Datorteknik TSEA57
Le1

C-programmering med AVR

LINKÖPINGS UNIVERSITET

Datorteknik Le1 : Agenda

- C på AVR
- Generell struktur för C-program
- Programbibliotek för AVR
- Datatyper
- Generella register och RAM
- I/O-register och bitmanipulering
- Tabeller och pekare
- Timers + timeravbrott
- Lab 5

C på AVR

- C är ett relativt litet högnivåspråk, som också kan jobba på en ganska låg nivå, vilket gör att det är vanligt förekommande i in byggda system.
- C (jämfört med assembler) underlättar språkkonstruktioner såsom rekursivitet (funktioner anropar sig själva) och dynamisk minnesallokering, men bör undvikas på en AVR då det är svårt att håll koll på den mängd arbetsminne som går åt.
- C ger också möjlighet till flyttalsberäkningar, dvs beräkningar med decimaltal, vilket dock kommer att implementeras med mjukvara på en AVR, då den saknar hårdvarustöd för flyttal. Dvs, beräkningar med flyttal kommer att ta relativt lång tid att utföra.

Generell struktur för C-program

```
/* 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)
  {
    ...
  }
}
```

Programbibliotek för AVR : <avr/io.h>

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

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

Därefter skulle man kunna använda det, t ex så här:

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

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

Programbibliotek för AVR : <util/delay.h>

Programbiblioteket `delay.h` deklarerar funktioner för busy-wait-loopar. Funktionerna är:

```
void _delay_ms(double __ms)      // wait milliseconds
void _delay_us(double __us)      // wait microseconds
```

För att bygga en vänteloop med korrekt längd måste kompilatorn få reda på processorns klockfrekvens, som deklareras med konstanten `F_CPU` enligt följande programexempel:

```
#define F_CPU 16000000UL    // 16 MHz clock
#include <avr/io.h>
#include <util/delay.h>

void main()
{
  DDRB = (1<<PB2);          // PB2 as output
  while(1)
  {
    PORTB = (1<<PB2);       // Generate en output ...
    _delay_ms(500);         // ... on PB2 with ...
    PORTB = 0;              // ... a frequency of ..
    _delay_ms(500);         // ... 1 Hz
  }
}
```

Programbibliotek för AVR : <avr/interrupt.h>

Programbiblioteket `interrupt.h` inkluderar funktionalitet för avbrott. Nedan några utvalda avbrottsvektorer:

```
INT0_vect          External Interrupt Request 0
INT1_vect          External Interrupt Request 1
...
TIMER2_COMPA_vect  Timer/Counter2 Compare Match A
TIMER2_COMPB_vect  Timer/Counter2 Compare Match B
...
TIMER1_COMPA_vect  Timer/Counter1 Compare Match A
TIMER1_COMPB_vect  Timer/Counter1 Compare Match B
...
TIMER0_COMPA_vect  Timer/Counter0 Compare Match A
TIMER0_COMPB_vect  Timer/Counter0 Compare Match B
...
ADC_vect           ADC Conversion Complete
...
```

Dessutom deklareras funktioner för att påverka I-flaggan i statusregistret SREG:

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

Programexempel:

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

ISR(ADC_vect)      // Execute when ADC complete
{
  ...
}

void main()
{
  // Code to init A/D converter
  // Code to activate interrupts
  // Code to start A/D conversion
  while(1)
  {
    // Code for waiting for interrupt
    // and conversion result
  }
}
```

Programbibliotek för AVR : <avr/pgmspace.h>

Programbiblioteket `pgmspace.h` deklarerar funktioner för läsning från programminnet (Flash-minnet):

```
uint8_t pgm_read_byte(address)   // Read byte
uint16_t pgm_read_word(address)  // Read word
```

Det blir också möjligt att lägga in tabeller i program-minnet med en konstant-deklaration tillsammans med ordet `PROGMEM`.

```
const uint8_t TABLE[] PROGMEM = {...}
```

Programexempel:

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

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

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

  result = pmg_read_byte(&SQUARE[data]);

  while(1);
}
```

Datatyper

Relevanta heltals-datatyper vid C-programmering för AVR:

Datatyp	Storlek	Talområde AVR	Likvärdig
char	8 bitar	[0, 255]	uint_8t
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

Om man använder t ex `char` eller `uint8_t` spelar ingen roll. Båda betyder samma sak, man `uint8_t` visar via sitt namn lite tydligare att det är ett heltal (int) utan teckan (u för unsigned) och har storleken 8 bitar.

Relevanta flyttals-datatyper vid C-programmering för AVR:

Datatyp	Storlek	Talområde AVR	Likvärdig
float	32 bitar	[1.175494e-38, 3.402823e+38]	
double	32 bitar	[1.175494e-38, 3.402823e+38]	float

Dessutom finns datatypen `void` som egentligen betyder obestämd datatyp eller ingen datatyp. Den används typiskt när en funktion inte tar eller returnerar några argument.

