekund. Vilken timer ska användas (8 eller 16 bitar), hur ska timerns pre-scaler konfigureras och vilket jämförelsevärde behöver användas för att avbrottet ska komma exakt 100 gånger per sekund? I alla fall så exakt det går att åstadkomma.

I princip behöver man ta grundfrekvensen (16 MHz), dela med önskad avbrottsfrekvens (100 Hz) och sedan dela med något pre-scalervärde (8, 64, 256, eller 1024) och resultatet ska då (helst) bli ett heltal. Om det blir ett heltal så blir det exakt rätt resulterande avbrottsfrekvens. Om det inte med någon kombination går att få exakt, så får man ta bästa närmevärde. Resultatet (minus ett) är alltså det som ska placeras i jämförelseregistret. Alltså:

16000000 / 100 / 1024 = 156.26 (inte heltal)

16000000 / 100 / 256 = 625 (heltal!)

Heltalet 625 ryms dock inte inom 8 bitar, så därför måste en 16-bitars timer användas. Dvs, 625-1=624 placeras i timerns jämförelseregister, och pre-scalerinställningen sätts till 256. Om man hade nöjt sig med ungefär 100 Hz så hade värdet 155 (156-1) kunnat placeras i jämförelseregistret till en 8-bitars timer med pre-scalerinställningen 1024.

Timerns mode (t ex CTC) och pre-scalerinställing görs i ett (eller flera) register som heter `TCCR*` (Timer Counter Control Register), där `*` byts ut mot något annat beroende på vilken timer som används.

Timerns jämförelsevärde sätts i ett register som heter `OCR*` (Output Compare Register), där `*` byts ut mot något annat beroende på vilken timer som används.

För mer information om detta hänvisas till processorns datablad, avsnitten ”Timer/Counter0”, ”Timer/Counter1” samt ”Timer/Counter2”

Programexempel:

```
/* Timer1, CTC mode, pre-scaler=256 */
TCCR1B = (1<<WGM12)|(1<<CS12);
/* Timer1 compare value : 624 */
OCR1A = 624;
```

2.7.2 Timeravbrott

En timer kan orsaka avbrott vid olika händelser. T ex då timern uppnått ett visst jämförelsevärde eller då timern nått sitt maximala värde och slår om till noll. Typiskt så är bara den ena av dessa händelser aktuell.

För att det ska bli ett avbrott vid önskad händelse så måste vissa bitar 1-ställas i ett register som heter `TIMSK1` (Timer/Counter Interrupt Mask Register). För mer information om detta hänvisas till processorns datablad, avsnitten ”Timer/Counter0”, ”Timer/Counter1” samt ”Timer/Counter2”.

Programexempel:

```
/* Timer1, enable interrupt on compare match */
TIMSK1 = (1<<OCIE1A);
```

2.8 Generell struktur för C-program

Den generella strukturen för ett C-program skulle kunna göras enligt följande:

```
/* include necessary libraries */
#include <...>
#include <...>

/* define variables and constants */
uint8_t ...
```

```
const uint8_t ...

/* forward declarations */
int function1(...);
int function2(...);

/* declare functions and interrupt service routines */
int function1(...)
{
  function2(...);
  ...
}

int function2(...)
{
  function1(...);
  ...
}

ISR(INTERRUPT_vect)
{
  ...
}

/* main program */
int main(void)
{
  ...
  while(1)
  {
    ...
  }
}
```

3 Uppgifter

3.1 Uppgift 1 : Digitalur med internt avbrott

Konstruera ett program som visar timmar, minuter och sekunder för ett 24-timmars digitalur och visar tiden på LCD-displayens första rad i formatet ”HH:MM:SS”.

Huvudprogrammet kommer här bara att vänta på ett timer-avbrott, 1 gång i sekunden. Avbrottets uppgift blir att göra två saker. Räkna upp tiden, och därefter visa tiden.

Uppräkningen av tiden görs lämpligen i en funktion som använder separata variabler för timmar, minuter och sekunder.

