C coderen op de AVR ATmega

Een aantal praktische tips en voorbeelden voor goede codeertechnieken op de AVR ATmega microcontroller.

DE HAAGSE HOGESCHOOL

Jesse op den Brouw
De Haagse Hogeschool
6 december 2018
Versie 0.5b
J.E.J.opdenBrouw@hhs.nl

Inhoudsopgave

Voor suggesties en/of opmerkingen over dit document kan je je wenden tot J. op den Brouw, kamer D1.047, of je kunt email versturen naar J.E.J.opdenBrouw@hhs.nl.

Lijst van figuren

Listings

0 Hoe codeer je het beste op een AVR?

De vraag is heel simpel gesteld, maar het antwoord is natuurlijk niet zomaar te geven. De titel suggereert dat dit document het ultieme antwoord is op alle vragen die gesteld kunnen worden over het coderen op de AVR-serie. Dat is natuurlijk niet waar.

Dit document behandelt een aantal aspecten van het coderen van software. Het gaat hier om hoe code wordt gepresenteerd, hoe je bepaalde valkuilen kunt omzeilen maar niet zozeer om een daadwerkelijke implementatie van een algoritme. Daarnaast blijkt dat de AVR-controller een intrigerende, maar vooral lastige processor is voor beginnende programmeurs. Veel problemen komen voort uit het gebruik van de specifieke hardware, zoals timers en seriële poorten. Ook hier zijn tips over opgenomen.

Een aantal van deze tips komt voort uit de jarenlange colleges en practica die gegeven worden aan studenten van de opleiding Elektrotechniek aan De Haagse Hogeschool te Delft. Tijdens de lessen wordt gebruik gemaakt van Atmel Studio versie 6 (6.2.1563-SP2, GNU-C compiler 4.8.1) de STK500, de JTAGICE-mkII en een ATmega32A. Daarnaast is een eigen opsteekprint gemaakt met een LCD en potmeter voor gebruik van de ADC. Er is een website beschikbaar waarop het hele curriculum, inclusief PowerPoint-presentaties en practicumopdrachten is ondergebracht. Surf maar naar http://elektro.eduweb.hhs.nl/micprg/. Een kanttekening: deze website wordt niet meer bijgewerkt.

C is een geweldige taal, je kan er ontzettend veel mee doen als je een microcontrollerbordje programmeert. Maar C heeft één nadeel: het legt de programmeer geen regels op voor het netjes vormgeven van de code. Je kan het hele programma, met uitzondering van de preprocessorregels, als één lange regel aan de compiler aanbieden: niet leesbaar, wel compileerbaar. Wil je meer weten van totaal niet leesbare code, ga dan naar http://www0.us.ioccc.org/.

Een aantal van deze tips en tricks wordt ondersteund met AVR-assembler code. Het is raadzaam om op de hoogte te zijn van de werking van de diverse instructies. Het toont maar weer eens aan dat alleen kennis van C niet altijd genoeg is.

Dank gaat uit naar oud-collega Harry Broeders en collega Ben Kuiper voor hun kritische blik en interessante aanvullingen.

1 Denk eerst na voor je code gaat schrijven.

Da's natuurlijk een mooie binnenkomer, maar zeker niet onbelangrijk. Eerst nadenken over je probleem en vervolgens je code schrijven werkt veel beter dan "gewoon beginnen en zien tot hoever we komen". Beschrijf wat de code moet doen en ontwerp alvast een soort framewerk waarbinnen alles moet passen.

```
uint16_t haal_ADC_waarde(void) {
}

void start_ventilator(void) {
}

uint8_t is_klep_gesloten(void) {
}

typedef enum (Begin_st, Klep_dicht_st, Tapkraan_aan_st) state_t;
state_t toestand = Begin_st;

int main(void) {

    while (1) {
        switch (toestand) {
            case Begin_st:         break;
            case Klep_dicht_st:    break;
            case Tapkraan_aan_st:  break;
        }
    }
    return 0;
}
```

Listing 1: *Voorbeeld van een framework*

2 Schrijf duidelijke code.

Da's natuurlijk nog een leuke binnenkomer, maar ook zeker niet minder waar. C heeft een aantal zeer mooie schrijfwijzen, zoals de ++-operator en alle toekenningsvarianten. Constructies als:

```
for (int i=0,j=10; ++i<j-2; j++) {   ... }

list[i++]&=(*(pint++)|(1<<5));
```

zijn niet te volgen, het is beter om ze uit te schrijven. Dit levert weliswaar meer broncode op, maar de compilers zijn tegenwoordig zo goed, dat de resulterende assemblercode minimaal is. Anderzijds is

```
TIMSK|=0x04;
```

weer minder goed te lezen (en zeker minder portable) dan

```
TIMSK|=(1<<TOIE1);
```

3 Laat duidelijk zien op welke niveau de code thuis hoort.

Met niveau wordt het inspringen bedoeld (engels: indentation), gebruikmakend van de TAB-toets. De meeste editors hebben een tab-afstand van 4 of 8 spaties. Een functie zoals `main()` ligt op niveau 0; dit ligt aan het begin van de regel. De code in een functie begint dus altijd op niveau 1. Statements zoals `if`, `while` en `for` leveren een niveau extra op, zoals te zien is in het voorbeeld. De verschillende niveau's zijn goed te volgen.

```
int main() {                                // niveau 0

    ...
    if (flag==1) {                          // niveau 1
        while (PORTB&0x01) {                // niveau 2
            for (i=0; i<8; i++) {           // niveau 3
                PORTA = 0x01;               // niveau 4
                wait();                     // niveau 4
                PORTA = 0x00;               // niveau 4
                wait();                     // niveau 4
            }                               // niveau 3
        }                                   // niveau 2
        PORTA = 0x80;                       // niveau 2
    }                                       // niveau 1

    return 0;                               // niveau 1
}                                           // niveau 0

void wait() {                               // niveau 0

    int i;                                  // niveau 1

    for (i=0; i<30000; i++) {               // niveau 1
        nop();                              // niveau 2
    }                                       // niveau 1
}                                           // niveau 0
```

Listing 2: *Voorbeeld van code met inspringen*

4 Gebruik #define voor constanten en parameters die door de gebruiker zijn aan te passen.

Niets is zo heerlijk als door je hele programma het getal '10' te moeten vervangen, omdat je nu besloten hebt dat je array 20 tekens mag bevatten. Een simpele

```
#define BUF_LEN 10
```

bespaart een hoop ellende. Veel mooier is om gebruikersparameters (nou ja, voor degene die de code compileert en gebruikt) in een aparte header-file op te nemen die te laten includen.

```
#define F_CPU 3686400UL
#define PRESCALER 1024UL
#define FREQ 4UL
```

Listing 3: *Voorbeeld header file met User Preferences*

In een andere header-file (of C-file) wordt deze file dan ingelezen.

```
#include "user_pref.h"
#ifndef F_CPU
#define F_CPU 3686400UL
#warning F_CPU set to 3.6864 MHz
#endif

#define COUNT ((F_CPU)/((PRESCALER)*(FREQ)))
#define LOAD (65536-(COUNT))

ISR(...) {
    TCNT1 = LOAD;
    ...
}
```

Listing 4: *Gebruik van header file*

5 Gebruik enum voor variabelen die niet door de gebruiker zijn aan te passen en maar enkele waarden kunnen aannemen.

Typisch voorbeeld hiervan zijn de toestanden van een toestandsmachine. Je kan zelf een nieuw type aanmaken met `typedef`.

```
1 /* The states of the state machine */
2 typedef enum {Idle, Wait1, Fetch, Temp, Load, Store} state_t;
3
4 /* Startup state */
```

```
5  state_t state = Idle;
6
7  int main(void) {
8
9      /* Do forever */
10     while (1) {
11         switch (state) {
12             case Idle:
13             case Wait1:
14             case Fetch:
15             default:
16                 state = Idle;
17                 break;
18         }
19     }
20 }
```

Listing 5: *Voorbeeld van een toestandsmachine*

6 Regels voor naamgeving.

Ieder bedrijf, projectgroep, programmeur heeft ongetwijfeld zijn eigen stijl met betrekking tot het geven van namen, maar neem een aantal regels in acht:

I. Namen met leidende underscores zijn gereserveerd voor systeemdoeleinden en moeten niet gebruikt worden voor variabele-namen.