Datatyper : type casting

Vid beräkningar med olika datatyper inblandade kan det vara nödvändigt att styra datatypen för resultatet, med s k type casting:

```
uint8_t  inches;
float centimeters;

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

Då variabeln `inches` är ett heltal kommer beräkningen att utföras med ett heltalsresultat, om man inte styr inches (med type casting) till att tolkas som en float.

Utan type casting kommer förvisso resultatet i `centimeters` att fortfarande vara ett flyttal, men då med bara nollor i decimalerna.

Datatyper : type casting och division

Vid heltalsdivision riskerar man att förlora precision, eftersom det inte finns någon decimaldel.
Det medför att ordningen på operationerna får betydelse:

```
uint8_t a = 7;                                  uint8_t a = 7;
uint8_t b = a * 10 / 3;     // b = 23           uint8_t b = a / 3 * 10;     // b = 20
```

Man kan förstås tillfälligt, med type casting, förändra datatypen till float för att behålla precisionen och gör sig oberoende av ordningen:

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

Det innebör ju dock att flyttalsberäkningar måste utföras, och det är krävande för en AVR-processor.
Bättre är att hålla koll på ordningen och undvika `float` om möjligt.

Skulle man vara intresserad av den s k resten vid heltalsdivision så kan den tas fram via procent (%):

```
uint8_t a = 170;
uint8_t b = a / 100;        // b = 1
uint8_t c = a % 100;        // c = 70 (170 – 1*100 = 70)
uint8_t d = c / 10;         // d = 7
uint8_t e = c % 10;         // e = 0 (70- 7*10) = 0
```

Generella register och RAM

Generella register (`r0` till `r31`) används inte vid C-programmering. Dvs, inte av C-programmeraren men likväl av C-kompilatorn för att åstadkomma olika saker.

Allt behöver istället vara variabler som ligger i processors RAM-minne.

C-kod

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

uint8_t data;
uint16_t result;

PORTB = data;
...
```

Motsvarande assembler-kod

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

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

I/O-register och bitmanipulering

För att påverka bitar i ett I/O-register använder man lämpligen bitarnas namn:

C-kod

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

Motsvarande assembler-kod

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

Två vanliga bitmanipulationer är att `0`-ställa eller `1`-ställa vissa bitar i ett register.
Man brukar säga att man `and`:ar in `0`:or och `or`:ar in `1`:or.

Exempel: `0`-ställ `PB3` i `DDRB` utan att påverka övriga bitar:

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

Exempel: `1`-ställ `PB2` i `DDRB` utan att påverka övriga bitar:

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

Tabeller och pekare

Pekare används, som bekant, för att indexera i tabeller.

C-kod

```
char TEXT[5];

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

Motsvarande assembler-kod

```
     .dseg
TEXT:    .byte 5

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

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

Tabellen (arrayen) `TEXT` blir egentligen en pekare (alltså en adress) i C och adressens innehåll kan nås med en asterisk (*) framför pekaren. Dvs, man pekar ut innehållet.

Tabeller och pekare

Exempel: En funktion som ”skriver ut” en NULL-terminerad sträng på PORTB:

```
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
```

Argumentet `str` deklareras com en `char`-pekare (p g a efterföljande asterisk *).
`TEXT` blir också en `char`-pekare (trots ingen efterföljande asterisk *), eftersom `TEXT` består att flera `char` efter varandra, och enda sättet att komma åt dom är via en pekare.

Tabeller och pekare

Om `TEXT` deklareras som bara ett tecken, dvs inte en array, så blir det ingen pekare:

```
char TEXT;
```

Man kan dock tar reda på vilken adress `TEXT` finns på med ampersand-tecknet (`&`):

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

Man kan därefter tilldela ett tecken till `TEXT`, via pekaren `t_pointer`:

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

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

För mer information om pekare i C:
https://www.tutorialspoint.com/cprogramming/c_pointers.htm

Timers + timeravbrott

ATmega16 har tre inbyggda hårdvarutimers, timer `0`, `1`, och `2`, som egentligen bara är räknare.

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

För att få en timer att ”mäta” en viss tid och ge avbrott efter den tiden, behöver man vanligen använda ett prescaler-värde (8, 64, 256 eller 1024).

Principen är:
Grundfrekvens / önskad avbrottsfrekvens / prescalervärde => räknarvärde
Räknarvärdet bör helst vara ett heltal för att få så exakt avbrottsfrekvens som möjligt.

Exempel: Antag 16 MHz systemklocka, avbrottsfrekvens 100 Hz:
16000000 / 100 / 1024 = 156.25 (inte heltal)
16000000 / 100 / 256 = 625 (heltal!)

Dvs, en 16-bitars timer måste användas (eftersom 625 inte ryms inom 8 bitar):
Timerns jämförelseregister ska då laddas med 625-1=624

Timers + timeravbrott

En timer kan jobba i olika moder, och CTC-mode (Clear Timer on Compare) nollställer timern så att den börjar om när ett jämförelsevärde har uppnåtts.

Timerns mode konfigureras via ett kontrollregister (`TCCR*`) och jämförelsevärdet placeras i ett jämförelseregister (`OCR*`).

För att generera avbrott när timern uppnått jämförelsevärdet, behöver detta konfigureras i ett register som heter `TIMSK1` (Timer/counter Interrupt Mask Register).

Exempel:

```
/* Timer1, CTC mode, pre-scaler = 256 */
TCCR1A = (1<<WGM12)|(1<<CS12);

/* Timer1 compare value : 624 */
OCR1A = 624;

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

```
/* Timer1 Compare A match, Interrupt Service Routine */
ISR(TIMER1_COMPA_vect)
{
  ...
}
```

För mer information om timers, se datablad för ATmega16

Lab5

...

Tid för Frågor

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET