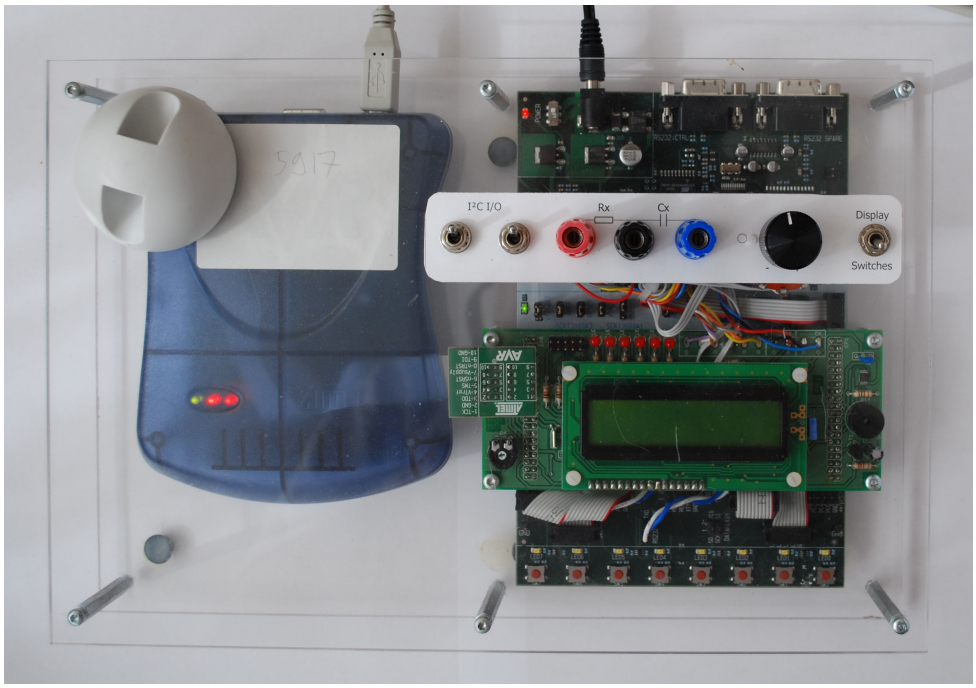
Studiemateriaal
MICPRG

versie 1.22



Harry Broeders en Ben Kuiper
De Haagse Hogeschool
Opleiding Elektrotechniek
9 december 2015
mailto:J.Z.M.Broeders@hhs.nl
mailto:B.Kuiper@hhs.nl

Inhoudsopgave

1 Inleiding

In dit document kun je de leerstof van MICPRG die hoort bij iedere les terugvinden. In vele gevallen zal je verwijzingen naar het boek van [MNN13] vinden, maar soms ook materiaal/toelichtingen van de auteurs van deze reader.

2 Inleiding MICPRG

Het vak microprocessorbesturingen verschaft je inzicht in de toepassingsmogelijkheden van microcontrollers. Heel veel producten en systemen zijn tegenwoordig uitgerust met een of meerdere microcontrollers die deze producten en systemen besturen. Microcontrollers worden ook wel ”embedded controllers” genoemd omdat de controller volledig in het product of systeem is opgenomen (je ziet er aan de buitenkant niets van). Het gebruik van een microcontroller in producten en systemen heeft (ten opzichte van een volledig hardwarematige besturing) de volgende voordelen:

- De functionaliteit van de besturing ligt vast in software (”embedded software”) en kan door de fabrikant van het produkt relatief eenvoudig gewijzigd worden als dat nodig is.
- Het produkt kan met behulp van embedded software vaak gebruiksvriendelijker, nauwkeuriger, veiliger en energiezuiniger gemaakt worden.

Omdat de producten waarin een microcontroller kan worden opgenomen sterk variëren in complexiteit en kostprijs bestaan er een groot aantal verschillende soorten microcontrollers. In http://www.faqs.org/faqs/microcontroller-faq/primer/ worden de volgende toepassingen van microcontrollers genoemd:

> Embedded processors and microcontrollers are frequently found in: appliances (microwave oven, refrigerators, television and VCRs, stereos), computers and computer equipment (laser printers, modems, disk drives), automobiles (engine control, diagnostics, climate control), environmental control (greenhouse, factory, home), instrumentation, aerospace, and thousands of other uses. In many items, more than one processor can be found. Microcontrollers are typically used where processing power isn't so important. Although some of you out there might find a microwave oven controlled by a Unix system an attractive idea, controlling a microwave oven is easily accomplished with the smallest of microcontrollers. On the other hand, if you're putting together a cruise missile to solve the problem of your neighbor's dog barking at 3 in the morning, you'll probably need to use processors with a bit more computing power. Embedded processors and microcontrollers are used extensively in robotics. A special application that microcontrollers are well suited for is data logging. Stick one of these chips out in the middle of a corn field or up in a ballon, and monitor and record environmental parameters (temperature, humidity, rain, etc). Small size, low power consumption, and flexibility make these devices ideal for unattended data monitoring and recording.

In EQ1.1 heb je al kennisgemaakt met de AVR ATmega32 microcontroller, een populaire 8 bits microcontroller van Atmel. (Zie: `http://www.atmel.com/products/AVR/`). In het onderwijsdeel MICPRG gaan we dieper op deze AVR ATmega32 controller in. We doen

dit zoveel mogelijk aan de hand van concrete voorbeelden en opdrachten waarbij naast de software ook de hardware een belangrijke rol speelt. Met name de koppeling tussen de controller en zijn omgeving (de interface) krijgt de nodige aandacht. In het bijzonder de timer, de analoge en de seriële interface. Ook het werken met interrupts komt aan de orde. Bij INLMIC heb je de microcontroller in assembler geprogrammeerd. Bij MICPRG leer je om C te gebruiken om de microcontroller te programmeren. De C kennis die je bij het onderwijsdeel GESPRG hebt verworven wordt bij MICPRG gebruikt en verder uitgebreid. De voordelen van het gebruik van C ten opzichte van assembler zijn:

- Een in C geschreven programma is eenvoudiger te lezen en te begrijpen. Daardoor is een in C geschreven programma beter aan te passen en eenvoudiger uit te breiden dan een in assembler geschreven programma.
- In C kunnen we gebruik maken van een gestructureerde aanpak om een programma beter leesbaar en dus beter aanpasbaar en onderhoudbaar te maken. We kunnen ook algemeen bruikbare functies ontwikkelen die het hergebruik van code eenvoudiger maken.
- In C geschreven programma's voor de AVR kunnen eenvoudiger worden omgezet naar een programma voor een andere microcontroller. Zeker als de microcontroller specifieke delen van het programma zijn "verborgen" in een aantal specifieke functies.

3 Les 1

Centraal in deze les staat hoe je in C een AVR microcontroller kan programmeren. Het aanpassen van bitjes in zgn. registers om bepaalde outputs aan te sturen of inputs uit te lezen is de volgende stap. De leerstof die hierbij hoort uit het boek van [MNN13] is 8.1 (exclusief blz. 261 en 262), 8.2 en 8.3. Naast het boek bevat deze reader ook nog leerstof. Deze leerstof is min of meer hetzelfde als hetgeen dat staat in 8.3. Je kunt dus kiezen welke van de twee je doorneemt. Mocht een van de twee onduidelijk zijn, dan kun je natuurlijk altijd even kijken in de andere.

Wel moet worden opgemerkt dat paragraaf 3.2.7 op pagina 14 in gaat op de relatie tussen rekenkundige en schuifoperatoren terwijl dit niet te vinden is in het boek van [MNN13]. Dit is iets dat je al gehad hebt bij GESPRG, dus dit zou je dan ook kunnen overslaan. Wel is het verstandig om te kijken hoe schuiven gaat bij signed variabelen en dit is te zien aan het einde van paragraaf 3.2.7 op pagina 14. Daarnaast wordt het verschil tussen logische en bitwise operatoren in paragraaf 3.2.1 op pagina 7 en paragraaf 3.2.2 op pagina 8 uitgebreider toegelicht dan in [MNN13]. Als dit verschil je niet duidelijk is aan de hand van 8.3 dan is het een verstandig idee om paragrafen 3.2.1 en 3.2.2 door te lezen.

Nadat je het bovenstaande hebt bestudeerd is het verstandig om je te verdiepen in de betekenis van *volatile*. Je zal dit voor de eerste practicumopdracht ook moeten bestuderen. Een beschrijving van *volatile* kun je hier vinden.

3.1 Opmerkingen [MNN13]

Er zijn nog wel enige opmerkingen te plaatsen over de paragrafen 8.1, 8.2 en 8.3. Deze zijn als volgt:

- Het boek gebruikt in vele voorbeelden de variabele declaratie:

  ```
  unsigned char z;
  ```

  In voorbeelden in de slides en het practicum zie je weer het gebruik van bijvoorbeeld:

  ```
  uint8_t z;
  ```

  In principe doen beide declaraties hetzelfde, want `uint8_t` is een macro die gedefinieerd is in de io.h header file (dit is uiteindelijk eigenlijk weer een andere header file). `uint8_t` is dus een alias voor een unsigned char en is als volgt gedeclareerd in de io.h header:

  ```
  typedef unsigned char uint8_t;
  ```

  Het is aan te bevelen om `uint8_t` te gebruiken in plaats van unsigned char. Het is allereerst een kortere schrijfwijze en daarnaast is het duidelijker dat je in dit geval een 8 bits variabele gebruikt.

- Bladzijde 259: Je kunt hier duidelijk merken dat sommige hoofdstukken in het boek erg lang geleden zijn geschreven. Men heeft het hier nog over een Pentium met 32 bit registers. Inmiddels zijn er natuurlijk nieuwere processoren (zoals bijv. de Intel Core series) en deze processoren hebben vaak 64 bit registers. De essentie van deze alinea blijft echter wel hetzelfde. Je kunt dus niet zomaar een int gebruiken op een AVR zonder dat je programma langzamer wordt en het benodigde programma- en datageheugen omhoog gaat.
- Bladzijde 267: Example 16 is voor nu nog niet zo zinvol. Later ga je nog gebruik maken van de LCD, dus dit voorbeeld kun je nu overslaan.
- Bladzijde 268: Ook example 17 kun je overslaan.

3.2 Bitn..... in C voor beginners

Deze paragraaf bevat informatie die ook terug te vinden is in 8.3 uit [MNN13]. Je kunt dus deze paragraaf bestuderen in plaats van 8.3.

Werktuigbouwkundig ingenieurs worden wel eens oneerbiedig aangesproken met de vakterm: ”fietsenmaker”. Zo bestaat er ook een vakterm voor E of TI ingenieurs die zich bezighouden met het programmeren van microcontrollers: bitn..... (`http://ti-aalst.powerhost.be/index.php?show=woordenboek&letter=B&id=178`)

In deze paragraaf wordt uitgelegd op welke manieren je met bitjes kunt spelen en hoe je dat in C (veilig ;-) moet doen.

3.2.1 Bitje setten

Je kunt een bitje setten (1 maken) met behulp van een bitwise-or operator. Je moet het bitje dat je wilt setten or-en met 1 en de rest met 0. Om dus bijvoorbeeld pin PB3 één te maken moeten we PORTB or-en met het binaire getal: 00001000.

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

int main(void) {
    DDRB = 0xFF; /* alle pinnen poort B op output */
    PORTB = 0x55; /* willekeurige test waarde op poort B */
    PORTB = PORTB | 0x08; /* alleen pin PB3 1 maken, de overige ↩
     ↪ pinnen van poort B worden niet gewijzigd */
    while (1);
    return 0;
}
```

Listing 1: Voorbeeld van het setten van een pin.

De regel:

```
PORTB = PORTB | 0x08;
```

kun je ook verkorten tot:

```
PORTB |= 0x08;
```

Let op! Er zit een groot verschil tussen de bitwise-or operator | en de logical-or operator ||. Bij de bitwise-or wordt de or bewerking bit-voor-bit uitgevoerd. 0x2c|0x09 is dus gelijk aan 0x2d. Zet de getallen even om naar het binaire talstelsel als je het niet meteen ziet. Bij de logical-or wordt het getal omgezet naar een logische (binaire) waarde (true of false). Daarna wordt de or bewerking uitgevoerd met als resultaat true (1) of false (0). 0x2c||0x09 is dus gelijk aan 0x01. Als in het bovenstaande programma de | operator vervangen wordt door de || operator wordt pin PB0 geset en alle overige pinnen van poort B gereset!

Er is nog een verschil tussen de bitwise-or operator |en de logical-or operator ||. Bij de logical-or operator worden de operanden van links naar rechts uitgerekend en zodra het antwoord bekend is wordt de berekening gestopt. Dit wordt short-circuit evaluation genoemd. Bij de bitwise-or wordt de expressie altijd helemaal doorgerekend. Voorbeeld: Als de expressie fun1()||fun2() wordt uitgevoerd en fun1() geeft true terug dan wordt fun2() niet aangeroepen (het antwoord van de expressie is true). Als fun1()|fun2() wordt uitgevoerd dan worden fun1() en fun2() altijd beiden aangeroepen (ook als fun1() allemaal enen teruggeeft).

Er is daarnaast nog een subtiel verschil. Bij de logical-or operator ligt de evaluatievolgorde van de operanden vast maar bij de bitwise-or niet. Voorbeeld: Als de expressie fun1()||fun2() wordt uitgevoerd wordt fun1() als eerste aangeroepen (fun2() wordt mogelijk helemaal niet aangeroepen). Als fun1()|fun2() wordt uitgevoerd dan is het compiler afhankelijk of eerst fun1() of eerst fun2() wordt aangeroepen (ze worden wel gegarandeerd beiden aangeroepen).

3.2.2 Bitje clearen

Je kunt een bitje clearen (0 maken) met behulp van een bitwise-and operator. Je moet het bitje dat je wilt clearen and-en met 0 en de rest met 1. Om dus bijvoorbeeld pin PB3 nul te maken moeten we PORTB and-en met het binaire getal: 11110111.

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

int main(void) {
    DDRB = 0xFF; /* alle pinnen poort B op output */
    PORTB = 0xAA; /* willekeurige test waarde op poort B */
    PORTB = PORTB & 0xF7; /* alleen pin PB3 0 maken, de overige ↩
     ↪ pinnen van poort B worden niet gewijzigd */
    while (1);
    return 0;
}
```

Listing 2: Voorbeeld van het clearen van een pin.