II. `#define` constanten moeten allemaal met HOOFDLETTERS geschreven worden.

III. Enum constanten moeten allemaal met HOOFDLETTERS geschreven worden, of beginnen met een Hoofdletter.

IV. Functie-, `typedef`-, en variabelennamen, als ook `struct`, `union`, en `enum` tag-namen moeten met kleine letters geschreven worden.

V. Typedeffed namen eindigen met `_t`.

VI. Lange namen kunnen opgedeeld worden met behulp van underscores of het gebruik van hoofdletters: `counter_for_indexing` of `counterForIndexing`. Dat laatste wordt de *camelCase* genoemd, waarbij de eerste letter met een kleine letter geschreven worden.

7 Gebruik duidelijke variabelennamen.

De beroemde éénletterige variabelennamen zijn natuurlijk uit den boze. Als lusteller misschien nog wel, maar voor de rest niet. De C-compiler kan variabelennamen tot 31 tekens verwerken[1], maar dat kost jou wel veel typewerk. Nog wat regels.

I. Vermijd namen die slechts in hoofdletters en kleine letters verschillen, zoals `foo` en `Foo`.

II. Vermijd namen die in bibliotheken voorkomen zoals `read` en `write`.

Hieronder een tweetal links:
http://www.ibm.com/developerworks/linux/library/l-clear-code/
http://www.doc.ic.ac.uk/lab/cplus/cstyle.html

8 Voeg commentaar toe aan de code.

– the source is the documentation – is een veel gehoorde uitspraak. En dat is jammer, want goede documentatie zorgt veel het sneller begrijpen van de code. Het is niet zinvol om achter elke regel die ene statement uit te leggen:

```
i++;        /* Verhoog i */
```

snappen de meesten van ons ook wel. Beter is het om een klein stukje code dat een afgebakende functie heeft, te begeleiden met een paar regels commentaar. Zie voorbeeld.

```
/*
 * This routine will bit-bang the least significant
 * bit of Port A, on which the device's clock line is
 * connected, for 8 times until the device will
 * respond with a reset (lsb on Port B). Then, the
 * restart command, a 1 on the msb of Port A, will be
 * asserted. After this the device will be in a
 * known (reset) state.
*/
while (PORTB&0x01) {
        for (i=0; i<8; i++) {
            PORTA = 0x01;
            wait();
            PORTA = 0x00;
            wait();
        }
}
PORTA = 0x80;
```

Listing 6: *Code met commentaar*

Aan de andere kant is het de kunst om je code zo te schrijven dat je geen commentaar nodig hebt door het gebruik van veel functies en duidelijke variabelenamen. Er zijn zeker wat mensen te vinden die dat ook ondersteunen, bv. http://apdevblog.com/comments-in-code/. Maar eigenlijk is het boek van Robbert C. Martin [2] nog wel de beste bron.

9 Doe foutafhandeling.

Ook jij maakt fouten. Dus zul je ze moeten afhandelen. Maar nog erger zijn je gebruikers. Niet gehinderd door enige kennis van zaken zullen deze personen alles proberen om je programma om zeep te helpen. Denk hier aan: invoer die niet voldoet aan gestelde eisen, zoals alleen cijfers invullen of een getal groter dan 0 en kleiner dan 1000, states waar je toestandmachine nooit in mocht komen, namen van files die niet bestaan. Het gebruik van de `if`-statement met goede afhandelingscode is hier natuurlijk voor bedoeld.

10 Integer datatypen

De AVR C-compiler kent een aantal integer datatypes:

`uint8_t`	8 bits unsigned (1 byte)	0 – 255
`uint16_t`	16 bits unsigned (2 bytes)	0 – 65535
`uint32_t`	32 bits unsigned (4 bytes)	0 – 4294967295
`uint64_t`	64 bits unsigned (8 bytes)	0 – 18446744073709551615
`int8_t`	8 bits signed (1 byte)	−128 – +127
`int16_t`	16 bits signed (2 bytes)	−32768 – +32767
`int32_t`	32 bits signed (4 bytes)	−2147483648 – +2147483647
`int64_t`	64 bits signed (8 bytes)	−9223372036854775808 – +9223372036854775807

In het bovenstaande staat het getal voor een exact aantal bits, de `u` staat voor unsigned. Zo is een variabele van het type `uint8_t` gegarandeerd een unsigned 8-bit integer (a.k.a. een byte). Het type `int32_t` is gegarandeerd een signed integer van 32 bits. Om deze typen te gebruiken met de header-file `inttypes.h` gebruikt worden.

Dit zijn niet de bekende types zoals `int` en `long`. Dat komt omdat de C-specificatie nogal losjes de grootten van de diverse integers beschrijft [3]. Dat kan wel eens problemen geven bijvoorbeeld als je met I/O-registers wil werken. Gebruik van de type `char`, `short`, `int` en `long` wordt daarom afgeraden. Zie https://en.wikipedia.org/wiki/C_data_types. Zie ook http://www.barrgroup.com/Embedded-Systems/How-To/C-Fixed-Width-Integers-C99.

De AVR is een zogenaamde 8-bitter. Dat houdt in dat de processor alle rekenkundige operaties in eenheden van 8 bits uitvoert. Verwerken van 16, 32 en 64 bits variabelen moet dus gebeuren in eenheden van 8 bits. Bij het optellen van twee 32-bits getallen zijn dus in totaal vier 8-bits optellingen nodig. Let daar op bij het gebruik van variabelen: zorg voor de variabelen met de minste geheugengrootte. Zo kan je voor het bewerken van een array prima een 8 bits unsigned variabele als lusvariabele gebruiken (als de array natuurlijk niet groter dan 256 elementen is). In de onderstaande listing worden twee unsigned 32-bits getallen opgeteld. Merk op dat ook het ophalen van de variabelen uit RAM behoorlijk wat executietijd kost, evenals het wegschrijven naar RAM.

Signed variabelen worden in de two's complement representatie opgeslagen. Alle moderne processoren ondersteunen (alleen nog) de unsigned en two's complement representaties.

```
Disassembly of section .text:

; uint32_t a, b, c;

; int main(void)
; {
;   c = a + b;
  7c:   40 91 60 00     lds r20, 0x0060
  80:   50 91 61 00     lds r21, 0x0061
  84:   60 91 62 00     lds r22, 0x0062
  88:   70 91 63 00     lds r23, 0x0063
  8c:   80 91 68 00     lds r24, 0x0068
  90:   90 91 69 00     lds r25, 0x0069
  94:   a0 91 6a 00     lds r26, 0x006A
  98:   b0 91 6b 00     lds r27, 0x006B
  9c:   84 0f           add r24, r20
  9e:   95 1f           adc r25, r21
  a0:   a6 1f           adc r26, r22
  a2:   b7 1f           adc r27, r23
  a4:   80 93 64 00     sts 0x0064, r24
  a8:   90 93 65 00     sts 0x0065, r25
  ac:   a0 93 66 00     sts 0x0066, r26
  b0:   b0 93 67 00     sts 0x0067, r27
  b4:   80 e0           ldi r24, 0x00   ; 0
  b6:   90 e0           ldi r25, 0x00   ; 0
  b8:   08 95           ret
; }
```

Listing 7: *32 bits optelling*

De AVR C-compiler ondersteunt ook de niet-standaard 24-bits typen `__int24` en `__uint24`. Dit is makkelijk te realiseren omdat de ATmega een 8-bitter is.

