

C-programmering på AVR ATMega16

Anders Nilsson

2026 version 1.0

Innehåll

1	ATMega16	3
2	Programspråket C	4
2.1	Programbibliotek	4
2.1.1	<avr/io.h>	4
2.1.2	<util/delay.h>	5
2.1.3	<avr/interrupt.h>	5
2.1.4	<avr/pgmspace.h>	7
2.2	Datatyper	7
2.2.1	Volatile	8
2.3	Register och RAM	8
2.4	Kommentarer	9
2.5	Timers	9
2.5.1	Pre-scaler och jämförelseregister	9
2.5.2	Timeravbrott	10
2.6	Generell struktur för C-program	11
3	Versionslog	12

1 ATMega16

Den här skriften är gjord med utgångspunkt från mikrokontrollern ATMega16. Dock så går själva C-programmeringen att tillämpa på många andra av AVR-familjens mikrokontrollrar genom att byta till rätt namn på register, databitar och dylikt för den mikrokontroller man använder. Dessa delar framgår i databladet för respektive mikrokontroller.

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 C-programmering på AVR.

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 millisekunder, och med `_delay_us(y)` väntar programmet i y mikrosekunder. 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 deklarerats 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<<PB0);    // PB0 as output, PB1-PB7 as input
    while(1)
    {
        PORTB = 1;      // Generate an output ...
        _delay_ms(500);  // ... on PB0 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:

2 Programspråket C

Vektornamn	Betydelse
ADC_vect	ADC Conversion Complete
ANA_COMP_vect	Analog Comparator
EE_RDY_vect	EEPROM Ready
INT0_vect	External Interrupt 0
INT1_vect	External Interrupt 1
INT2_vect	External Interrupt 2
SPI_STC_vect	SPI Serial Transfer Complete
SPM_RDY_vect	Store Program Memory Ready
TIMER0_COMP_vect	Timer/Counter0 Compare Match
TIMER0_OVF_vect	Timer/Counter0 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
Timer2_COMP_vect	Timer/Counter2 Compare Match
Timer2_OVF_vect	Timer/Counter2 Overflow
TWI_vect	2-wire Serial Interface
USART_RXC_vect	USART, Rx Complete
USART_TXC_vect	USART, Tx Complete

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 deklaras 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]	float
double	32 bitar	[1.175494e-38, 3.402823e+38]	

Datatypernas namn är standardiserade i C. Däremot är deras motsvarande storlek och talområde implementations- och plattformsb beroende. 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 Volatile

En variabel kan ibland behöva deklarerats som `volatile`. Det talar om för kompilatorn att variabelns värde kan tänkas förändras även om kompilatorn inte ser hur det går till. T ex om variabeln bara läses i en huvuloop men att dess värde skrivs i en avbrottsrutin. Om variabeln då inte deklarerats som `volatile` så kan kompilatorn eventuellt optimera bort variabelns betydelse i programmet.

```
volatile uint8_t a = 7;

void main(void)          ISR(INT0_vect)
{                          {
    if (a >= 10) {          a = a + 1;
        PORTB = a;        }
    }
}
```

Om variabeln `a` i scenariot ovan inte vore deklarerad som `volatile`, skulle kompilatorn kunna ta bort `if`-satsen eftersom `a` (ur kompilatorn synvinkel) alltid har värdet 7 och därmed uppfylls aldrig villkoret (`a >= 10`). Med `volatile` förstår dock kompilatorn att `a` kan förändras (i detta fall via avbrottsrutinen `ISR`) och därmed får `if`-satsen vara kvar.

2.3 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 deklarerats 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 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 */
PORTB = 0x0F;
PORTB = 12;

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

2.5 Timers

2.5.1 Pre-scaler och jämförelseregister

Processorn ATmega16 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 jämförelsregister 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 jämförelsregistret. 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 jämförelsregister, och pre-scalerinställningen sätts till 256. Om man hade nöjt sig med ungefär 100 Hz så hade värdet 155 ($156-1$) kunnat placeras i jämförelsregistret till en 8-bitars timer med pre-scalerinställningen 1024.

Timerns mode (t ex CTC) och pre-scalerinställning 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:

```
/* Timer2, CTC mode, pre-scaler=1024 */
TCCR2 = (1<<WGM21) | (0<<WGM20) | (1<<CS22) | (1<<CS21) | (1<<CS20);
/* Timer2 compare value : 155 */
OCR2 = 155;
```

2.5.2 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 TIMSK (Timer/Counter Interrupt Mask Register). För mer information om detta hänvisas till processorns datablad, avsnitten "Timer/Counter0", "Timer/Counter1" samt "Timer/Counter2".

Programexempel:

```
/* Timer2, enable interrupt on compare match */  
TIMSK = (1<<OCIE2);
```

2.6 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 ...  
  
/* declare functions and interrupt service routines */  
  
int function1(...) {  
    ...  
}  
  
ISR(INTERRUPT_vect) {  
    ...  
}  
  
/* main program */  
int main(void) {  
    ...  
    while(1) {  
        ...  
    }  
}
```

3 Versionslog

2026 version 1.0	Ursprunglig version
------------------	---------------------

—o-O-o—