De regel:

```
    PORTB = PORTB & 0xF7;
```

kun je ook verkorten tot:

```
    PORTB &= 0xF7;
```

Het is ook mogelijk om bij het clearen hetzelfde bitpatroon te gebruiken als bij het setten. Je moet dan de compiler zelf de inverse laten uitrekenen door middel van de bitwise-not ~ operator:

```
    PORTB &= ~0x08;
```

Let op! Er zit een groot verschil tussen de bitwise-and operator & en de logical-and operator &&. Bij de bitwise-and wordt de and bewerking bit-voor-bit uitgevoerd. 0x2c&0x09 is dus gelijk aan 0x08. Zet de getallen even om naar het binaire talstelsel als je het niet meteen ziet. Bij de logical-and wordt het getal omgezet naar een logische (binaire) waarde (true of false). Daarna wordt de and bewerking uitgevoerd met als resultaat true (1) of false (0). 0x2c&&0x09 is dus gelijk aan 0x01. Als in het bovenstaande programma de & operator vervangen wordt door de && operator dan wordt pin PB0 geset en alle overige pinnen van poort B gereset!

Er is nog een verschil tussen de bitwise-and operator & en de logical-and operator &&. Bij de logical-and operator worden de operanden van links naar rechts uitgerekend en zodra het antwoord bekend is wordt de berekening gestopt. Dit wordt short-circuit evaluation genoemd. Bij de bitwise-and wordt de expressie altijd helemaal doorgerekend. Voorbeeld: Als de expressie fun1()&&fun2() wordt uitgevoerd en fun1() geeft false terug dan wordt fun2() niet aangeroepen (het antwoord van de expressie is false). Als fun1()&fun2() wordt uitgevoerd dan worden fun1() en fun2() altijd beiden aangeroepen (ook als fun1() allemaal nullen teruggeeft).

Er is nog een subtiel verschil. Bij de logical-and operator ligt de evaluatievolgorde van de operanden vast maar bij de bitwise-and niet. Voorbeeld: Als de expressie fun1()&&fun2() wordt uitgevoerd wordt fun1() als eerste aangeroepen (fun2() wordt mogelijk helemaal niet aangeroepen). Als fun1()&fun2() wordt uitgevoerd dan is het compiler afhankelijk of eerst fun1() of eerst fun2() wordt aangeroepen (ze worden wel gegarandeerd beiden aangeroepen).

Let op! Er zit een groot verschil tussen de bitwise-not operator ~ en de logical-not operator !. Bij de bitwise-not wordt de not bewerking bit-voor-bit uitgevoerd. ~0x2c is dus gelijk aan 0xd3. Zet de getallen even om naar het binaire talstelsel als je het niet meteen ziet. Bij de logical-not wordt het getal omgezet naar een logische (binaire) waarde (true of false). Daarna wordt de not bewerking uitgevoerd met als resultaat true (1) of false (0). !0x2c is dus gelijk aan 0x00.

3.2.3 Bitje flippen

Je kunt een bitje flippen (inverteren) met behulp van een bitwise-exor operator. Je moet het bitje dat je wilt flippen exor-en met 1 en de rest met 0. Om dus bijvoorbeeld pin PB3 te inverteren moeten we PORTB exor-en met het binaire getal: 00001000.

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

int main(void) {
    DDRB = 0xFF; /* alle pinnen poort B op output */
    PORTB = 0xAA; /* willekeurige test waarde op poort B */
    PORTB = PORTB ^ 0x08; /* alleen pin PB3 inverteren, de overige ↩
     ↪ pinnen van poort B worden niet gewijzigd */
    while (1);
    return 0;
}
```

Listing 3: Voorbeeld van het inverteren van een pin.

De regel:

```
PORTB = PORTB ^0x08;
```

kun je ook verkorten tot:

```
PORTB ^= 0x08;
```

Er bestaat in C vreemd genoeg geen logical-exor operator.

3.2.4 Meerdere bitjes veranderen

Als je meerdere bitjes wilt setten, meerdere bitjes wilt clearen of meerdere bitjes wilt inverteren dan kun je dat doen door in het bitpatroon waarmee je respectievelijk de bitwise-or, bitwise-and of bitwise-exor uitvoert meerdere bitjes te setten.

In het onderstaande voorbeeld worden PB4 en PB2 geset, PB5 en PB1 gecleared en PB7, PB6 en PB0 geïnverteerd:

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

int main(void) {
    DDRB = 0xFF; /* alle pinnen poort B op output */
    PORTB = 0x5A; /* willekeurige test waarde op poort B */
    PORTB |= 0x14; /* alleen PB4 en PB2 1 maken */
    PORTB &= ~0x22; /* alleen PB5 en PB1 0 maken */
    PORTB ^= 0xc1; /* alleen PB7, PB6 en PB0 inverteren */
    while (1);
    return 0;
}
```

Listing 4: Voorbeeld waarin pinnen worden geset gecleard en geïnverteerd.

3.2.5 Bitje testen

De onderstaande voorbeelden testen een bitje in het PINA register van de AVR ATmega32. Op het STK500 practicumbordje zijn de pinnen van poort A PA0 t/m PA7 verbonden met 8 drukschakelaars (SW0 t/m SW7) zodat we de programma's eenvoudig kunnen testen. Als de test true oplevert worden de pinnen PB0 t/m PB6 één en pin PB7 nul gemaakt (ledje LED7 wordt als enige aangezet) en als de test false oplevert worden de pinnen PB1 t/m PB7 één en pin PB0 nul gemaakt (ledje LED0 wordt als enige aangezet) zodat we meteen het resultaat van de test kunnen zien. Het DDRA register moet dan wel geladen worden met 0x00 om alle pinnen van poort A op input te zetten en het DDRB register moet geladen worden met 0xFF om alle pinnen van poort B op output te zetten. Denk er aan dat de drukschakelaars op het STK500 bord de pin 0 maken als de schakelaar wordt ingedrukt en de pin 1 maken als de schakelaar niet wordt ingedrukt.

Je kunt testen of een bitje 1 is door dit bitje te "isoleren" van de andere bitjes in de betreffende variabele. De overige bits worden gemaskeerd. Je kunt een bitje isoleren door een bitwise-and bewerking. Het volgende voorbeeld zal als schakelelaar SW3 niet ingedrukt is (pin PA3 is dan 1) alleen LED7 laten branden (PB7 is als enige pin van poort B nul) en anders (SW3 wel ingedrukt) alleen LED0 laten branden:

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        if ((PINA & 0x08) == 0x08) {
            PORTB = 0x7F;
        }
        else {
            PORTB = 0xFE;
        }
    }
```

```
    return 0;
}
```

Listing 5: Voorbeeld waarbij gekeken wordt of een pin 1 is

De extra haakjes in de if instructie zijn noodzakelijk omdat de bitwise-and operator & een lagere prioriteit heeft dan de vergelijkings operator ==.

De regel:

```
    if((PINA & 0x08) == 0x08) {
```

kun je ook verkorten tot:

```
    if(PINA & 0x08) {
```

De expressie (PINA & 0x08) geeft namelijk als resultaat 0x08 als pin PA3 één is en 0x00 als pin PA3 nul is. 0x08 is ongelijk aan nul en wordt dus gezien als de logische waarde true en 0x00 is gelijk aan nul en wordt dus gezien als de logische waarde false.

Je kunt testen of een bitje 0 is door dit bitje te ”isoleren” van de andere bitjes in de betreffende variabele. De overige bits worden gemaskeerd. Je kunt een bitje isoleren door een bitwise-and bewerking. Het volgende voorbeeld zal als schakelelaar SW3 ingedrukt is (PA3 is dan 0) alleen LED7 laten branden en anders alleen LED0 laten branden:

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        if ((PINA & 0x08) == 0x00) {
            PORTB = 0x7F;
        }
        else {
            PORTB = 0xFE;
        }
    }
    return 0;
}
```

Listing 6: Voorbeeld waarbij gekeken wordt of een pin 0 is.

De regel:

```
    if ((PINA & 0x08) == 0x00) {
```

kun je ook verkorten tot:

```
    if (!(PINA & 0x08)) {
```

of tot:

```
    if (~PINA & 0x08) {
```

De extra haakjes in de tweede if instructie zijn noodzakelijk omdat de bitwise-and operator & een lagere prioriteit heeft dan de logical-not operator !.

De expressie (PINA & 0x08) geeft namelijk als resultaat 0x00 als schakelaar PA3 nul is en 0x08 als schakelaar PA3 één is. 0x00 is gelijk aan nul en wordt dus gezien als de logische waarde false en 0x08 is ongelijk aan nul en wordt dus gezien als de logische waarde true. Als je deze logische waarde met een logical-not operator inverteert krijg je de waarde true als bit PA3 nul is en false als PA3 één is.

Je kunt ook eerst een bitwise-not uitvoeren op de uit PINA gelezen waarde. Alle bitjes (dus ook bitje PA3) worden dan geinverteerd. Hierna kan je dan op de hierboven beschreven manier testen of bitje 3 in de geïnverteerde waarde van PINA één is (~PINA & 0x08). Er zijn daarbij geen extra haakjes nodig omdat de bitwise-not operator een hogere prioriteit heeft dan de bitwise-and operator.

3.2.6 Meerdere bitjes testen

Je kunt vaak meerdere bitjes met één bewerking testen door meerdere bitjes te isoleren.

In het onderstaande voorbeeld wordt alleen LED7 aangezet als PB5 één is **en** PB3 één is (dus als SW5 niet ingedrukt is **en** SW3 niet ingedrukt is). Als dit niet het geval is (SW5 of SW3 is ingedrukt of beiden zijn ingedrukt) dan wordt alleen LED0 aangezet.

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        if ((PINA & 0x28) == 0x28) {
            PORTB = 0x7F;
        }
        else {
            PORTB = 0xFE;
        }
    }
    return 0;
}
```

Listing 7: Voorbeeld waarbij meerdere bitjes worden getest.

De waarheidstabel (tabel 1) kan helpen bij het doorgronden van de werking van het bovenstaande programma:

Let op! De regel:

```
    if((PINA & 0x28) == 0x28) {
```

kun je nu **niet** verkorten!

De regel:

Tabel 1: Waarheidstabel bij listing 7.

SW5	SW3	PA5	PA3	PINA & 0x28	(PINA & 0x28) == 0x28	PORTB	PB7	PB0	LED7	LED0
niet inge-drukt	niet inge-drukt	1	1	0x28	true	0x7F	0	1	aan	uit
niet inge-drukt	wel inge-drukt	1	0	0x20	false	0xFE	1	0	uit	aan
wel inge-drukt	niet inge-drukt	0	1	0x08	false	0xFE	1	0	uit	aan
wel inge-drukt	wel inge-drukt	0	0	0x00	false	0xFE	1	0	uit	aan

```
    if(PINA & 0x28) {
```

geeft namelijk een **heel ander resultaat**. De expressie (PINA & 0x28) levert namelijk ook true op als alleen pin PA5 één is of alleen PA3 één is!

In het onderstaande voorbeeld wordt alleen LED7 aangezet als PA5 één is **of** PA3 één is (dus als SW5 niet ingedrukt is of SW3 niet ingedrukt is). Als dit niet het geval is (SW5 en SW5 zijn beiden ingedrukt) dan wordt alleen LED0 aangezet.

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        if ((PINA & 0x28) != 0x00) {
            PORTB = 0x7F;
        }
        else {
            PORTB = 0xFE;
        }
    }
    return 0;
}
```

Listing 8: LED7 aanzetten als PA5 één is of PA3 één is.

De waarheidstabel (tabel 2) kan helpen bij het doorgronden van de werking van het bovenstaande programma:

Tabel 2: Waarheidstabel bij listing 8.

SW5	SW3	PA5	PA3	PINA & 0x28	(PINA & 0x28) != 0x00	PORTB	PB7	PB0	LED7	LED0
niet inge-drukt	niet inge-drukt	1	1	0x28	true	0x7F	0	1	aan	uit
niet inge-drukt	wel inge-drukt	1	0	0x20	true	0x7F	0	1	aan	uit
wel inge-drukt	niet inge-drukt	0	1	0x08	true	0x7F	0	1	aan	uit
wel inge-drukt	wel inge-drukt	0	0	0x00	false	0xFE	1	0	uit	aan

De regel:

```
if((PINA & 0x28) != 0x00) {
```

kun je verkorten tot:

```
if(PINA & 0x28) {
```

De expressie (PINA & 0x28) levert namelijk ook true op als alleen pin PA5 één is of alleen PA3 één is.