11 Float en double.

Op de AVR zijn de grootte en nauwkeurigheid van een float en double precies hetzelfde. Het maakt dus niet uit of je een variabele als `float` of als `double` declareert. De grootte is 32 bits (4 bytes) en de nauwkeurigheid is ongeveer 7 decimalen[1]. Het kleinste positieve getal dat je kan opslaan is ongeveer $1,18 \cdot 10^{-38}$. Het grootste positieve getal is ongeveer $3,402823 \cdot 10^{38}$. Zie ook http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_reg_usage. Noot: bij programmeren op een PC wordt aangeraden om altijd een `double` te gebruiken. Zie bijvoorbeeld https://bitbucket.org/HR_ELEKTRO/cpl01/wiki/double.md. Zie ook 13.

[1] De mantisse is 24 bits (waarvan er 23 worden opgeslagen), dus $^{10}\log 2^{24} = 7,2$ decimalen. Afhankelijk van het getal dat opslagen wordt is de nauwkeurigheid tussen de 6 en 9 decimalen.

12 Vermijd delen.

De AVR-microcontroller heeft geen hardware deler aan boord. Dat houdt in dat alle delingen in software moeten worden uitgevoerd. Zelfs de simpelste deling kost dan aanzienlijk veel klokpulsen. Uitzonderingen zijn delingen door machten van 2. Deze worden omgezet in schuifoperaties. De oplettende lezer begrijpt dat dit ook geldt voor de modulus-operatie. Zie ook 13.

13 Float versus long.

Is het gebruik van floats eigenlijk wel aan te raden op een 8-bitter? Het antwoord is: geef het een kans. Althans als je gebruik maakt van een wat recentere AVR GNU C-compiler. Het hangt helemaal af van de berekeningen die uitgevoerd moeten worden. In de onderstaande code worden drie min of meer identieke berekeningen uitgevoerd.

In de eerste reeks wordt een echte floating point optelling en vermenigvuldiging uitgevoerd, zie de regels 21 t/m 24. In de tweede reeks is de vermenigvuldiging omgezet in een deling met de omgekeerde waarde bij de deling, zie regels 28 t/m 31. De laatste reeks is een long integer variant van de eerste twee, zie de regels 35 t/m 38. Hier zijn een naast de optelling een vermenigvuldiging en deling nodig. Draaien van alle reeksen levert op dat de tweede en derde reeks ongeveer even lang duren en 2 keer zo lang duren als de eerste reeks.

Wat betreft de code-omvang wint de floating point ruim van de integer berekeningen: bijna 2 keer zoveel code. Het gebruik van de optimalisatievlag `-O2` helpt weinig (zeg maar gewoon: niets) want de routines voor vermenigvuldigen en delen zijn al geoptimaliseerd door de programmeurs. Zie ook 29.

```
1  /* compile with -O0 and -O2 successively */
2
3  #define nop() asm volatile ("nop" ::)
4
5  float  ftime = 13456.0;
6  float  fdiv = 3.6864;
7  float  recfdiv = 1.0/3.6864;
8  volatile float  fo;
9
10 long   ltime = 13456;
11 long   lmul = 10000;
12 long   ldiv = 36864;
13 volatile long   lo;
14
15 long i;
16
17 int main() {
18
19    /* Floating point multiply reciprocal */
20    nop();                      // -O2: 295655-359 (295296)
```

```
21      for (i=0; i<1000; i++) {
22          ftime=ftime+1.0;
23          fo = ftime*recfdiv;
24      }
25
26      /* Floating point divide */
27      nop();                          // -O2: 924887-295655 (629232)
28      for (i=0; i<1000; i++) {
29          ftime=ftime+1.0;
30          fo = ftime/fdiv;
31      }
32
33      /* Long integer equivalent */
34      nop();                          // -O2: 1560747-924887 (635860)
35      for (i=0; i<1000; i++) {
36          ltime=ltime+1;
37          lo = (ltime*lmul)/ldiv;
38      }
39
40      nop();
41      return 0;
42  }
```

Listing 8: *Voorbeeld float versus integer.*

Zoals gezegd hangt het ook af van de berekening. In het onderstaande codefragment worden twee vermenigvuldigingen uitgevoerd:

```
1  lo = ltime*3;   /* integer multiply */
2  fo = ftime*3;   /* float multiply */
```

De eerste berekening is een integer-vermenigvuldiging. De compiler is behoorlijk slim en voert niet een echte vermenigvuldiging uit (met de `mul`-instructie) maar gebruikt de formule:

```
1  lo = (ltime+ltime) + ltime;   /* multiply with 3 using adds */
```

Hieronder is de gegenereerde assemblercode te zien. Het eerste deel betreft de integer-vermenigvuldiging, het tweede deel de floating-point-vermenigvuldiging. De floating-point-vermenigvuldiging roept de library-routine `__mulsf3` aan die een aanzienlijk langere rekentijd vergt.

```
1     ;lo = ltime*3;
2    92:   40 91 60 00     lds r20, 0x0060   ; get integer variable ltime
3    96:   50 91 61 00     lds r21, 0x0061
4    9a:   60 91 62 00     lds r22, 0x0062
5    9e:   70 91 63 00     lds r23, 0x0063
```

```
  a2:   db 01           movw    r26, r22  ; make a copy of the integer
  a4:   ca 01           movw    r24, r20
  a6:   88 0f           add r24, r24      ; perform ltime = ltime + ltime
  a8:   99 1f           adc r25, r25
  aa:   aa 1f           adc r26, r26
  ac:   bb 1f           adc r27, r27
  ae:   84 0f           add r24, r20      ; ltime = ltime + copy_of_ltime
  b0:   95 1f           adc r25, r21
  b2:   a6 1f           adc r26, r22
  b4:   b7 1f           adc r27, r23
  b6:   80 93 6c 00     sts 0x006C, r24   ; write ltime to RAM
  ba:   90 93 6d 00     sts 0x006D, r25
  be:   a0 93 6e 00     sts 0x006E, r26
  c2:   b0 93 6f 00     sts 0x006F, r27
    ;fo = ftime*3;
  c6:   20 e0           ldi r18, 0x00     ; load float constant 3
  c8:   30 e0           ldi r19, 0x00
  ca:   40 e4           ldi r20, 0x40
  cc:   50 e4           ldi r21, 0x40
  ce:   60 91 64 00     lds r22, 0x0064   ; get float variable ftime
  d2:   70 91 65 00     lds r23, 0x0065
  d6:   80 91 66 00     lds r24, 0x0066
  da:   90 91 67 00     lds r25, 0x0067
  de:   0e 94 7d 00     call    0xfa    ; 0xfa <__mulsf3> (call multiply)
  e2:   60 93 70 00     sts 0x0070, r22   ; write ftime to RAM
  e6:   70 93 71 00     sts 0x0071, r23
  ea:   80 93 72 00     sts 0x0072, r24
  ee:   90 93 73 00     sts 0x0073, r25
  ...

000000fa <__mulsf3>:
  fa:   0b d0           rcall   .+22        ; 0x112 <__mulsf3x>
  ...                                     ; more code here
```