3.1.1 Delmoment

1. Gör en funktion `void lcd_init()` som initierar LCD:n
2. Gör en funktion `void lcd_home(int line)` som sätter markören i början på raden `line`. Dvs om `line = 1` ska markören sättas i början på LCD:ns första rad och om `line = 2` ska markören sättas i början på LCD:ns andra rad.
3. Gör en funktion `void lcd_ascii(char ch)` som skriver ut tecknet `ch` till LCD:n.
4. Gör en funktion `void lcd_print(char* str)` som skriver ut tecken för tecken i en NUL-terminerad sträng `str` fram till NUL-tecknet.
5. Gör en funktion `void time_set(uint8_t h, uint8_t m, uint8_t s)` som initierar variablerna för timmar minuter och sekunder till argumenten `h`, `m` respektive `s`.
6. Gör en funktion `void time_tick()` som räknar upp tiden för timmar, minuter och sekunder, med en sekund.
7. Gör en funktion `void printf_time()` som

   a. sätter markören i början på LCD:ns första rad

   b. formaterar tiden i variablerna för timmar, minuter och sekunder till en NUL-terminerad sträng i formatet ”HH:MM:SS”

   c. skriver ut den NUL-terminerade strängen på LCD:n
8. Gör en funktion `void timer1_init()` som initierar timer1 i processorn till att ge ett avbrott 1 gång i sekunden.
9. Gör en avbrottsrutin `ISR(TIMER1_COMPA_vect)` som räknar upp klockans tid med en sekund och skriver ut tiden på LCD:n.
10. Gör ett huvudprogram `main` som väntar på avbrott.

3.2 Uppgift 2 : Voltmeter

Tillsammans, dvs samtidigt, med klockan i uppgift 1, ska du konstruera en voltmeter som omvandlar knapparnas analoga värde och visar spänningen i volt på LCD-displayens andra rad. Dvs, samtidigt som klockan tickar på, och tiden visas på LCD:ns första rad, ska det vara möjligt att trycka på någon av knapparna och då visa vilken analog spänning den knappen skickar in till processorns A/D-omvandlare.

3.2.1 Delmoment

1. Gör en funktion `void adc_init()` som aktiverar A/D-omvandlaren med AVcc (5 volt) som referensspänning, ett vänsterjusterat värde för 8-bitars omvandling och en prescaler på 128.
2. gör en funktion `uint8_t adc_read()` som gör en A/D-omvandling och returnerar omvandlat värde.
3. Gör en funktion `void printf_volt()` som

   a. sätter markören i början på LCD:ns andra rad

   b. formaterar det A/D-omvandlade värdet till en NUL-terminerad sträng på formatet ”V,VV”, dvs en spänning med två decimaler. Även om man i slutändan vill ha ett decimaltal i utskriften så ska inte beräkningen (omvandlingen från A/D-värde till spänning) göras med decimaltal (dvs flyttal). En spänning på t ex 3,25 volt kan i beräkningen representeras som heltalet 325 (trehundratjugofem) men att man själv får stoppa in decimalkomma på rätt plats vid formateringen.

   c. skriver ut den NUL-terminerade strängen på LCD:n
4. Förändra huvudprogrammet `main` så att det kontinuerligt skriver ut A/D-omvandlarens spänning och väntar 200 ms mellan varje utskrift.

Sannolikt kommer det resulterande programmet inte att fungera bra. Dvs, utskrifterna till LCD:n från digitaluret och volmetern kommer att blandas om vartannat. LCD:n är ju en gemensam resurs som vi måste skriva till på ett kontrollerat sätt i god ordning för att det ska fungera.

Så hur kan vi får det att fungera på ett enkelt sätt? *Tips*, det handlar om att tillåta avbrott vid rätt tillfälle.

4 Versionshistorik

Version	Datum	Kommentar
1.0	2020-05-17	Publik version
1.1	2020-05-24	Korrektion av vektortabell sid 9
1.2	2021-06-05	Korrektion av register `TCCR1A` till `TCCR1B` sid 17, samt Uppgift 2, delmoment 3, flyttal **ska inte** användas sid 20

—o-O-o—