3.2.7 Schuiven met bitjes

In C zijn ook operatoren gedefinieerd waarmee je een bitpatroon kunt schuiven. Deze operatoren worden shift-operators genoemd en het zijn binaire operatoren (er zijn 2 operanden). De operator « schuift naar links en de operator » naar rechts. Aan de linkerkant van de shift-operator staat het patroon dat verschoven moet worden en aan de rechterkant staat het aantal plaatsen dat geschoven moet worden, de zogenaamde shift-count.

In het onderstaande voorbeeld wordt het bitpatroon van de schakelaars ingelezen en 2 plaatsen naar links geschoven naar de leds gestuurd. Als op de schakelaars 0xBD staat (SW6 en SW1 zijn ingedrukt) zal op de leds dus 0xF4 verschijnen (LED0, LED1 en LED3 branden). Zet de getallen om naar het binaire talstelsel als je het niet meteen ziet.

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
```

```
    while (1) {
        PORTB = PINA << 2;
    }
    return 0;
}
```

Listing 9: Bitpatroon schuiven naar links

Er bestaat ook een «= en een operator »= waarmee schuiven en assignment gecombineerd kunnen worden. In het volgende programma wordt de waarde die op de schakelaars staat eerst ingelezen in een variabele en daarna drie plaatsen naar rechts geschoven:

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        uint8_t b;
        b = PINA;
        b >>= 3;
        PORTB = b;
    }
    return 0;
}
```

Listing 10: Schuiven en assignment gecombineerd

Bij het schuiven naar links worden er altijd nullen ingeschoven. Schuiven van y plaatsen naar links komt overeen met vermenigvuldigen met 2^y. Het onderstaande programma levert dus exact hetzelfde resultaat als het bovenstaande programma waarin 2 plaatsen naar links wordt geschoven:

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

int main(void) {
    DDRB = 0xFF;
    DDRA = 0x00;
    while (1) {
        PORTB = PINA * 4;
    }
    return 0;
}
```

Listing 11: Vermenigvuldiging met 4 is hetzelfde als het schuiven van twee plaatsen naar links.

Bij schuiven naar **rechts** is het wat ingewikkelder.

Als het patroon **unsigned** is worden er ook nullen ingeschoven. Als de unsigned 8 bits waarde 0xBD twee plaatsen naar rechts wordt geschoven dan levert dat de waarde 0x2F

op. Bij **unsigned** getallen komt y plaatsen schuiven naar rechts overeen met delen door $2y$. De twee onderstaande programma's leveren dus exact hetzelfde resultaat:

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

int main(void) {
    uint8_t b = 0xBD;
    b >>= 2;
    DDRB = 0xFF;
    PORTB = b;
    while (1);
    return 0;
}
```

Listing 12: 2 plaatsen schuiven naar rechts

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

int main(void) {
    uint8_t b = 0xBD;
    b /= 4;
    DDRB = 0xFF;
    PORTB = b;
    while (1);
    return 0;
}
```

Listing 13: Delen door 4

Als het patroon **signed** is wordt bij het inschuiven de tekenbit (bit7) gekopieerd. In het onderstaande voorbeeld wordt de 8 bits signed waarde 0xBD plaatsen naar rechts geschoven. Dit levert de waarde 0xEF op. Zet de getallen om naar het binaire talstelsel als je het niet meteen ziet.

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

int main(void) {
    int8_t b = 0xBD;
    b >>= 2;
    DDRB = 0xFF;
    PORTB = b;
    while (1);
    return 0;
}
```

Listing 14: 2 plaatsen schuiven naar rechts met een signed 8-bits int.

Bij negatieve **signed** getallen komt *y* plaatsen schuiven naar rechts ook overeen met delen door 2 maar is het resultaat vreemd genoeg niet hetzelfde als het resultaat van de / operator. Als je in het bovenstaande programma de schuifbewerking vervangt door een deling dan wordt de variabele b na de deeloperatie gelijk aan 0xF0.

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

int main(void) {
    int8_t b = 0xBD;
    b /= 4;
    DDRB = 0xFF;
    PORTB = b;
    while (1);
    return 0;
}
```

Listing 15: Delen door 4 met een signed 8-bits int.

Bij signed schuiven naar rechts is de rest (wat er wordt uitgeschoven) altijd positief bij signed delen is de rest negatief als het deeltal negatief is.

Bij delen met behulp van de » operator: 0xBD gedeeld door 4 = 0xEF rest 0x01 (rest is wat er wordt uitgeschoven). In het signed two's complement talstelsel is dit dus decimaal: -67 gedeeld door 4 = -17 rest 1. Deze vorm van delen wordt "Euclidean division" genoemd: `http://en.wikipedia.org/wiki/Euclidean_division`.

Bij delen met behulp van de / operator: 0xBD gedeeld door 4 = 0xF0 rest 0xFD (rest kun je bepalen met de % operator). In het signed two's complement talstelsel is dit dus decimaal: -67 gedeeld door 4 = -16 rest -3. Deze manier van delen wordt "truncated division" genoemd: `http://en.wikipedia.org/wiki/Modulo_operation`.

Beide antwoorden zijn wiskundig correct. Want -17 * 4 + 1 = -67 en -16 * 4 + -3 = -67. Zie eventueel `http://en.wikipedia.org/wiki/Remainder#The_case_of_general_integers`

3.2.8 Maskers en patronen samenstellen door een 1 naar links te schuiven

Bij het manipuleren en testen van afzonderlijke bits wordt vaak gebruik gemaakt van bitpatronen of maskers waarin slecht op één positie een 1 voorkomt. Om bijvoorbeeld pin PB6 één te maken moeten we PORTB or-en met het binaire patroon: 01000000.

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

int main(void) {
    DDRB = 0xFF;
    PORTB |= 0x40;
    while (1);
    return 0;
```

```
}
```

Listing 16: PB6 één maken

Je kunt het benodigde patroon ook uit laten rekenen door de compiler door de constante 1 zes plaatsen naar links te schuiven:

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

int main(void) {
    DDRB = 0xFF;
    PORTB |= 1<<6;
    while (1);
    return 0;
}
```

Listing 17: PB6 één maken door te schuiven

De expressie 1«6 wordt door de compiler uitgerekend en levert de waarde 0x40 op zodat beide programma's exact dezelfde machinecode opleveren. De meeste mensen vinden het tweede programma beter leesbaar omdat je meteen ziet dat bit 6 van PORTB geset wordt.

Als in het benodigde patroon meer dan 1 bit geset moeten worden dan kan dit door verschillende schuifexpressies met een bitwise-or met elkaar te combineren. In het onderstaande programma worden de pinnen PB6, PB4, PB2 en PB0 één gemaakt:

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

int main(void) {
    DDRB = 0xFF;
    PORTB |= 1<<6 | 1<<4 | 1<<2 | 1<<0;
    while (1);
    return 0;
}
```

Listing 18: Setten van verschillende pinnen door schuifexpressies te combineren.

De regel:

```
    PORTB |= 1<<6 | 1<<4 | 1<<2 | 1<<0;
```

kan natuurlijk ook vervangen worden door:

```
    PORTB |= 0x55;
```

Dit is misschien minder duidelijk maar wel minder typewerk ;-).

In de headerfile avr/io.h zijn alle namen van de verschillende bitjes in de I/O registers van de AVR met #define gekoppeld aan hun bitnummer. Op deze manier kun je dus met behulp van een schuifoperatie bitjes manipuleren en testen zonder dat je het bitnummer hoeft te weten (je moet dan natuurlijk wel de naam van het bitje weten).

Als je bijvoorbeeld wilt wachten tot het TOV1 bitje (Timer OVerflow 1 flag) in het TIFR register (TImer Flag Register) 1 wordt dan kan dit met de volgende C instructie:

```
while ((TIFR & 1<<TOV1) == 0); /* wacht tot TOV1 is set */
```

Je hoeft dan dus niet te weten welk bitnummer het TOV1 bitje heeft.

In de file avr/io.h wordt gekeken naar het ingestelde type AVR microcontroller om te bepalen welke bitnamen aan welke bitnummers moeten worden gekoppeld. Het is dus van groot belang om bij de project opties het juiste AVR type, in ons geval de ATmega32, te selecteren.

4 Les 2

Deze les gaat over timers en counters. Bij INLMIC ben je al eerder in aanraking gekomen met timers en counters, maar je hebt deze toen uitgelezen en bestuurd met behulp van assembler. Uiteraard zullen we dat nu gaan doen met behulp van C.

4.1 leerstof

De leerstof voor deze les is blz. 261, 262 en 10.3 uit [MNN13].

Als het goed is heb je 10.1 (t/m bladzijde 331) en 10.2 uit [MNN13] al doorgelezen bij INLMIC. Als iets je niet duidelijk is of als je iets niet begrijpt uit deze paragrafen, bestudeer dan deze paragrafen opnieuw!

Maak daarnaast ook het huiswerk dat aangegeven is aan het einde van de slides.

4.2 Opmerkingen [MNN13]

Ook nu zijn er wel weer een aantal op- of aanmerkingen te plaatsen bij hetgeen dat in [MNN13] staat.

Op blz. 261 wordt gezegd dat de compiler een delay loop weg kan optimaliseren. Dit is inderdaad waar. Het boek spreekt over het aanpassen van compiler opties om dit te voorkomen. De optimalisatie kan echter ook voorkomen worden door een volatile variabele te gebruiken (ook te zien in slides). Het plaatsen van het keywoord volatile voor een variabele vertelt de compiler dat de inhoud van de variabele altijd direct weggeschreven moet worden naar het geheugen. Dit wordt bij andere 'normale' variabelen namelijk niet altijd gedaan, want de inhoud wordt ook wel eens even vast gehouden in een register. De achterliggende gedachte is vaak dat de inhoud van volatile variabelen (en dus ook de geheugenlocatie van de variabele!) ook door de hardware gebruikt of aangepast kan worden en dus moeten bewerkingen aan deze variabelen altijd teruggeschreven worden naar het geheugen.

Doordat de inhoud van de loop variabele nu altijd weggeschreven moet worden naar het geheugen (vanwege volatile) kan de compiler geen optimalisaties meer toepassen en zal de loop dus niet meer weg worden geoptimaliseerd.

In paragraaf 10.3 zie je een aantal voorbeelden van hoe een delay functie gemaakt kan worden. In de slides zie je dat eigenlijk min of meer hetzelfde wordt gerealiseerd, maar dan op een andere manier (overflow bit wordt hier bijv. niet gebruikt). Je kan dus zoals je ziet op vele verschillende manieren gebruik maken van de timers.

5 Les 3

In deze les worden interrupts behandeld. Ook dit is al aan de orde geweest bij INLMIC en we gaan nu, zoals je waarschijnlijk al verwacht had, bekijken hoe je hier gebruik van kan maken in C.

5.1 leerstof

De leerstof voor deze les is 11.5 uit [MNN13] en paragraaf 5.3 op pagina 21 uit deze reader.

Bij INLMIC heb je de paragrafen 11.1, 11.3 en 11.4 uit [MNN13] moeten bestuderen. Ook hier geldt weer dat als iets je onduidelijk is je natuurlijk deze paragrafen weer moet bestuderen. In dit geval is de assembler wat minder interessant, maar de principes van een interrupt en ook het prioriteit-mechanisme (paragraaf 11.4) zijn daarentegen zeer belangrijk om te weten.

5.2 Opmerkingen [MNN13]

Opmerkingen over de leerstof uit [MNN13]:

- In de code voorbeelden van 11.5 is vaak te zien dat een hexadecimale waarde wordt weggeschreven naar een register. De regel

  ```
  TCCR0 = 0x06;   //Normal mode, falling edge, no prescaler
  ```

  is bijvoorbeeld te zien op bladzijde 388. Een dergelijke toekenning heeft niet echt de voorkeur, want je kunt beter het volgende doen:

  ```
  TCCR0 = 1<<CS02|1<<CS01;
  ```

  Dit levert uiteindelijk dezelfde constante op. Voordeel van deze constructie is dat je direct kunt zien welke bits worden ingesteld (het commentaar ernaast is in dit geval ook minder noodzakelijk). Ook hoeft je code niet meer gewijzigd te worden als de positie van de CS01 en CS02 bits veranderen in het TCCR0 register. Dit kan het geval zijn als je je code wilt laten draaien op een ander type microcontroller. Je hoeft je daarnaast ook geen zorgen te maken dat dit langzamer zou zijn, omdat de microcontroller extra zou moeten gaan shiften en een bitwise OR zou moeten doen. De compiler ziet van te voren namelijk dat 1«CS02|1«CS01 een waarde is die tijdens het compileren al kan worden bepaald en zal 1«CS02|1«CS01 omzetten naar 0x06. Je kunt dit trouwens ook controleren door naar de gegenereerde assembler te kijken
- Op bladzijde 387 is de volgende regel te zien:

  ```
  TCNT1H = (-640)>>8;
  TCNT1L = (-640);
  ```

  De reden voor het gebruik van deze regels is als volgt: De constante -640 (Let op, binair gezien zal het min-teken met de 2's complement notatie worden gerealiseerd) is groter dan 8 bits en zal in het TCNT1L (lower byte) en TCNT1H (higher byte) register moeten worden gezet. Voor de lower byte is dit vrij eenvoudig, want als je -640 toekent aan TCNT1L dan worden automatisch alleen de laagste 8 bits van deze

constante gepakt. Als je TCNT1H wilt vullen, dan wil je dit doen met de hoogste 8 bits en de laagste 8 bits wil je niet hebben. De laagste 8 bits kun je simpelweg verwijderen door de constante 8 bits naar rechts te schuiven.