Listing 9: *Assemblercode van integer en float vermenigvuldigingen met 3.*

14 Maak gebruik van functies voor het aan- en uitschakelen van de hardwaresturing.

Code wordt beter leesbaar als er een aanroep wordt gedaan naar `pompAan()` in plaatst van `PORTB|=0x40`. De inhoud van `pompAan()` is dan zeer klein, maar dat moet niet uit maken. Als je optimalisatie `-O3` gebruikt heb je grote kans dat de compiler de functies helemaal niet aanmaakt maar de body van de functies plaatst op de plek van de aanroep (inline). Bij herhaald gebruik kost dit iets meer ruimte, maar levert wel een tijdbesparing omdat het aanroepen en terugkeren van de functie wordt vermeden.

```
#define POMP_BIT 6
```

```

void pompAan() {
    PORTB |= (1<<POMP_BIT);
}

void pompUit() {
    PORTB &= ~(1<<POMP_BIT);
}

int main() {

    if (s == Load) {
        pompAan();
    } else {
        pompUit();
    }

    return 0;
}
```

Listing 10: *Voorbeeld gebruik functies voor hardwaresturing*

15 Plaatsing van accolades.

Dit is altijd het gevecht tussen twee kampen: zij die de accolades achter `if`, `for`, `while` en `switch` zetten en zij die het eronder zetten. Beide zijn goed, maar houd je wel aan één vorm en ga ze niet door elkaar gebruiken. Nog een tip: ook `if`, `for`, `while`-statements met slechts één regel code moet je tussen accolades zetten. Da's handig voor later als je beslist om er nog wat regels code bij te plaatsen, want dán vergeet je ze natuurlijk.

```
/* Use with functions! */
void test()
{
    /* Variant 1 */
    if (a==b) {
        c=d;
        d=d+1;
    } else {
        c=c+1;
        d=c;
        k=k2;
    }

    /* Variant 2 with only one statement with {} */
    if (c==d)
    {
        a=b;
```

```
18      }
19      else
20      {
21          a=f;
22      }
23  }
24
25  int main(void) {
26
27      test();
28      return 0;
29  }
```

Listing 11: *Plaatsing van accolades*

16 Interruptroutines moeten zo snel mogelijk uitgevoerd worden.

Een interrupt is een onderbreking van het lopende programma. De hardware van de AVR signaleert dat er iets gebeurd is zoals een timer die over de kop gaat (overflow) of de ADC die klaar is met een nieuwe conversie. Als een Interrupt Service Routine (ISR) wordt uitgevoerd ligt het lopende programma stil. Als hier besturing in is geprogrammeerd, bijvoorbeeld van een robot, is het programma dus niet in staat de robot te sturen. ISR's moeten daarom zo snel mogelijk worden uitgevoerd. Dit betekent dat je bv. geen ingewikkelde berekeningen moet uitvoeren of de LCD-display moet aansturen, maar ook geen lus die blijft wachten tot een variabele is veranderd. Korte lusjes, bijvoorbeeld om een array te initialiseren is geen probleem. Probeer de executie-tijd van een ISR (ruim) onder de 100 μs te houden. Dit kan je prima testen met de simulator.

```
1   #define NUM_ADC_VALUES (10)
2
3   float x, y, angle;
4   uint16_t adc_val[NUM_ADC_VALUES];
5
6   ISR(TIMER1_COMPA_vect) {
7
8       uint8_t i;
9
10      /* BAD! Waits for bit change (waits till ADC conversion complete) */
11      ADCSRA |= (1<<ADSC);
12      while (ADCSRA & (1<<ADSC));
13
14      /* OK */
15      PORTB=PORTB^(1<<LED);
16
17      /* BAD ! Uses LCD. Takes a long time */
18      lcd_puts("In T/C1 ISR.\n");
19
```

```
20      /* BAD ! Takes a long time, are float operations */
21      x = cos(angle); y = sin(angle);
22
23      /* OK, it is a loop, but doesn't take a long time */
24      for (i=0; i<NUM_ADC_VALUES; i++) {
25          adc_val[i]=0;
26      }
27  }
28
29  int main() {
30
31      lcd_init();
32      ...
33      ...
34
35      return 0;
36  }
```

Listing 12: *ISR's moeten kort duren*

17 Volatile.

Een veel gebruikte oplossing voor het signaleren van een interrupt is de flag-variabele: een variabele wordt op één gezet in een ISR en getest en op nul gezet in `main()`. Dit is een prima methode.

De huidige compilers zijn echter erg slim. Een ISR wordt nooit aangeroepen door een software-instructie maar door een hardwaregebeurtenis. Een compiler kan dan beslissen om de flag helemaal weg te laten, immers de ISR wordt toch nooit aangeroepen en dus nooit uitgevoerd. Maar dan kan de compiler ook beslissen om de test-en-op-nul-zet actie in `main()` er uit te laten. De flag variabele wordt toch nooit één.

Om dit te voorkomen moet bij de declaratie van de flag het keyword `volatile` worden opgegeven wat zoveel wil zeggen als: compiler, blijf met je vingers van de variabele af want er gebeurt iets onder de motorkap waar jij geen weet van hebt.

Het nadeel is nu dat de compiler bij elke actie met die flag de waarde uit het RAM-geheugen moet ophalen waardoor langere executietijden ontstaan. Gebruik `volatile` daarom ook alleen als het nodig is. Als je in een ISR een `volatile`-variabele vaak nodig hebt, kan het soms slim zijn om die `volatile`-variabele te kopiëren naar een gewone, non-`volatile`-variabele en dan verder de gewone variabele gebruiken. Optimalisatie zorgt er voor dat de gewone variabele één keer geladen wordt en vervolgens in een register vastgehouden wordt tijdens de uitvoer van de ISR. Je codeert nu meer C-code, met als resultaat dat de gegenereerde machinecode kleiner is! Als je de optimalisatie van de compiler uitzet, heb je grote kans dat `volatile` niet nodig is omdat de compiler dan in een "domme" modus staat, maar er is geen garantie.

Zie als voorbeeld de code bij 19. Overigens is niet iedereen blij met `volatile`. Zie https://lkml.org/lkml/1996/5/2/87 voor opmerkingen van Linus Torvalds.

18 Let op het gebruik van gemeenschappelijke variabelen.

In de wereld van Operating Systems is er een probleem met variabelen die gemeenschappelijk gebruikt worden door twee of meer processen. Hier kan het updaten van variabelen door deze processen soms tot foutieve uitkomsten leiden. Maar de AVR is een processor waarop standaard geen OS draait. Dat betekent dat je alles in de hand hebt. Dus waar maken we ons zorgen om? Welnu, als je gebruik maakt van Interrupt Service Routines (ISRs), dan heb je eigenlijk een processor (de AVR) met daarop twee processen: de eigen `main()` en de ISR. De ISR wordt namelijk door een hardware-gebeurtenis gestart asynchroon met het hoofdprogramma. Terwijl het hoofdprogramma een variabele aan het aanpassen is, wordt de ISR aangeroepen die de variabele óók aanpast. Dat is vragen om ellende. Hier geldt ook voor: het is niet de vraag óf het gebeurt, maar wanneer!

In onderstaande code wordt de operatie op Port B in regel 33 atomair uitgevoerd. Dit gebeurt door middel van de `ATOMIC_BLOCK`-macro die te vinden is in de header-file `util/atomic.h`. Zie voor meer informatie http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html.

In feite schermt deze macro de port-operatie af met een `cli`/`sei`-paar. Direct gebruik van `cli`/`sei` is echter niet aan te raden omdat compiler-optimalisatie de executie-volgorde kan veranderen. Zowel `cli` als `sei` veranderen immers geen variabelen of geheugenplaatsen. Zie ook http://www.nongnu.org/avr-libc/user-manual/optimization.html.

```
1  #include <util/atomic.h>
2
3  ISR(TIMER1_COMPA_vect) {
4
5      /* ... more code ... */
6
7      /* toggle LED, this is atomically because the */
8      /* ISR cannot be interrupted */
9      PORTB=PORTB^(1<<LED);
10 }
11
12 int main() {
13
14     /* Initialization here... */
15
16     while {1) {
17         if (...) {
18             /* Bad!                                    */
19             /* Assembler code for next statement is:   */
20             /*    in   r24,PORTB                       */
21             /*    ori  r24,0xf0                        */
22             /*    out  PORTB,r24                       */
23             /* Things go wrong when interrupt occurs  */
24             /* directly after the in-instruction      */
25             /* because R24 now holds a COPY of PORTB! */
26             PORTB=PORTB|0xf0;
```

```
27          }
28
29          if (...) {
30              /* Good!                                    */
31              /* Make PORTB operation atomically          */
32              ATOMIC_BLOCK(ATOMIC_FORCEON) {
33                  PORTB=PORTB|0xf0;
34              }
35          }
36      }
37
38      return 0;
39  }
```

Listing 13: *Afschermen van gemeenschappelijke variabelen*

19 Testen van variabelen i.c.m ISR's.

In veel situaties wordt een ISR gebruikt om een bepaalde gebeurtenis te signaleren, bijvoorbeeld als de ADC klaar is met een conversie. Er wordt dan een gemeenschappelijke variabele gebruikt (een *flag*, niet te verwarren met de status flags in het Status Register) die gezet wordt in de ISR en die getest en gereset wordt in `main()`.

In de onderstaande code wordt de flag `adc_ready` op `ADC_IS_READY` gezet in de ISR van de ADC. Tevens wordt ook een kopie gemaakt van de ADC-waarde. Met de kopie wordt in het hoofdprogramma gerekend. Het is verstandig om in de ISR van de ADC een kopie te maken want dan heb je een "verse" waarde. Direct gebruik maken van de ADC-waarde in het hoofdprogramma is listig omdat de `ADCH`- en `ADCL`-registers niet door de ADC-hardware kunnen worden geschreven als er in een programma tegelijkertijd gelezen wordt [4]. Zie ook 18.

In de `while`-lus in `main()` wordt een kopie gemaakt van de flag en de flag wordt ook gelijk op `ADC_IS_NOT_READY` gezet. Deze operaties worden afgeschermd door een `ATOMIC_BLOCK` -macro. Bij de `if`-statement wordt dan de kopie getest. Merk op dat bij compiler-optimalisatie de waarde van `copy_adc_ready` in een CPU-register worden vastgehouden. Daarnaast worden de `enum`'s in een 8-bits variabele opgeslagen.

```
1  #include <avr/interrupt.h>
2  #include <stdint.h>
3  #include <util/atomic.h>
4
5  /* New type for ADC ready or not */
6  typedef enum ADCReadyValues {ADC_IS_NOT_READY, ADC_IS_READY} adc_ready_t;
7
8  /* ADC conversion complete flag */
9  volatile adc_ready_t adc_ready=0;
10 /* copy of ADC value */
11 volatile uint16_t adc_val=0;
```

```
12
13  ISR(ADC_vect) {
14
15      /* Flag ADC has a new value */
16      adc_ready = ADC_IS_READY;
17      /* Make a copy of the ADC value */
18      adc_val = ADC;
19
20      /* Start new conversion, use with simulator */
21      /* ADCSRA |= (1<<ADSC); */
22  }
23
24  int main() {
25
26      volatile int16_t temp=0;
27      adc_ready_t copy_adc_ready;
28
29      /* Set up ADC */
30      ADMUX = 0b00000000;
31      ADCSRA = (1<<ADEN)|(1<<ADSC)|(1<<ADATE)|(1<<ADIF)|
32      (1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
33
34      /* Free interrupts */
35      sei();
36
37      /* Do forever */
38      while (1) {
39
40          /* Make a copy of the ADC-is-ready and reset the flag */
41          ATOMIC_BLOCK(ATOMIC_FORCEON)
42          {
43              copy_adc_ready = adc_ready;
44              adc_ready = ADC_IS_NOT_READY;
45          }
46
47          /* Check if the ADC is ready using the copy of the status */
48          if (copy_adc_ready == ADC_IS_READY) {
49
50              /* More stuff, probably with the ADC */
51              /* Calculate temp from an AD592 temp-sensor */
52              temp = ((adc_val-559)*100)/204;
53          }
54
55          /* ... Do more stuff */
56      }
57
58      return 0;
59  }
```

Listing 14: *Atomaire acties*

In het volgende voorbeeld wordt een 16-bit variabele in een ISR bewerkt en in het hoofdprogramma getest. Zie onderstaande code. Deze code en tekst is overgenomen van http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html. De tekst en code is door de auteur bewerkt.

```
1  #include <inttypes.h>
2  #include <avr/interrupt.h>
3  #include <avr/io.h>
4  #include <util/atomic.h>
5
6  volatile uint16_t ctr;
7
8  ISR(TIMER1_OVF_vect) {
9      ctr--;
10 }
11
12 int main(void) {
13
14     uint16_t ctr_copy;
15
16     TIMSK = (1<<TOIE1);
17     TCCR1A = 0x00;
18     TCCR1B = 0x01; // 1;
19
20     ctr = 0x200;
21
22     sei();
23
24     /* WRONG !!!!!! */
25     while (ctr != 0);
26
27     /* Correct way of testing the 16-bit variable */
28     do {
29         ATOMIC_BLOCK(ATOMIC_FORCEON) {
30             ctr_copy = ctr;
31         }
32     } while (ctr_copy != 0);
33
34     return 0;
35 }
```

Listing 15: *Testen van een 16-bit variabele i.c.m. een ISR.*

In regel 25 wordt in een lus gewacht tot de variabele 0 is. Er is een kans het hoofdprogramma de wachtlus zal verlaten wanneer de variabele `ctr` de waarde `0xFF` bereikt.

Dit gebeurt omdat de compiler een 16-bits variabele niet atomair (als zijnde één eenheid) in een 8-bit CPU kan benaderen. Stel dat de de variabele is bijvoorbeeld `0x100`. De compiler test dan eerst de lage byte op `0`, die test slaagt. Daarna wordt de hoge byte getest, maar op dat moment wordt de ISR getriggerd en het hoofdprogramma wprdt onderbroken. De ISR zal de variabele van `0x100` naar `0xFF` verlagen en terugkeren naar het hoofdprogramma. Deze test nu de hoge byte van de variabele die (nu) ook 0 is zodat de lus beëindigt wordt.

De juiste manier van benaderen is te zien in de regels 28 t/m 32. Er wordt eerst een kopie gemaakt van de variabele en die kopie wordt getest. Het maken van de kopie wordt afgeschermd met een`ATOMIC_BLOCK`-macro.

Er zijn trouwens wel wat overwegingen te maken. Een en ander is is sterk afhankelijk van de situatie. Als de executie van een lus bijv. korter is dan het interrupt interval, dan gaat dit probleem nooit voorkomen (en dat is vaak zo). In andere situaties wil je juist niet de interrupt blokkeren, want je wil allicht de resultaten uit de interrupt bufferen en anderzijds is het soms ook niet zo erg om een meting te missen. (Met dank aan Ben).