Je kunt overigens beide regels versimpelen tot:

```
TCNT1 = -640;
```

Voor de enthousiastelingen: De compiler ziet dat TCNT1 een 16 bits register is, doordat TCNT1 een macro is die weer staat voor een pointer naar een 16 bits (integer) volatile variabele. De compiler zal uiteindelijk netjes assembler genereren die beide bytes van het TCNT1 register zal vullen met de juiste constante.

5.3 Bitn..... in C voor gevorderden

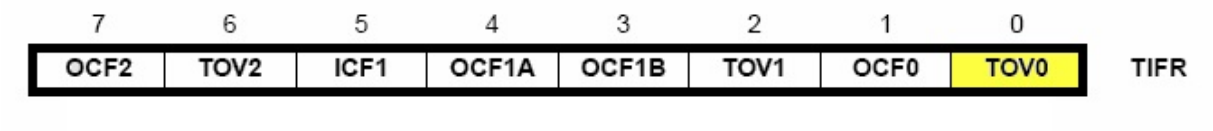
Bij bepaalde gebeurtenissen (events) zal de AVR microcontroller een bit in een van de I/O registers 1 maken. De programmeur kan de waarde van dit bitje testen (paragraaf 3.2.5 op pagina 10) om te kijken of de gebeurtenis is opgetreden. Het is ook mogelijk om de AVR bij zo'n gebeurtenis een interrupt op te laten wekken, maar we gaan er hier vanuit dat dit **niet** gedaan is. Een bit dat gebruikt wordt om een bepaalde gebeurtenis aan te geven wordt meestal een vlag (Engels: flag) genoemd. Al in de Romeinse tijd werden vlaggen namelijk gebruikt voor het doorgeven van signalen (gebeurtenissen) en ook nu worden vlaggen nog vaak gebruikt om signalen (gebeurtenissen) door te geven.

Figuur 1: Voorbeeld waarbij een vlag wordt gebruikt om een signaal door te geven :)

Als de vlag die door de AVR gehesen is, is opgemerkt door het programma dan moet het programma de vlag weer strijken (de bit moet weer 0 worden). Zodat de vlag bij de volgende gebeurtenis weer geset kan worden. Een vlag in een van de I/O register van de AVR kan gereset (gestreken) worden door een 1 naar de vlag te schrijven. **Raar maar waar!**

5.3.1 Voorbeeld

De Timer/Counter0 van de AVR maakt de TOV0 vlag in het TIFR register 1 als het TCNT0 register "overloopt" van 0xFF naar 0x00.



Figuur 2: Overzicht van de inhoud van het TIFR register.

Als de TOV0 vlag door de Timer/Counter0 1 gemaakt is dan kan de vlag weer 0 gemaakt worden door er een 1 naar toe te schrijven.

Je zou misschien verwachten dat dit geprogrammeerd moet worden als:

```
TIFR |= 0x01; /* onjuiste code! */
```

Je wilt toch alleen het TOV0 bitje setten (paragraaf 3.2.1 op pagina 7)?

Stel dat zowel de TOV0 als de TOV1 flag geset is. Het TIFR I/O register bevat dan de waarde 0x05. Als we nu de bovenstaande instructie uitvoeren dan wordt de waarde 0x05 $|0x01 =$ 0x05 naar het TIFR I/O register geschreven. Omdat een vlag bitje gereset wordt als er een 1 naar toe geschreven wordt, wordt nu dus niet alleen de TOV0 vlag maar ook de TOV1 vlag gereset. Dat is natuurlijk niet de bedoeling

De juiste code om alleen de TOV0 flag te resetten is:

```
TIFR = 0x01; /* juiste code! */
```

Of anders genoteerd:

```
TIFR = 1<<TOV0;
```

Stel dat zowel de TOV0 als de TOV1 flag geset is. Het TIFR I/O register bevat dan de waarde 0x05. Als we nu de bovenstaande instructie uitvoeren dan wordt de waarde 0x01 naar het TIFR I/O register geschreven. Omdat een vlag bitje gereset wordt als er een 1 naar toe geschreven wordt, wordt nu dus alleen de TOV0 vlag gereset.

6 Les 4

Centraal staat de ADC (Analog to Digital Converter) in deze les.

6.1 leerstof

- Bladzijde 169 (paragraaf 5.11) uit [KP09]

- Bladzijde 463 t/m 466 uit [MNN13] (exclusief Parallel versus serial ADC)
- Bladzijde 469 t/m 483 uit [MNN13]

Werp daarnaast ook eens een blik op het ADC hoofdstuk in de ATmega32A datasheets:

`http://www.atmel.com/Images/Atmel-8155-8-bit-Microcontroller-AVR-ATmega32A_Datasheet.pdf`

6.2 Opmerkingen [MNN13]

Wederom zijn er een aantal opmerkingen te plaatsen over de leerstof uit [MNN13]:

- Bladzijde 467 gaat over externe ADC's en is voor dit vak niet zo interessant. Mocht je toch geïnteresseerd zijn, dan kun je dit natuurlijk wel doorlezen.
- Als je het interessant vindt om te weten hoe een ADC intern werkt (dit is slechts één methode) dan kun je bladzijde 468 doorlezen.
- In figuur 11 op blz. 475 is het circuit dat voor de AND poort zit niet van belang voor nu. Dit heeft te maken met de auto trigger van de ADC en is voor nu nog niet belangrijk.
- De assembly voorbeelden die te zien zijn kun je overslaan.
- In paragraaf 3 wordt een temperatuursensor als voorbeeld genomen voor een sensor die men aansluit op de ADC. Er bestaan tegenwoordig ook vele temperatuursensoren die geen analoge spanning afgeven, maar die via een digitaal protocol hun temperatuur kunnen doorgeven aan de microcontroller. Later zullen we hier nog een voorbeeld van zien.

7 Les 5

In deze les wordt er ingegaan op hoe een ADC kan worden gebruikt vanuit C-code. Je hebt als het goed is dit al gelezen in [MNN13]. In de les worden dus enkele dingen hier uit herhaald en er wordt daarnaast wat achtergrondinformatie en andere voorbeelden gegeven.

Allicht wil je de spanning die in een variabele staat ook tonen op het LCD. Om karakters op het LCD te tonen kunnen verschillende functies gebruikt worden die in het practicum worden gegeven. Deze functies kunnen echter alleen een string op het LCD zetten. Als je data die in een integer of float variabele staat wil tonen op het LCD, dan zal je deze om moeten zetten naar een string. Hiervoor kan de *snprintf* functie worden gebruikt.

7.1 snprintf functie

De functie snprintf kan worden gebruikt om data geformatteerd naar een C-string te sturen. Het prototype van deze functie is gegeven in *<stdio.h>*:

```
int snprintf (char* s, size_t n, const char* fmt, ...);
```

Deze functie formatteert de na *fmt* meegegeven parameters op de manier zoals in de format string *fmt* is opgegeven. Het resultaat wordt weggeschreven op de plaats die door de *char**

s wordt aangewezen en er worden maximaal *n* karakters weggeschreven (het laatste weggeschreven karakter is altijd het `'\0'` karakter). De functie geeft het aantal geformatteerde karakters (ook inclusief `'\0'`) terug. Als de return-waarde groter is dan *n* dan was er dus niet voldoende ruimte.

Deze functie kan dus gebruikt worden om data te formatteren en naar een *char* array te sturen op dezelfde manier als de bekende *printf* functie gebruikt kan worden om data te formatteren en naar het beeldscherm te sturen. Bij het programmeren van microcontrollers is er meestal geen ”normaal” beeldscherm (*stdout*) aanwezig. De functie *snprintf* kan dan gebruikt worden om data te formatteren in een C-string. Deze string kan dan vervolgens op een eenvoudig LCD display weergegeven worden of via een seriële verbinding naar een PC of andere microcontroller verstuurd worden.

De avr-libc library bevat diverse varianten van de *printf* functie waaronder *snprintf*. Onderaan pagina 188 en op pagina 189 en 190 in avr-libc-user-manual.pdf staat beschreven hoe de format string *fmt* gebruikt kan worden.

Ter info: Microsoft Visual Studio 2010 bevat de functie *snprintf* niet (en voldoet dus niet aan de C99 standaard!), zie `http://stackoverflow.com/questions/2915672/snprintf-and-visual-studio-2010`. In Microsoft Visual Studio 2010 kun je wel gebruik maken van de functie *sprintf* maar bij deze functie ligt altijd het gevaar van een buffer overrun op de loer. Zie: `http://bd.eduweb.hhs.nl/micprg/string.htm#buffer_overflow`

7.1.1 Voorbeeld met een int

Hieronder is een AVR gcc programma gegeven dat een int formatteert in een C-string als decimaal getal en rechts uitlijnt op een veldbreedte van 6 karakters met de format string ”%6d”. De C-string wordt vervolgens op een LCD display weergegeven met behulp van de LCD functies.

```
#include <stdio.h>
#include "lcd.h"

int main(void) {
    int i = 0;
    char buffer[11];
    lcd_init();
    lcd_cursor(false, false);
    while (1) {
        lcd_home();
        snprintf(buffer, sizeof buffer, "i = %6d", i++);
        lcd_puts(buffer);
    }
    return 0;
}
```

Listing 19: Afdrukken van een int op het LCD display.

De operator *sizeof* bepaalt de grootte van de variabele buffer in bytes. Dit is dus het maximale aantal karakters wat *snprintf* mag wegschrijven, omdat anders voorbij het einde van de variabele buffer wordt geschreven.

7.1.2 Voorbeeld met een uint16_t

In programma's voor microcontrollers wordt vaak gebruik gemaakt van de integer types die gedefinieerd zijn in C99 (in de header file stdint.h). Zie: `http://en.wikibooks.org/wiki/C_Programming/C_Reference/stdint.h` en/of pagina 164 evt. in avr-libc-user-manual.pdf. Door gebruik te maken van deze types kunnen we exact bepalen hoe groot (hoeveel bits) een integer variabele is. Als we de C99 integer types willen printen dan moeten we gebruik maken van de formats die in de C99 include file *inttypes.h* zijn gedefinieerd. Zie: `http://en.wikibooks.org/wiki/C_Programming/C_Reference/inttypes.h` en/of pagina 137 evt. in avr-libc-user-manual.pdf. Voor een variabele van het type *uint16_t* is het bijbehorende print format gedefinieerd in de macro *PRIu16*. Deze macro definieert een C-string literal en kan via C-string literal concatination gecombineerd worden met de rest van de format string.

Hieronder is een AVR gcc programma gegeven dat een *uint16_t* formatteert in een C-string als decimaal getal en rechts uitlijnt op een veldbreedte van 5 karakters met de format string "%5" PRIu16. De C-string wordt vervolgens op een LCD display weergegeven met behulp van de LCD functies.

```
#include <stdio.h>
#include <inttypes.h>
#include "lcd.h"

int main(void) {
    int i = 0;
    char buffer[11];
    lcd_init();
    lcd_cursor(false, false);
    while (1) {
        lcd_home();
        snprintf(buffer, sizeof buffer, "i = %5"PRIu16, i++);
        lcd_puts(buffer);
    }
    return 0;
}
```

Listing 20: Afdrukken van een uint16_t op het LCD display.

7.1.3 Voorbeeld met een double

Hieronder is een AVR gcc programma gegeven dat een double formatteert in een C-string als floatingpoint getal en rechts uitlijnt op een veldbreedte van 8 karakters met 6 cijfers achter de decimalpoint met de format string "%8.6lf". De C-string wordt vervolgens op een LCD display weergegeven met behulp van de LCD functies.

```
#include <stdio.h>
#include <math.h>
#include "lcd.h"

int main(void) {
    char buffer[14];
    lcd_init();
    lcd_cursor(false, false);
    snprintf(buffer, sizeof buffer, "pi = %8.6lf", M_PI);
    lcd_puts(buffer);
    while (1);
    return 0;
}
```

Listing 21: Afdrukken van een double op het LCD display.

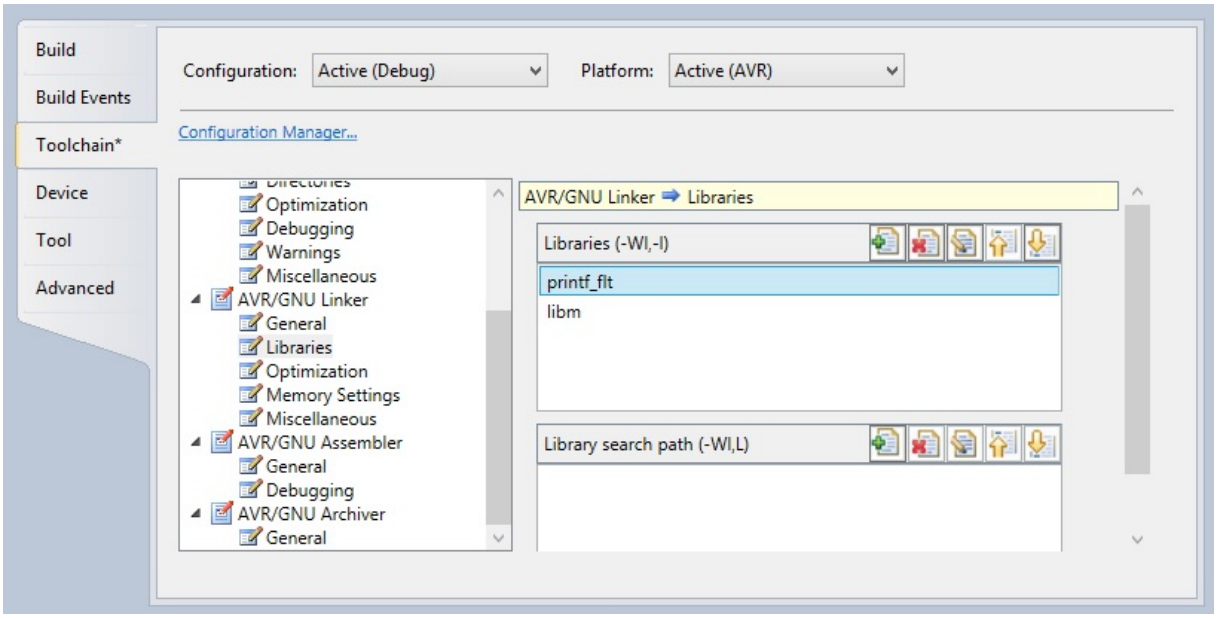
De constante M_PI is gedefinieerd in <math.h>. Zie pagina 155 in avr-libc-user-manual.pdf.

De uitvoer van dit programma is:

```
pi =      ?
```

Om doubles (en floats) te kunnen gebruiken moet je het volgende doen in Atmel Studio 6:

- Klik op Project, daarna op Properties (Alt+F7) en selecteer het tabblad Toolchain. Selecteer vervolgens Libraries onder AVR/GNU Linker. Voeg nu de library *printf_flt* toe **boven** *libm* (zie Figuur 7.1.3). De volgorde is dus belangrijk!

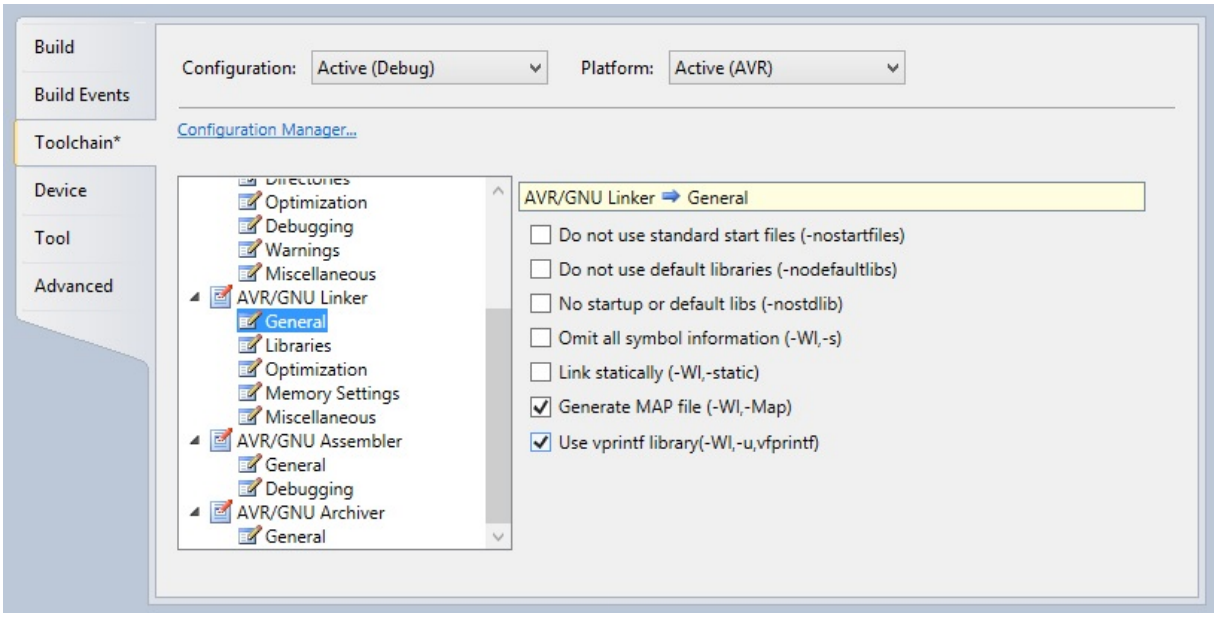


Figuur 3: Screenshot waarin de te linken assembler files te zien zijn

- Selecteer General onder AVR/GNU Linker. Zet een vink bij *Use vprintf library(-Wl,-u,vfprintf)*

De uitvoer van het programma is nu wel correct:

pi = 3.141593



Figuur 4: Aanpassen van linker options

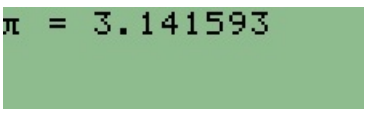
7.1.4 Gebruik van het griekse karakter pi

De LCD wordt aangestuurd met een een Hitachi HD44780 controller. Op pagina 17 van dit document kun je de karakterpatronen vinden. Je ziet dat de griekse letter pi kan worden weergegeven met karaktercode 0xF7. In het volgende programma maken we hier gebruik van:

```
#include <stdio.h>
#include <math.h>
#include "lcd.h"

int main(void) {
    char buffer[13];
    lcd_init();
    lcd_cursor(false, false);
    snprintf(buffer, sizeof buffer, "%c = %8.6lf", 0xF7, M_PI);
    lcd_puts(buffer);
    while (1);
    return 0;
}
```

Listing 22: Afdrukken van het karakter π op het LCD display.

De uitvoer van dit programma is:



Figuur 5: Screenshot van het LCD display

Helaas geeft de LCD simulatie van de AVE AVR Plugins het teken pi niet weer :-(

8 Les 6

In les 6 en les 7 staat het genereren van PWM signalen centraal. In les 6 wordt het genereren van een PWM signaal m.b.v. de output compare behandeld. Hierbij kan de timer in normal en CTC mode worden gezet.

8.1 Leerstof

Op bladzijde 328 t/m 331 staat uitgelegd hoe de timer in CTC mode werkt. Op zich heb je dit bij INLMIC ook al eens moeten bestuderen, maar het is allicht verstandig om dit nog een keer door te lezen.

In 16.1 staat hoe een blokgolf in normal mode en CTC mode kan worden gegenereerd. Er staan in deze paragraaf geen C-code voorbeelden. Deze zijn wel te vinden in 16.4.

Verder dien je 17.1 en 17.2 te bestuderen.

8.2 Opmerking [MNN13]

In 17.1 wordt de H-brug (H-bridge) behandeld. Te zien is hoe je met een schakeling vanuit de AVR een DC motor kan aansturen. Je kan uiteindelijk met behulp van PWM ook de draaisnelheid van de DC motor beïnvloeden. De reden dat de H-brug wordt behandeld is dat het een voorbeeld is van hoe je PWM in de praktijk kunt gebruiken. Het is belangrijk om te begrijpen wat het principe is van een H-brug, maar de ins- en outs van de schakelingen zijn weer niet het allerbelangrijkst voor dit vak. Je zult op de toets dan ook niet bevraagd worden over de details van specifieke H-brug schakelingen.

9 Les 7

In deze les wordt PWM verder behandeld. Aan bod komen de Fast-PWM en de phase correct PWM.

9.1 Leerstof

Bladzijde 575, 576 (figuur 31), 583 en 17.4 uit [MNN13].

9.2 Opmerking bladzijden 575, 576 en 583

Timer/counter 1 heeft zoals te zien is op bladzijde 575 vele verschillende modi. Belangrijk om op zijn minst te weten is modus 14. Je kunt in deze modus zowel de PWM periode (met ICR1) en de duty cycle (OC1A of OC1B) instellen. Je kunt hiermee dus een PWM signaal maken met een zeer specifieke frequentie en duty cycle. Met name doordat timer/counter1 16 bits groot is kun je bijna iedere frequentie en duty cycle genereren.

Zorg er dus voor dat je de voorbeelden op bladzijde 583 over modus 14 begrijpt.

10 Les 8

Centraal in deze les staat input capturing. Met behulp van input capturing kun je bijvoorbeeld de tijd die een puls duurt meten of de periode van een PWM signaal. Daarnaast komt kort aan bod het tellen van pulsen.

10.1 Leerstof

Paragraaf 16.3 uit [MNN13]. Je hoeft niet te kijken naar de assembler voorbeelden in deze paragraaf, want in 16.4 zijn dezelfde programma's te vinden, maar dan in C.

10.2 Opmerking over 16.3 uit [MNN13]

Het onderschrift bij figuur 18 klopt niet. Afgebeeld is **niet** de inhoud van TCCR1B, maar de inhoud van het ACSR register.

11 Les 9

In les 9 en les 10 staat communicatie tussen de microcontroller en andere devices centraal. Met devices bedoelen we in dit geval o.a. sensoren en actuatoren buiten de microcontroller en ook andere microcontrollers. De communicatie zal altijd worden bewerkstelligd met behulp van een communicatiestandaard. Het eerste doel van deze lessen is om een klein idee te krijgen over wat de verschillende standaarden zijn en wat ook de grootste verschillen onderling zijn.

Er zijn vele communicatiestandaarden en omwille van de tijd behandelen we er maar twee in detail. UART en SPI zullen worden behandeld in les 9 en 10, omdat dit twee standaarden zijn die veel gebruikt worden in embedded systemen.

Bij inleiding datacommunicatie heb je natuurlijk ook al het een en ander geleerd over communicatie. In eerste instantie heb je geleerd hoe applicaties met elkaar kunnen communiceren en ook hoe datapakketten kunnen worden getransporteerd van de ene host naar de andere host. Kort samengevat heb je je dus bezig gehouden met de applicatie-, transport- en netwerklaag uit het OSI lagenmodel. Belangrijk om te weten is dat je je nu bezig gaat houden met de fysieke en datalinklaag als we naar het OSI lagen model kijken. We gaan ons nu dus ook bezig houden met communicatie, maar je zult zien dat de communicatietechnieken die we gaan gebruiken op een veel lager niveau zitten dan wat je bij inleiding datacommunicatie te zien krijgt.

We gaan ons concreet bezighouden met hoe we elektrisch gezien ervoor kunnen zorgen dat we überhaupt kunnen communiceren (fysieke laag) en ook hoe we uiteindelijk hiermee bitjes kunnen versturen (ook fysieke laag). Een laag erboven (datalink laag) houden we ons bezig met hoe we bytes we kunnen sturen en hoe enige error checking kan worden gedaan. Voor beide zaken kunnen we veel verschillende protocollen gebruiken en in dit geval bespreken we RS232 en SPI.

11.1 Leerstof

Paragraaf 12.1 t/m 12.5 uit [MNN13]. Let goed op de opmerkingen hieronder, want sommige dingen uit deze paragrafen hoef je niet te weten.

11.2 Opmerkingen 12.1 t/m 12.5

Er zijn nogal wat opmerkingen te plaatsen over hoofdstuk 12. Deze zijn als volgt:

- De tekst op bladzijde 397 die gaat over synchronous en asynchronous **klopt niet**. Bij synchrone datacommunicatie is er een apart (klok)signaal waaruit afgeleid kan worden bij de ontvanger op welk tijdstip een '0' of een '1' op de lijn is gezet. SPI, die je dus de volgende les ziet, gebruikt een dergelijke synchrone communicatie. Bij asynchrone communicatie is er dus geen signaal dat aangeeft wanneer er op welk tijdstip een bit geplaatst is op een lijn. De ontvanger zal zelf moeten timen met een eigen klok wanneer een bepaald bitje is gearriveerd. RS232 is hier een voorbeeld van.
- Het is duidelijk te zien dat het erg lang geleden is dat hoofdstuk 12 geschreven is. Vermoedelijk is het hoofdstuk halverwege jaren 90 geschreven. Onderaan bladzijde 397 wordt een voorbeeld genoemd die gaat over een printer en een simplex verbinding. Voor printercommunicatie wordt natuurlijk al lang geen simplex verbinding meer gebruikt, maar een full duplex verbinding zoals USB.
- De tweede alinea bij de paragraaf *Data transfer rate* kun je overslaan. Waarschijnlijk zie je zelf ook wel dat deze tekst uit grootvaders tijd komt :)
- De paragrafen *Data communication classification* en *Examining RS232 handshaking signals* kun je overslaan. De term DCE en DTE zie je nog wel staan in de slides, dus als je het interessant vindt om deze termen te begrijpen, dan kun je de eerste paragraaf nog doorlezen, maar dit is verder eigenlijk niet van belang.

  De paragraaf *Examining RS232 handshaking signals* gaat over de betekenis van de verschillende RS232 pinnen, maar velen zijn helemaal niet interessant om te weten, want ze worden in de praktijk niet meer gebruikt. De niet meer gebruikte pinnen hebben ooit bestaan, omdat ze in de jaren '70 en '80 nog wel gebruikt werden.
- Figuur 7 klopt niet. PD0 van de ATmega32 moet worden aangesloten op pin 12 van de MAX232 en PD1 op pin 11. Zie ook dit voorbeeld. In de tekst wordt ook gesproken over een null modem connection. Een null modem connectie is een aansluiting waarbij de RXD van het ene device wordt aangesloten op de TXD van het andere device. Uiteraard wordt TXD ook weer aangesloten op de RXD van het andere device. (zie ook `http://en.wikipedia.org/wiki/Null_modem`) Om dit te kunnen bewerkstelligen zorg je voor deze aansluiting in de kabel of in de stekker, maar niet zoals weergegeven is in figuur 7 voor de MAX232.
- Halverwege bladzijde 405 wordt gesproken over *HyperTerminal*. Dit is een programma dat nog bestond op Windows98 en WindowsXP. Als je met Windows 7 of 8 wilt communiceren via RS232 met een ander RS232 device, dan kun je hiervoor *putty* gebruiken. In de slides is hiervan ook een screenshot te zien.
- Op bladzijde 409 wordt gezegd dat er gecommuniceerd gaat worden via synchronous mode. Als je uiteindelijk via RS232 gaat communiceren, dan heeft deze mode niet

zoveel zin, want RS232 is asynchroon. Je hoeft dan dus niet de UMSEL bit te setten. Voor de enthousiastelingen: Je kan echter nog wel gebruik maken van de synchronous mode, maar dat heeft tot gevolg dat er een synchronisatiekarakter wordt verstuurd. Zie `http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#Synchronous_transmission`. Dit kan echter onwenselijk zijn bij een aantal RS232 toepassingen.