20 Gebruik zo min mogelijk bronnen (resources).

Naast dat je natuurlijk een zo'n klein mogelijk programma moet coderen en zo min mogelijk RAM-geheugen moet gebruiken, is het verstandig om óók te besparen op hardwarebronnen. Een veel voorkomend gebruik is om twee timers te nemen als twee tijden (tijdspannen) moeten worden bijgehouden. Een voorbeeld hiervan is een led elke seconde te laten knipperen en de ADC elke tien seconden een conversie te laten doen. Veel van deze problemen kunnen prima met één timer worden opgelost. De truuk is om met één hardware-timer meerdere software-timers te maken.

Op de volgende pagina's is een typische toepassing geprogrammeerd. Timer/Counter 0 (T/C0) wordt gebruikt als tijdmeter. Elke 100^{ste} van een seconde wordt de Interrupt Service Routine (ISR) van T/C0 aangeroepen (de Compare Match ISR om precies te zijn). Eén teller houdt bij hoeveel 100^{sten} van een seconde voorbij zijn. Als dat er precies 100 zijn is 1 seconde tijd verstreken. Op dat moment worden twee andere tijden, te weten een tijdmeter van 3 seconden en één van 7 seconden aangepast. Is ook die tijd verstreken (het zijn dus twee onafhankelijke tijdmeters), dan wordt een actie ondernomen. In het voorbeeld worden twee leds getoggeld. Merk op dat T/C0 slechts een 8 bit telregister heeft. Het is bijvoorbeeld niet mogelijk om 10^{den} van seconden in één keer te tellen bij gegeven frequentie.

```
1 //Include the two needed header files
2 #include <avr/io.h>
3 #include <avr/interrupt.h>
4
5 // There are two time, at 3 and 7 seconds
6 #define TIME1 3
7 #define TIME2 7
8
```

```
// Check for CPU frequency set, or bail out
#ifndef F_CPU
#error Please set CPU frequency in the project options.
#endif

// Timer frequency is 100 Hz, prescaler and prescaler bits
#define FREQ 100UL
#define PRESCALER 1024
#define PRESCALER_BITS ((1<<CS02)|(1<<CS00))

// Nice trick if you need a debug point ;-)
#define nop() asm volatile ("nop" ::)

// The three time keepers.
uint8_t time1_tick = 0;
uint8_t time2_tick = 0;
uint8_t hundredths = 0;

// The Interrupt Service Routine for T/C0 compare match
ISR(TIMER0_COMP_vect) {

    // Increment hundredths of seconds
    hundredths++;
    if (hundredths == FREQ) {
        // We have one second now
        hundredths = 0;
        time1_tick++;
        if (time1_tick == TIME1) {
            // We have Time 1 now...
            time1_tick = 0;
            PORTB^=(1<<PB7);
        }
        time2_tick++;
        if (time2_tick == TIME2) {
            // We have Time 2 now...
            time2_tick = 0;
            PORTB^=(1<<PB6);
        }
    }
}

// Main, as in every C program
int main() {

    // Initialize T/C0 to Compare Match mode with ISR
    TIMSK = (1<<OCIE0);
    TIFR  = (1<<OCF0);
    TCCR0 = (1<<WGM01) | PRESCALER_BITS;
    OCR0  = ((F_CPU/PRESCALER)/FREQ)-1;
```

```
58
59      // Port B is leds output
60      DDRB  = 0xff;
61      PORTB = 0xff; // For STK500
62
63      // Free interrupts
64      sei();
65
66      while (1) {
67          // You can set a breakpoint here.....
68          nop();
69      }
70
71      // Yeah, yeah, never reached
72      return 0;
73  }
```

Listing 16: *Meerdere software timers d.m.v. een hardware timer*

21 Gebruik Flash ROM geheugen voor lange strings

Als je veel tekst hebt af te drukken, bijvoorbeeld via de RS232-interface of via de lcd, dan is het niet handig om de tekst ”gewoon” op te slaan als een string. De tekens van de strings moeten met het programma mee in de AVR geflashd worden, dus die tekens staan in de Flash-ROM. Als het programma start worden de tekens naar RAM gekopiëerd, nog voordat `main()` wordt aangeroepen. Aangezien de ATmega's niet zo heel veel RAM aan boord hebben, is het beter om de strings in ROM te laten staan en de tekens op te halen via de `pgm_read_*`-functies te benaderen. Documentatie is te vinden in [5]

```
1   /*****************************************
2   * Chip type           : ATmega32A
3   * Clock frequency     : 3.6864 MHz
4   *****************************************/
5   #include <avr/io.h>
6   #include <avr/interrupt.h>
7   #include <inttypes.h>
8   #include <avr/io.h>
9   #include <avr/pgmspace.h>
10
11  /* Note: define F_CPU in Configuration Options */
12  #define USART_BAUD_RATE 9600
13  #define USART_BAUD_CALC(USART_BAUD_RATE,F_CPU) \
14          ((F_CPU)/((USART_BAUD_RATE)*16l)-1)
15
16  /* Define strings in Program Memory (Flash) using PROGMEM */
17  char welcome1[] PROGMEM = "Welcome to AVR ATmega32A.......\r\n";
18  char welcome2[] PROGMEM = "Type some text to see the copy.\r\n";
```

```

/* Initialize the USART */
void usart_init(void) {
    /* Set baud rate */
    UBRRH = (uint8_t)(USART_BAUD_CALC(USART_BAUD_RATE,F_CPU)>>8);
    UBRRL = (uint8_t)USART_BAUD_CALC(USART_BAUD_RATE,F_CPU);

    /* Enable the transmitter and receiver, no interrupts */
    UCSRB = (1<<RXEN)|(1 << TXEN);

    /* Asynchronous 8N1 */
    UCSRC = (1 << URSEL) | (3 << UCSZ0);
}

/* Wait for a character reception */
char usart_getc(void) {
    /* Wait until receiver has data */
    while(!(UCSRA & (1 << RXC)));
    /* Return character */
    return UDR;
}

/* Transmit a character */
void usart_putc(char c) {
    /* Wait until transmitter is ready to send data */
    while(!(UCSRA & (1 << UDRE)));
    /* Send character */
    UDR = c;
}

/* Transmit a PROGMEM string */
void usart_puts_P(char *s) {

    uint8_t ch;
    while ( (ch = pgm_read_byte(s)) != '\0') {
        usart_putc(ch);
        s++;
    }
}

int main(void) {

    char c;

    usart_init(); /* init USART */

    /* Send greeting text */
    usart_puts_P(welcome1);
    usart_puts_P(welcome2);
```

```
68
69      /* Forever copy input to output */
70      while (1) {
71          /* Read character */
72          c = usart_getc();
73          /* If the character is a CR, send CR & LF */
74          if (c == '\r')
75              usart_puts_P(PSTR("\r\n"));
76          /* If the character is ESC, send nice info */
77          else if (c == '\033')
78              /* Use PSTR to use a string constant */
79              usart_puts_P(PSTR("You pressed the ESC-key!\r\n"));
80          /* just send the character */
81          else
82              usart_putc(c);
83      }
84      return 0;
85  }
```

Listing 17: *Gebruik van PROGMEM strings*

22 Let op het gebruik van veel RAM.

Al eerder is het gebruik van strings in RAM aan de orde geweest. Een AVR heeft al niet zo veel RAM aan boord. Natuurlijk kan je al het beschikbare geheugen gebruiken maar als de teller op groter dan 90% staat, kan je problemen krijgen. Het probleem zit 'm in de stack. De stack wordt gebruikt om terugkeeradressen van code die functies aanroepen op te slaan. De stack wordt ook, afhankelijk van compileroptimatisaties, gebruikt bij gegevensoverdracht (parameters) bij functie-aanroepen. Traditioneel "groeit" de stack van hoge adressen naar lage adressen. Daarbij kan het dus voorkomen dat de stack de data van de variabelen overschrijft. De stack "trashed" de data. Eén ander is natuurlijk afhankelijk van het gebruik van de stack, hoe diep genest de aanroepen zijn (probeer eens recursieve functies) en het gebruik van de RAM.