- Uiteraard kun je de voorbeelden in assembly overslaan en de C voorbeelden bekijken in paragraaf 4 op bladzijde 419. Helaas lijkt example 7 er weer niet te zijn in C :(.
- De paragraaf op bladzijde 416 met de titel *Doubling the baud rate in the AVR* is niet van belang en mag je overslaan.
- Het antwoord op de vraag op bladzijde 423 is natuurlijk dat als je nieuwe data in het UDR register kan zetten, nog niet per se de voorgaande data al helemaal verzonden is.

12 Les 10

Centraal in deze les staat SPI.

12.1 Leerstof

Paragraaf 18.1 en 18.2.

12.2 Opmerkingen 18.1 en 18.2

- Het onderste stuk tekst op bladzijde 604 over de 3 wire interface is voor nu niet interessant en kun je overslaan.
- In figuur 4 wordt de read gedaan op de falling edge (SPI mode 1).
- In de tekst staat niet echt duidelijk vermeld dat *CE* (Chip Enable) in principe hetzelfde is als het *SS* (Slave Select) signaal. *CE* is de naam die wordt gebruikt aan de master zijde en *SS* aan de slave zijde. Daarnaast wordt overigens de naam *CE* niet helemaal correct genoteerd, want eigenlijk zou deze als $\overline{CE}$ moeten worden genoteerd. (*CE* staat voor Chip Enable en als deze dus 1 is, dan zou de 'Chip' dus 'ge-enabled' worden. De werking is precies andersom, dus vandaar dat er een inverteer streep boven moet staan)
- Interessant is het om te zien dat bij het SPI protocol er ook wordt voorgeschreven wat de vorm van de data is die over wordt verstuurd (dus bijvoorbeeld eerst adres versturen en daarna bijhorende data). Dit is eigenlijk iets dat niet wordt vastgelegd bij de datalink en fysieke laag, maar in hogere lagen als je kijkt naar het OSI model. Je zou verwachten dat SPI alleen zaken definieert voor de datalink en fysieke laag, maar dat is dus niet het geval.
- Voor de enthousiastelingen: De *SS* pin is er ook voor bedoeld om het mogelijk te maken om meerdere SPI devices aan te kunnen sluiten op dezelfde *MISO* en *MOSI* lijnen. De microcontroller kan dan van een device de *SS* (ofwel $\overline{CE}$) pin laag maken

(en de andere devices hoog) en dan zal alleen met het desbetreffende device worden gecommuniceerd.

13 Les 11

In deze les komen maar liefst drie zaken aan bod: Strings, Structs en Files. Strings heb je al gehad bij GESPRG, dus wat je in de les ziet over strings is een herhaling van hetgeen dat je al eerder hebt gezien.

Het nut van het kunnen gebruiken van strings in microcontroller programma's zal inmiddels wel duidelijk zijn geworden (denk aan bijvoorbeeld de LCD). Structs zullen we in volgende lessen nodig hebben bij bijvoorbeeld de Real Time Clock (RTC) en files kom je weer tegen in het practicum.

13.1 Leerstof

Paragraaf 6.9 en 6.10 uit [KP09] heb je als het goed is al eens door moeten lezen bij GESPRG. Als de inhoud je hier niet meer van bekend is of als de werking van strings je niet meer bekend is, bestudeer dan deze paragrafen.

Daarnaast dienen paragrafen 9.1, 9.2, 9.3, 9.5 en 9.6 uit [KP09] te worden bestudeerd. Mocht je iets uit deze paragrafen niet begrijpen, dan kun je ook kijken naar de uitleg die hier wordt gegeven over Strings en hier over structs.

Als laatst dient ook nog paragraaf 13.3 op pagina 33 te worden bestudeerd.

Let nog wel even op de opmerkingen die hieronder staan over de te lezen paragrafen.

13.2 Opmerkingen over 9.1, 9.2, 9.3, 9.5 en 9.6

- Bladzijde 294: De stukken die gaan over een pointer naar een functie (alles wat te maken heeft met *PFD*) kun je overslaan en hoef je niet te weten.
- Bladzijde 296: Je kunt hier duidelijk zien dat het boek een vertaling is uit het Engels. De woorden *lidmaat* en *naamplaatje* zijn misschien een beetje ongelukkig vertaald, want doorgaans gebruikt men in Nederland gewoon de Engelse namen *member* en *tag name* in plaats van de vertaling die je hier ziet.
- Bladzijde 297: Onderaan deze bladzijde wordt een zogenaamde syntaxis gegeven waaruit een struct declaratie kan bestaan. Dit is een formele manier van beschrijven van hoe een struct declaratie er uit ziet. Het is echter wel erg lastig te begrijpen als je de notatiewijze niet begrijpt. Je kunt daarom dit stuk overslaan en je richten op de voorbeelden die later worden gegeven.
- Bladzijde 305: Wij maken gebruik van ANSI C, dus je kunt ook *auto*-variabelen van een beginwaarde voorzien. *auto*-variabelen zijn overigens 'normale' variabelen die je bijvoorbeeld declareert in een functie. (Zie ook blz. 166 in [KP09])

13.3 Lezen uit en schrijven in tekstfiles

De programmeertaal C onderscheidt 2 soorten files: tekstfiles en binaire files. Deze paragraaf behandelt alleen tekstfiles. Meer informatie over binaire files kun je als je wilt hier vinden. Desondanks sommen we even op wat de verschillen zijn tussen tekstfiles en binaire files:

- **Tekstfiles**: In deze files is alle informatie opgeslagen in ASCII codering (of een andere karaktercodering). Deze files kun je met elke willekeurige teksteditor lezen. Het getal 123 wordt in een tekstfile dus opgeslagen als 3 ASCII karakters. De inhoud van de file in hexadecimale notatie is dan 31 32 33. Tekstfiles worden ook wel sequentiële files genoemd. Als je informatie uit een bepaalde plaats uit de file wilt lezen (bijvoorbeeld de 150ste regel) dan moet je alle voorgaande informatie lezen omdat alle regels een verschillende lengte (kunnen) hebben. Lezen uit tekstfiles gaat met de functie: getc of fscanf. Schrijven in tekstfiles gaat met de functie: putc of fprintf.
- **Binaire files**: In deze files is alle informatie opgeslagen in binaire codering. Deze files kun je wel openen met een teksteditor maar de informatie is dan onleesbaar (allemaal vreemde karakters). Het getal 123 wordt in een tekstfile opgeslagen als 4 bytes (ervan uitgaande dat het getal opgeslagen is in een 32 bits int variabele). De inhoud van de file in hexadecimale notatie is dan 00 00 00 7B. Binaire files worden ook wel random access files genoemd. Als je ervoor zorgt dat de file ingedeeld is in blokken van gelijke grootte dan kun je snel het blok met een specifieke index zoeken zonder dat je alle voorgaande blokken moet inlezen. Lezen uit binaire files gaat met de functie: fread. Schrijven in binaire files gaat met de functie fwrite. Zoeken gaat met de functie fseek.

Je kunt de volgende bewerkingen op tekstfiles uitvoeren:

13.3.1 Tekstfile openen

Om een tekstfile te kunnen lezen of beschrijven moet de file eerst geopend worden. Je kunt een file openen met de functie *fopen*. Deze functie is in de include file *stdio.h* als volgt gedeclareerd:

```
FILE *fopen(const char *filename, const char *mode);
```

Het returntype is een *FILE**. Dit wordt een filepointer genoemd. Deze filepointer kun je, als de file eenmaal geopend is, gebruiken om uit de file te lezen of om in de file te schrijven. Deze pointer wijst niet rechtstreeks naar de file. Dat kan namelijk niet want de file bevindt zich niet in het werkgeheugen (RAM) van de PC maar in het achtergrond geheugen (bijvoorbeeld op de harddisk). De filepointer wijst naar een zogenaamde file control structure waarin informatie wordt bijgehouden waarmee het operating systeem de echte file kan benaderen. Een filepointer zou dus eigenlijk file control structure pointer genoemd moeten worden.

Via de parameter *filename* kun je de naam van de file die je wilt openen meegeven. Als je alleen een naam opgeeft wordt de file in het ”huidige” directory opgezocht. Je kunt ook de relatieve of absolute padnaam opgeven. Let er daarbij op dat de backslash in C strings

gebruikt wordt als escape karakter. Één backslash moet dus worden ingetyped als twee backslashes. Bijvoorbeeld *"C:\\Mijn documenten\\testbestand.txt"*.

Via de parameter mode kun je opgeven hoe je de file wilt openen:

- **"r"** Open om te lezen (read). Als de file niet bestaat is dat een fout.
- **"w"** Open om te schrijven (write). Als de file niet bestaat dan wordt deze aangemaakt. Als de file al bestaat wordt deze file overschreven.
- **"a"** Open om te schrijven aan het einde van de file (append). Als de file niet bestaat dan wordt deze aangemaakt.
- **"r+"** Open om te lezen en te schrijven. Als de file niet bestaat is dat een fout.
- **"w+"** Open om te lezen en te schrijven. Als de file niet bestaat dan wordt deze aangemaakt. Als de file al bestaat wordt deze file overschreven.
- **"a+"** Open om te lezen en te schrijven aan het einde van de file. Als de file niet bestaat dan wordt deze aangemaakt.

Als alles goed gaat geeft *fopen* een geldige *FILE** terug. Als er een fout optreedt (bijvoorbeeld omdat de file die je wilt lezen niet bestaat of omdat je een file wilt schrijven naar een CD-ROM) geeft *fopen* de waarde *NULL* terug. *NULL* is gedefineerd in *stdio.h*.

13.3.2 Tekstfile sluiten

Nadat de filepointer gebruikt is om uit de file te lezen of om in de file te schrijven moet de file weer worden gesloten. Het is belangrijk om een file zo snel mogelijk te sluiten omdat andere applicaties de file niet kunnen openen zolang de file al geopend is. Als een programma eindigt zal het operating systeem alle open files zelf sluiten. Het is echter netter als het programma dit zelf doet met de functie *fclose*. Deze functie is in de include file *stdio.h* als volgt gedeclareerd:

```
int fclose(FILE *stream);
```

Deze functie geeft de integer waarde *EOF* terug als er een fout optreed. Als parameter moet een *FILE** worden meegegeven die je hebt teruggekregen van de functie *fopen*.

13.3.3 Tekstfile lezen

Er zijn verschillende functies beschikbaar om uit een tekstfile te lezen. Allereerst behandelen we *getc*.

getc

De eenvoudigste functie is *getc*. Deze functie leest 1 karakter uit de file en is in de include file *stdio.h* als volgt gedeclareerd:

```
int getc(FILE *stream);
```

Als parameter moet een *FILE** worden meegegeven die je hebt teruggekregen van de functie *fopen*.

Deze functie geeft geen char terug maar een int! Als het lukt om een karakter uit de file te lezen geeft *getc* (de ASCII waarde van) het ingelezen karakter terug. De functie geeft de speciale integer waarde *EOF* terug als er iets fout gaat. *EOF* is gedefinieerd in *stdio.h*. Doordat de functie een int teruggeeft kan elke char waarde worden teruggegeven plus de speciale errorcode *EOF*.

Let goed op:

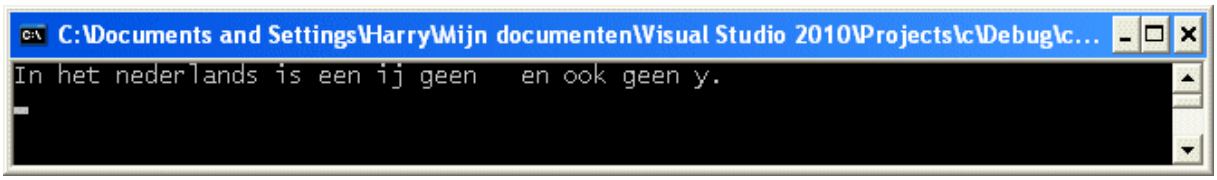
- **EOF is geen karakter!** Aan het einde van de file staat geen speciaal karakter. Het operating systeem houdt voor elke file het aantal bytes in de file bij (de filelengte). Op deze manier kan een file alle mogelijke bytewaarden bevatten. Op het moment dat een programma meer karakters probeert te lezen als de filelengte geeft de functie getc de integer waarde *EOF* terug.
- **EOF kun je niet opslaan in een char!** *EOF* is meestal gedefinieerd als de integer -1. Deze waarde is hexadecimaal FF FF FF FF. Als je deze waarde in een char variabele stopt passen daar alleen de laatste 8 bits in. Deze char variabele wordt dan gelijk aan hexadecimaal FF. Het hangt van de karakterset af welk karakter dit is. Op de PC is dit voor de meeste lettertypes het karakter ÿ en dat is dus heel wat anders dan de integer waarde *EOF*.
- **Sla de returnwaarde van getc altijd op in een int!** Een veelgemaakte fout is het opslaan van de returnwaarde van *getc* in een char. Dat is niet goed omdat deze waarde dan niet meer correct met *EOF* vergeleken kan worden. Zie onderstaand voorbeeld.

Het programma showfile.c leest de file testfile.txt karakter voor karakter in en schrijft deze karakters naar het beeldscherm.

```
#include <stdio.h>

int main(void) {
    int c;
    FILE *fp;
    fp = fopen("testfile.txt", "r");
    if (fp == NULL) {
        printf("Kan file testfile.txt niet openen!");
    }
    else {
        c = getc(fp);
        while (c != EOF) {
            putchar(c);
            c = getc(fp);
        }
        fclose(fp);
    }
    getchar();
    return 0;
}
```

Listing 23: Karakter voor karakter inlezen van een file.

Als in de file testfile.txt de volgende regel staat: ”In het Nederlands is een ij geen ÿ en ook geen y.” Dan wordt deze regel als volgt afgedrukt op het scherm:



Figuur 6: Uitvoer van showfile.c

De ÿ wordt niet afgedrukt omdat dit karakter in het, in het console window gebruikte, lettertype niet bestaat.

Als in het bovenstaande programma de regel:

```
int c;
```

vervangen wordt door:

```
char c;
```

dan geeft dit programma de volgende uitvoer:

Figuur 7: Uitvoer van een aangepaste showfile.c

Je ziet dat het programma nu stopt **voordat** het eind van de file bereikt is! Deze fout wordt veroorzaakt doordat het karakter ÿ wordt aangezien voor *EOF*.

fscanf

In plaats van karakter voor karakter in te lezen kun je ook zogenaamde geformatteerde invoer doen. Dit houdt in dat de ingelezen karakters worden omgezet naar het gewenste type. Je bent al bekend met geformatteerde invoer vanaf het toetsenbord met de functie *scanf*. Geformatteerde invoer vanuit een file gaat op vergelijkbare wijze met de functie *fscanf*. Deze functie is in de include file *stdio.h* als volgt gedeclareerd:

```
int fscanf(FILE *stream, const char *format, ...);
```

Als eerste parameter moet een *FILE** worden meegegeven die je hebt teruggekregen van de functie *fopen*. De overige parameters zijn hetzelfde als bij de al bekende functie *scanf*.

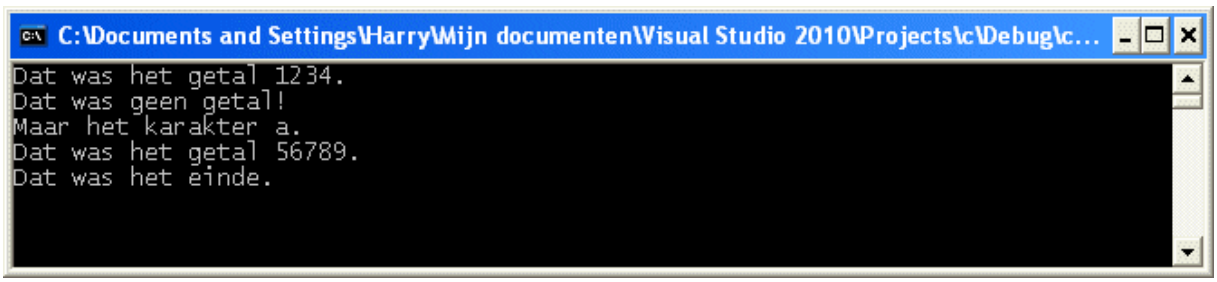
De functie geeft de integer waarde *EOF* terug als er iets fout gaat bij het inlezen. Anders geeft de functie het aantal variabelen terug dat succesvol is geconverteerd en ingelezen. Zie onderstaande voorbeeldprogramma scan.c:

```
#include <stdio.h>

int main(void) {
    int i;
```

```
    int ret;
    FILE *fp;
    fp = fopen("getallenfile.txt", "r");
    if (fp == NULL) {
        printf("Kan file getallenfile.txt niet openen!");
    }
    else {
        do {
            ret = fscanf(fp, "%d", &i);
            switch (ret) {
                case 0:
                    printf("Dat was geen getal!\n");
                    printf("Maar het karakter %c.\n", getc(fp));
                    break;
                case 1:
                    printf("Dat was het getal %d.\n", i);
                    break;
                case EOF:
                    printf("Dat was het einde.\n");
                    break;
                default:
                    printf("Dat kan niet!\n");
                    break;
            }
        }
        while (ret != EOF);
        fclose(fp);
    }
    getchar();
    return 0;
}
```

Listing 24: Het inlezen van gehele getallen uit een file.

Als in de file getallenfile.txt de volgende regel staat: ”1234 a56789” Dan wordt het volgende afgedrukt op het scherm:



Figuur 8: Uitvoer van scan.c

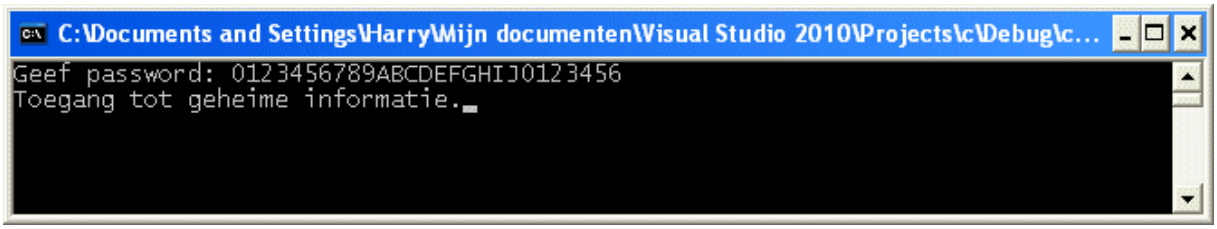
Met behulp van *scanf* en *fscanf* kun je ook woorden inlezen vanaf het toetsenbord of vanuit een file. Deze woorden worden dan opgeslagen in een char array. Hierbij moet je goed oppassen voor buffer overrun. Dit probleem treedt op als de invoer meer karakters bevat als de char array kan bevatten. Voorbeeld:

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char password[] = "Geheim", buffer[10];
    printf("Geef password: ");
    scanf("%s", buffer);
    if (strncmp(password, buffer, 7) == 0) {
        printf("Toegang tot geheime informatie.");
    }
    else {
        printf("Onjuist password!");
    }
    fflush(stdin);
    getchar();
    return 0;
}
```

Listing 25: Een programma waarin een gevaar op de loer ligt.

De functie *strncmp* hebben we nog niet behandeld. Deze functie uit *string.h* vergelijkt een aantal karakters van twee als parameters meegegeven strings. Het aantal karakters dat vergeleken moet worden wordt als derde parameter meegegeven. Deze functie geeft 0 terug als de strings gelijk zijn (waarbij alleen gekeken wordt naar het aantal meegegeven karakters).

Je zou denken dat je alleen toegang tot de geheime informatie krijgt als je het password kent[1]. Als dit programma met Microsoft Visual C++ wordt gecompileerd blijkt dat de variabele password 20 bytes na de variabele buffer in het geheugen wordt geplaatst. Als we meer dan 10 karakters invoeren worden deze karakters allemaal in het geheugen weggeschreven (buffer overrun). Op deze manier kan ook de variabele password overschreven worden.



Figuur 9: Uitvoer van bufferoverrun.c waarin een buffer overrun plaatsvindt.

Buffer overrun kun je eenvoudig voorkomen door in de functies *scanf* en *fscanf* altijd een maximaal in te lezen karakters bij het format %s te gebruiken. In het bovenstaande programma moet de regel:

```c
    scanf("%s", buffer);
```

worden vervangen door:

[1] We gaan er hier wel vanuit dat de gebruiker alleen uitvoerrechten heeft voor het programma en geen leesrechten! Anders zou de gebruiker namelijk eenvoudig het geheime password kunnen opsporen door de bufferoverrun.exe file in notepad te laden. Als de gebruiker ook leesrechten heeft op de executable moeten we het password ëncryptedïn de exe file opslaan.

```
    scanf("%9s", buffer);
```

De functie zal nu hooguit 9 karakters inlezen (en de tiende plaats in de buffer vullen met het nul karakter).

fgets en andere functies

Er zijn nog meer functies gedeclareerd in *stdio.h* om informatie uit tekstfiles te kunnen lezen, bijvoorbeeld *fgets*. Deze functies worden hier niet verder behandeld maar kunnen in de cppreference worden opgezocht.

13.3.4 Tekstfile schrijven

Er zijn verschillende functies beschikbaar om in een tekstfile te schrijven. De eerste die we bekijken is *putc*.

putc

De eenvoudigste functie is *putc*. Deze functie schrijft 1 karakter in de file en is in de include file stdio.h als volgt gedeclareerd:

```
int putc(int c, FILE *stream);
```

Als eerste parameter moet het karakter worden meegegeven dat in de file moet worden weggeschreven. Als tweede parameter moet een *FILE** worden meegegeven die je hebt teruggekregen van de functie *fopen*.

Als het lukt om een karakter in de file te schrijven geeft *putc* het weggeschreven karakter terug. De functie geeft de speciale integer waarde *EOF* terug als er iets fout gaat.

Het programma toupper.c kopieert de file infile.txt naar de file outfile.txt waarbij alle kleine letters worden omgezet naar hoofdletters.

```
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c;
    FILE *infile, *outfile;
    infile = fopen("infile.txt", "r");
    if (infile == NULL) {
        printf("Kan file infile.txt niet openen!");
    }
    else {
        outfile = fopen("outfile.txt", "w");
        if (outfile == NULL) {
            printf("Kan file outfile.txt niet openen!");
        }
        else {
            c = getc(infile);
            while (c != EOF) {
```

```
                c = putc(toupper(c), outfile);
                if (c == EOF) {
                    printf("Fout tijdens schrijven in outfile.txt!");
                }
                else {
                    c = getc(infile);
                }
            }
            fclose(outfile);
        }
        fclose(infile);
    }
    getchar();
    return 0;
}
```

Listing 26: Kopiëren van een file waarbij alle kleine letters worden omgezet naar hoofdletters.

fprintf

In plaats van karakter voor karakter weg te schrijven kun je ook zogenaamde geformatteerde uitvoer doen. Dit houdt in dat variabelen van verschillende typen in een opgegeven formaat worden omgezet naar karakters die naar de file worden geschreven. Je bent al bekend met geformatteerde uitvoer naar het beeldscherm met de functie *printf*. Geformatteerde uitvoer naar een file gaat op vergelijkbare wijze met de functie *fprintf*. Deze functie is in de include file *stdio.h* als volgt gedeclareerd:

```
int fprintf(FILE *stream, const char *format, ...);
```

Als eerste parameter moet een *FILE** worden meegegeven die je hebt teruggekregen van de functie *fopen*. De overige parameters zijn hetzelfde als bij de al bekende functie *printf*.

De functie geeft de integer waarde *EOF* terug als er iets fout gaat bij het wegschrijven. Anders geeft de functie het aantal karakters terug dat succesvol is weggeschreven.

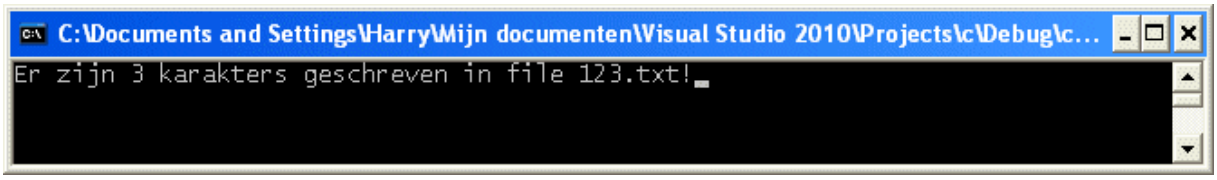
In het volgende voorbeeldprogramma printf_123.c wordt aan de integer variabele *i* de waarde 123 toegekend en daarna wordt deze variabele weggeschreven in een tekstfile. (Dit is natuurlijk volkomen zinloos maar maakt wel duidelijk hoe het werkt.)