23 USART werkt niet met een ATmega32 out-of-the-box.

Out-of-the-box draait een ATmega32 (en andere) op de interne 1 MHz RC-oscillator. De USART werkt dan niet correct omdat de RX- en TX-clock teveel afwijking heeft, -7% (bij 9600 baud, zie [6]). Aangetoond kan worden dat 4% à 5% [citation needed] echt het maximum is. Oplossing: zet de interne RC-oscillator op 2, 4 of 8 MHz óf zet het U2X-bit in het UCSRA-register op 1 (hiermee wordt de transmissiesnelheid verdubbeld). Voor de geoefende lezer: ook een extern kristal van 1 MHz werkt dus niet!

24 Let op met compiler-optimalisatie

De compiler is een geavanceerd stuk software. Het neemt een enorme last van onze schouders door assemblercode te genereren zodat wij dat niet hoeven te doen. Eén van de mooie dingen die een compiler kan, is optimaliseren. De compiler kan codefragmenten herkennen en zo slimme code genereren.

Neem als voorbeeld onderstaande code. De programmeur is er stiekem van uit gegaan dat de `while`-lus automatisch wordt beëindigd als `i` groter wordt dan 32767, dat is namelijk het grootst mogelijk positieve gehele getal in 16-bits 2's complement notatie. Bij geen optimalisatie (optie `-O0`) wordt code gegenereerd die test of de waarde van `i` inderdaad negatief wordt voordat de lus wordt gestopt. Maar zoals gezegd is een compiler een slim stukje software. Bij optimalisatie `-O2` verwijdert de compiler de gehele `while`-constructie, inclusief variabele `i`, en zal `x` tot in de eeuwigheid met één verhoogd worden. De reden hiervoor is dat de compiler "ziet" dat `i` op 30000 wordt geïnitialiseerd en alleen verhoogd wordt. Variabele `i` is dus altijd groter dan 0 en dan is de conditie voor de `while`-lus altijd waar.

Merk op dat als `x` niet als `volatile` was gedefinieerd, de compiler ook die had verwijderd.

```
1  #include <stdio.h>
2
3  volatile uint16_t x = 0;
4
5  int main(void) {
6
7      int i = 30000;
8
9      while (i > 0) {
10         x = x+1;
11         i = i+1;
12     }
13
14     return 0;
15 }
```

Listing 18: *Code met impliciet testen*

Voor meer optimalisatie tips & tricks, zie [7].

25 Bij gebruik van interrupts lijkt het alsof de ATmega32 steeds herstart.

Het gebruik van interrupts levert vele voordelen op. Maar het kan ook een bron zijn van allerlei problemen. Eén van die problemen betreft de zogenaamde *bad interrupt*. Soms heb je het idee dat het programma op de AVR (met interrupts) opnieuw start. Bij debuggen kom je er achter dat inderdaad de bekende gele pijl op de eerste instructie staat. Wat nu? Dit kan gebeuren als je een interrupt aanzet (door middel van interrupt enble flags) waar geen Interrupt Services Routine (ISR) voor is gedefinieerd. Er kan dus

ook niet naar de ISR gesprongen worden. De compiler lost die nogal rigoureus op door naar de reset vector te springen. Een discussie over dit ”probleem” is te vinden op [8].

In het volgende voorbeeld is alleen de ISR voor de T/C1 Compare Match Interrupt gedefinieerd, de ISR van de ADC Completion Interrupt is uitgecommentarieerd.

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

#define PRESCALER 1024UL
#define FREQ 10UL
#define LOAD ((F_CPU/(PRESCALER*FREQ))-1)

volatile uint8_t flag;
volatile uint16_t ADCval;

/* Compare A Match ISR */
ISR(TIMER1_COMPA_vect) {

    /* some code */
}

/* ADC ISR, signals new value is ready, copy value */
/*ISR(ADC_vect) {
    flag = 1;
    ADCval = ADC;
} */

int main() {

    /* Set up T/C1 */
    TCCR1A = (0<<WGM11)|(0<<WGM10);
    TCCR1B = (0<<WGM13)|(1<<WGM12)|(1<<CS12)|(0<<CS11)|(1<<CS10);
    OCR1A  = LOAD;
    TIMSK  = 1<<OCIE1A;

    /* Set up  ADC in single shot mode, select channel 0, interrupt */
    /* Refs is AREF, internal V off, ADC prescaler is 64 */
    ADMUX  = (0<<REFS1)|(0<<REFS0)|(0<<MUX4)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(0<<
        MUX0);
    SFIOR  = (0<<ADTS2)|(0<<ADTS1)|(0<<ADTS0);
    ADCSRA = (1<<ADEN)|(1<<ADSC)|(0<<ADATE)|(0<<ADIF)|(1<<ADIE)|(1<<ADPS2)|(1<<
        ADPS1)|(0<<ADPS0);

    /* Free interrupts */
    sei();

    /* Forever... */
```

```
42      while (1) {
43          /* code here */
44      }
45
46      return 0;
47  }
```

Listing 19: *ISR vergeten*

De compiler genereert een vector table waarin alle niet gedefinieerde vectoren naar de *default interrupt handler* verwijzen.

In onderstaande listing is een deel van de gegenereerde assemblercode afgebeeld (deze file is te vinden met de extensie `.lss`, althans bij gebruik van AVR Studio).

```
1   Disassembly of section .text:
2
3   00000000 <__vectors>:
4      0:   0c 94 2a 00     jmp 0x54    ; 0x54 <__ctors_end>
5      4:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
6      8:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
7      c:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
8     10:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
9     14:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
10    18:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
11    1c:   0c 94 3e 00     jmp 0x7c    ; 0x7c <__vector_7>
12    20:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
13    24:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
14    28:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
15    2c:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
16    30:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
17    34:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
18    38:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
19    3c:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
20    40:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
21    44:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
22    48:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
23    4c:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
24    50:   0c 94 3c 00     jmp 0x78    ; 0x78 <__bad_interrupt>
25
26  00000054 <__ctors_end>:
27
28  ; code skipped.
29
30  00000078 <__bad_interrupt>:
31    78:   0c 94 00 00     jmp 0   ; 0x0 <__vectors>
```

Listing 20: *Deel van de assemblercode*

In de bovenstaande code is zien dat op regel 11 een sprong wordt gemaakt naar de ISR voor T/C1, maar in regel 20 wordt gesprongen naar de default interrupt handler. Deze routine staat beschreven op regel 31 en zoals te zien is, wordt gesprongen naar adres 0. Daar wordt weer gesprongen naar de startup code.

26 Debuggen gooit communicatie met de buitenwereld in de war.

Tijdens debuggen op de ATmega heb je de mogelijkheid om de executie van de instructies stap voor stap uit te voeren. Je kan dan mooi de inhouden van de register en de status flags bekijken en nog veel meer. Maar de buitenwereld (lees: randapparatuur) weet dat niet. Die trekken zich niks aan van jouw debugmomentje. Neem als voorbeeld de USART. Je wil bijvoorbeeld weten of bepaalde interruptroutines voor zenden en ontvangen goed werken. Dan zet je dus een breakpoint bij het begin van de betreffende ISR. Na ontvangst van een karakter wordt de processor gestopt op het breakpoint-adres. De zendende partij weet dat niet en gaat vrolijk door. Zo mis je dus de nodige informatie. De externe interrupts `int0`, `int1` en `int2` hebben hier ook last van. Real-time Operating Systems (RTOS) kunnen dus op deze manier niet gedebugd worden.

27 Compiler-optimalisatie gooit debuggen in de war.

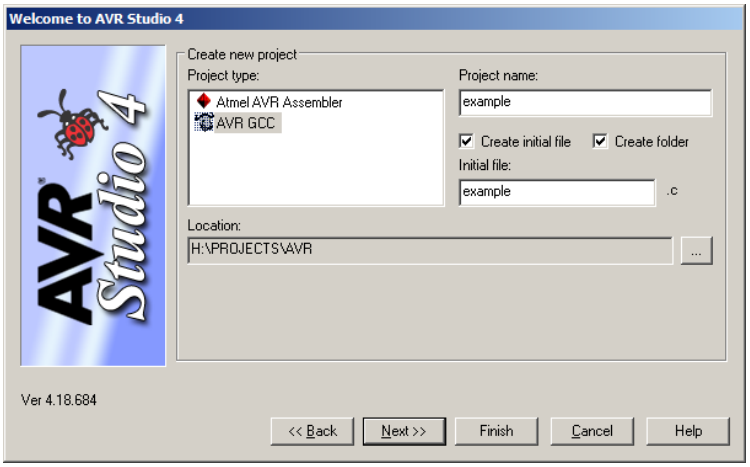
Meestal werkt de code van een programma niet in één keer. Gelukkig is er de mogelijkheid om de code te debuggen door stap voor stap door de coderegels te lopen. Helaas zorgt compiler-optimalisatie voor problemen. Net als bij 24 zet de compiler C-code om naar eigen inzicht. Dat kan bijvoorbeeld zijn dat een `while`-lus wordt omgewerkt tot een `do-while`-lus. Vaak is het dan niet mogelijk om één-op-één de C-code te koppelen aan de gegenereerde assemblercode. Tijdens het debuggen verdwijnt de gele pijl ineens. Soms komt deze dan weer terug, maar meestal moet je het debuggen even pauzeren (en dan nog komt de pijl niet altijd terug). De oplossing is simpel: zet de compiler-optimalisatie uit. Dit kan je doen door optie `-O0` in te stellen in de project-omgeving (in AVR Studio).

28 [AS4] Bij debuggen werken I/O registers niet naar behoren.

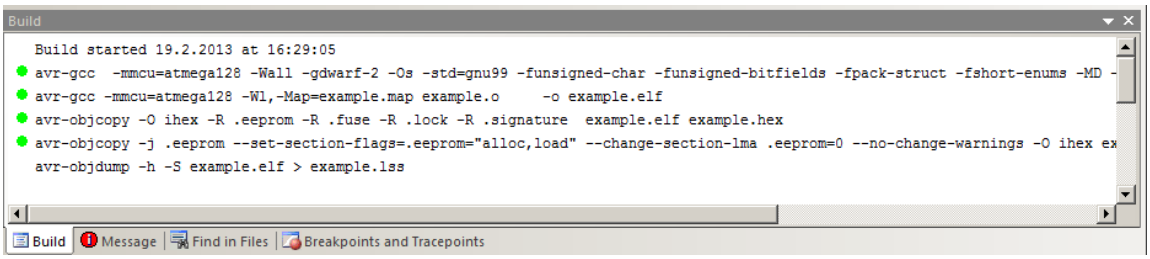
De code compileert, er zijn geen syntaxfouten. Maar bij het debuggen blijken sommige I/O-registers niet te werken. Na onderzoek blijkt dat de compiler staat ingesteld op `atmega128`. Wat is er mis gegeaan? Waarschijnlijk heb je tijdens het aanmaken van het project te vroeg op `Finish` geklikt (zie figuur 1) zodat het tweede deel, waarin je het debugplatform en microcontrollertype opgeeft, is overgeslagen. De compiler staat dan automatisch ingesteld op `atmega128`, zie figuur 2.

29 [AS4] Vermijd gebruik van floating point.

De AVR-controller heeft geen hardware voor floating point-berekeningen aan boord. Dat betekent dat zelfs een simpele vermenigvuldiging of deling in software moet worden geëmuleerd. En dat kost veel ruimte in Flash-ROM (programmageheugen) en veel executietijd. Onderstaande code werd gedraaid op een ATmega16 met GNU-C compiler



Figuur 1: *Eerste scherm bij aanmaken nieuw project*



Figuur 2: *De compiler genereert code voor de ATmega128*

versie 4.1. De floating point-berekeningen duren meer dan twee keer zo lang als de integer-berekeningen. Het gebruik van de optimalisatievlag `-O2` helpt weinig (zeg maar gewoon: niets) want de routines voor vermenigvuldigen en delen zijn al geoptimaliseerd door de programmeurs. Je kan veel eenvoudige floating point-delingen makkelijk omzetten naar integer-delingen (let wel op het bereik van int's en long's) (let ook op de berekenvolgorde!):

$$fo = \frac{ftime}{fdiv} = \frac{ftime}{3,686} \quad \Rightarrow \quad lo = \frac{ltime \cdot lmul}{ldiv} = \frac{ltime \cdot 1000}{3686} \qquad \left(\frac{1}{3,686} \leftrightarrow \frac{1000}{3686}\right)$$

```
1  /* compile with -O0 and -O2 successively */
2
3  volatile float fx=13.0;
4  volatile float fy=23.535;
5  volatile float fz;
6
7  volatile long  lx=13UL;
8  volatile long  ly=23535;
9  volatile long  lz;
10
11 volatile long   ftime = 3456;
12 volatile float  fdiv = 3.686;
13 volatile float  fo;
```

```
14
15 volatile long   ltime = 3456;
16 volatile long   lmul = 1000;
17 volatile long   ldiv = 3686;
18 volatile long   lo;
19
20 int main() {
21
22     fz = fy/fx;
23
24     lz = ly/lx;
25
26     fo = ftime/fdiv;
27
28     lo = (ltime*lmul)/ldiv;    /* Mind parentheses!!! */
29
30     return 0;
31 }
```

Listing 21: *Code met integer en floating point-berekeningen*

Let ook op het gebruik van `#define`-constanten. Constructies als

```
#define R_OPT (14750.0)
```

levert extreem veel code op omdat er tóch FP-routines worden gebruikt.

Referenties

[1] ISO/JTC1/SC22/WG14. *ISO/IEC 9899:1999 specification*. pag. 20, §5.2.4.1. 2005, p. 20. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (bezocht op 22-01-2013) (blz. 9).

[2] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009. ISBN: 9780132350884 (blz. 10).

[3] ISO/JTC1/SC22/WG14. *ISO/IEC 9899:1999 specification*. §5.2.4.2.1. 2005, p. 21–23. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (bezocht op 22-01-2013) (blz. 11).

[4] Atmel. *Datasheet ATmega32(L)*. Versie 8155G. Okt 2015, p. 263. URL: http://www.atmel.com/Images/Atmel-8155-8-bit-Microcontroller-AVR-ATmega32A_Datasheet.pdf (bezocht op 27-12-2015) (blz. 20).

[5] Atmel. *AVR libc v1.8.0*. pag. 27, hoofdstuk 5. 2012. URL: http://savannah.nongnu.org/projects/avr-libc/ (bezocht op 22-01-2013) (blz. 25).

[6] Atmel. *Datasheet ATmega32(L)*. 2011, 165, tabel 68. URL: http://www.atmel.com/Images/doc2503.pdf (bezocht op 26-01-2015) (blz. 27).

[7] Atmel. *Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers*. Nov 2011. URL: http://www.atmel.com/images/doc8453.pdf (bezocht op 06-01-2016) (blz. 28).

[8] Joerg Wunsch. URL: http://savannah.nongnu.org/task/?7546 (bezocht op 16-11-2015) (blz. 29).

Index