```
#include <stdio.h>

int main(void) {
    int i = 123;
    FILE *fp;
    int ret;
    fp = fopen("123.txt", "w");
    if (fp == NULL) {
        printf("Kan file 123.txt niet openen!");
    }
    else {
        ret = fprintf(fp, "%d", i);
```

```
        if (ret == EOF) {
            printf("Er is iets fout gegaan bij schrijven in file ↩
             ↪ 123.txt!");
        }
        else {
            printf("Er zijn %d karakters geschreven in file ↩
             ↪ 123.txt!", ret);
        }
        fclose(fp);
    }
    getchar();
    return 0;
}
```

Listing 27: Wegschrijven van een int in een file.

De uitvoer van dit programma is als volgt:



Figuur 10: Uitvoer van printf_123.c.

In de directory waar de executable file is uitgevoerd is nu de file 123.txt aangemaakt. In de filebrowser kunnen de ”eigenschappen” van deze file bekeken worden. De file blijkt 3 bytes groot te zijn. Deze file kan met notepad worden bekeken:

Met een zogenaamde hexeditor kunnen we de inhoud van de file zowel in hexadecimale codering als in ASCII codering bekijken. Een eenvoudige hexeditor kun je downloaden vanaf: `http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm`.

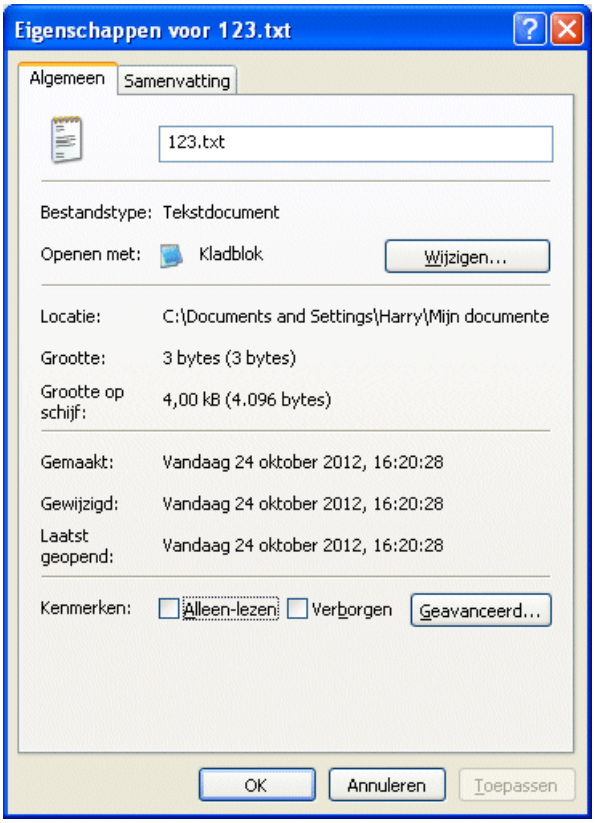
In het deel met de grijze achtergrond zien we de hex codes van de bytes uit de file. In het deel met de witte achtergrond zien we de bijbehorende ASCII karakters. Het is nu duidelijk te zien dat de file bestaat uit 3 bytes. Er is **geen** end-of-file karakter! De drie bytes bevatten de ASCII codes voor karakter 1, karakter 2 en karakter 3.

fputs en andere functies

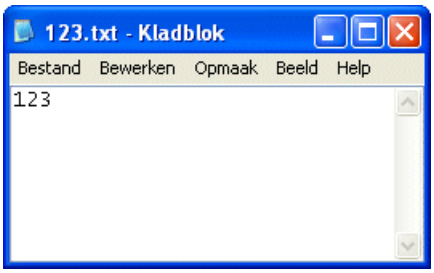
Er zijn nog meer functies gedeclareerd in *stdio.h* om informatie naar tekstfiles te kunnen schrijven, bijvoorbeeld *fputs*. Deze functies worden hier niet verder behandeld maar kunnen in de cppreference worden opgezocht.

13.3.5 Diverse andere functies voor tekstfiles

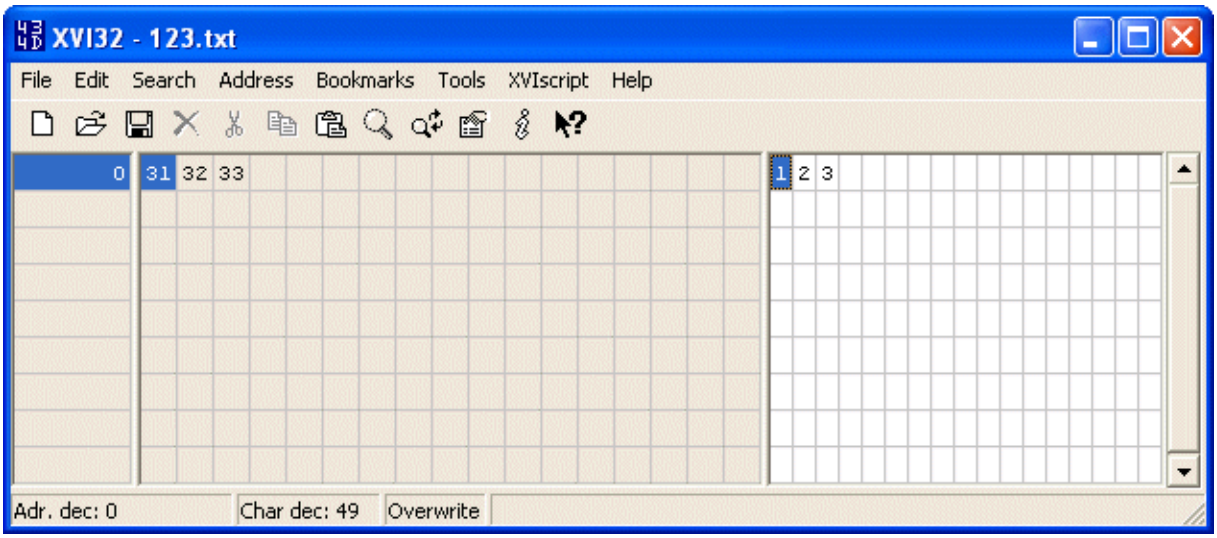
Er zijn nog diverse andere functies beschikbaar in *stdio.h* die voor tekstfiles gebruikt kunnen worden: feof, ferror, ffush, freopen, remove, rename, rewind, tmpfile enz. Deze functies worden hier niet verder behandeld maar kunnen via de voorgaande links in de cppreference worden opgezocht.



Figuur 11: Eigenschappen voor 123.txt



Figuur 12: Inhoud van 123.txt getoond m.b.v. notepad



Figuur 13: Inhoud van 123.txt getoond m.b.v. een hex-editor

14 Les 12

In deze les worden twee zaken behandeld: Het ophalen van de tijd op een pc en het communiceren via RS232 op de pc.

14.1 Leerstof

Paragraaf 14.2 op pagina 43 en de pdf die te vinden is op `http://bd.eduweb.hhs.nl/micprg/pdf/serial-win.pdf` dient te worden bestudeerd. Op de pagina `http://bd.eduweb.hhs.nl/micprg/serieel.htm` zie je een voorbeeldprogramma van hoe je kan communiceren via RSS232 op de pc. Deze dien je ook te bestuderen.

14.2 Gebruik van de standaard C include file <time.h>

De include file time.h is opgenomen in de C ANSI standaard. Deze headerfile bevat een aantal types en functies om te werken met het begrip tijd. Dit zijn de belangrijkste types die in time.h gedefinieerd zijn:

- *time_t* (een grote integer)
- *struct tm* (een struct)

14.2.1 time_t

Een tijdstip inclusief de datum kan in standaard C worden opgeslagen als een groot geheel getal. Dit getal geeft het aantal seconden aan dat verlopen is sinds 00:00:00 GMT, January 1, 1970. Om tijdstippen in deze vorm te kunnen gebruiken kun je een variabele genaamd *tijd* als volgt definiëren:

```
time_t tijd;
```

De variabele tijd kan nu bewerkt worden met de in tabel 3 gegeven functies:

Tabel 3: Functies waarmee een variabele van het type `time_t` bewerkt kan worden.

Functie	Beschrijving
time(&tijd)	Haalt de huidige tijd en datum uit het operating systeem. Als de tijd en datum niet beschikbaar zijn wordt *tijd* gelijk aan -1.
ctime(&tijd)	Converteert de variabele *tijd* naar een string. Deze functie geeft een pointer naar de string terug.
localtime(&tijd)	Converteert de variabele *tijd* naar een voorgedefineerde structuur *tm*. Deze functie geeft een pointer naar een *struct tm* terug.

14.2.2 struct tm

In standaard C is een structuur gedefinieerd waarin de tijd kan worden opgeslagen. Dit is de *struct tm*. Om de struct te gebruiken wordt hieronder een variabele *t* gedeclareerd met:

```
struct tm t;
```

De *struct tm* heeft de in tabel 4 gegeven veldnamen.

Tabel 4: Velden het type `struct tm`.

Veldnaam	Beschrijving
tm_sec	Seconden
tm_min	Minuten
tm_hour	Uren
tm_mday	Dag van de maand (1..31)
tm_mon	Maand (0=januari)
tm_year	Jaar - 1900 (Dus 110 = 2010)
tm_wday	Dag van de week (0=zondag)
tm_yday	Dag van het jaar (0..365)
tm_isdst	Zomertijd 0=Nee, >0=Ja, <0=Onbekend

De huidige tijd kan als volgt naar een *struct tm* worden omgezet:

```
time_t tijd;
struct tm* pt;
time(&tijd);
pt = localtime(&tijd);
printf("De datum is: %02d-%02d-%4d.\n", pt->tm_mday, pt->tm_mon + ↩
 ↪ 1, pt->tm_year + 1900);
printf("De tijd is: %02d:%02d:%02d.\n", pt->tm_hour, pt->tm_min, ↩
 ↪ pt->tm_sec);
```

De *struct tm* kan bewerkt worden met de in tabel 5 gegeven functies.

Tabel 5: Functies waarmee de `struct tm` bewerkt kan worden.

Functie	Beschrijving
asctime(&t)	Converteert de structuur *tm* naar een string.
mktime(&t)	Conveerteert de structuur *tm* naar een variabele van het type *t_time*.

14.2.3 Voorbeeldprogramma's

Het programma demotijd.c haalt de huidige tijd en datum op en drukt die op verschillende manieren af:

```
#include <stdio.h>
#include <time.h>
```

```
int main(void) {
    time_t tijd;
    struct tm* pt;
    /* De tijd en datum ophalen uit het operating systeem */
    time(&tijd);
    if (tijd != -1) {
        /* De tijd en datum afdrukken zonder het gebruik van de ↩
         ↪ structuur tm */
        printf("Vandaag is het volgens het operating systeem van ↩
         ↪ de computer:\n%s\n", ctime(&tijd));
        /* De tijd en datum omzetten naar de structuur tm*/
        pt = localtime(&tijd);
        /* De tijd en datum afdrukken met de structuur tm */
        printf("Vandaag is het dag %d van het jaar %d.\n", ↩
         ↪ pt->tm_yday + 1, pt->tm_year + 1900);
        printf("De datum is: %02d-%02d-%4d.\n", pt->tm_mday, ↩
         ↪ pt->tm_mon + 1, pt->tm_year + 1900);
        printf("De tijd is: %02d:%02d:%02d.\n", pt->tm_hour, ↩
         ↪ pt->tm_min, pt->tm_sec);
        if (pt->tm_isdst >= 0) {
            printf("Het is ");
            if (pt->tm_isdst == 0) {
                printf("wintertijd.\n");
            }
            else {
                printf("zomertijd.\n");
            }
        }
    }
    else {
        printf("Datum en tijd zijn niet beschikbaar op dit ↩
         ↪ systeem.\n");
    }
    getchar();
    return 0;
}
```

Listing 28: Voorbeeldprogramma dat de datum en tijd afdrukt.

Uitvoer:

Het programma welkedag.c leest een datum in en drukt dan af welke dag het op die datum is.

```
#include <stdio.h>
#include <time.h>

int main(void) {
    struct tm t, *pt;
```

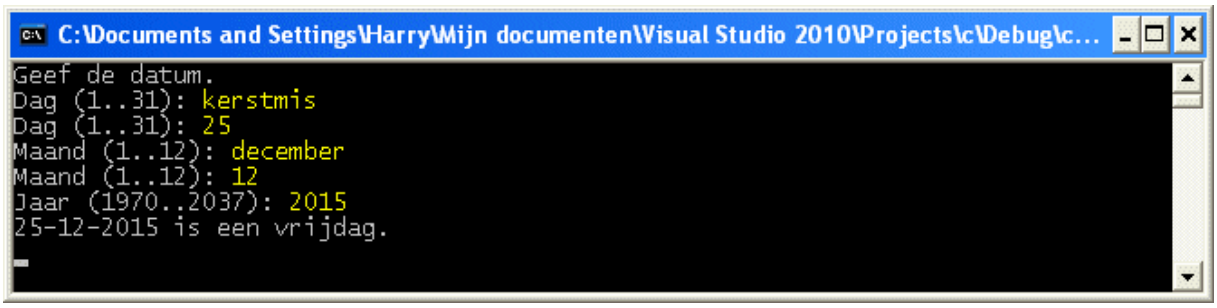
Figuur 14: Uitvoer van demotijd.c.

```
time_t tijd;
char* dag[] = {"zondag", "maandag", "dinsdag", "woensdag", ↩
 ↪ "donderdag", "vrijdag", "zaterdag"};

/* stuct tm t vullen */
printf("Geef de datum.\n");
do {
    printf("Dag (1..31): ");
    fflush(stdin);
}
while (scanf("%d", &t.tm_mday) != 1 || t.tm_mday < 1 || ↩
 ↪ t.tm_mday > 31);
do {
    printf("Maand (1..12): ");
    fflush(stdin);
}
while (scanf("%d", &t.tm_mon) != 1 || t.tm_mon < 1 || t.tm_mon ↩
 ↪ > 12);
t.tm_mon -= 1;
do {
    printf("Jaar (1970..2037): ");
    fflush(stdin);
}
while (scanf("%d", &t.tm_year) != 1 || t.tm_year < 1970 || ↩
 ↪ t.tm_year > 2037);
t.tm_year -= 1900;

t.tm_sec = 0;
t.tm_min = 0;
t.tm_hour = 0;
t.tm_isdst = -1;

/* struct tm omzetten naar een time_t */
tijd = mktime(&t);

if (tijd == -1) {
    printf("Ongeldige datum!\n");
}
```

```
    else {
        /* time_t weer terug omzetten naar een struct tm */
        pt = localtime(&tijd);
        /* De dag van de week afdrukken */
        printf("%02d-%02d-%4d is een %s.\n", pt->tm_mday, ↩
         ↪ pt->tm_mon + 1, pt->tm_year + 1900, dag[pt->tm_wday]);
    }
    fflush(stdin);
    getchar();
    return 0;
}
```

Listing 29: Welke dag is het.

Uitvoer:



Figuur 15: Uitvoer van welkedag.c. De ingetypte invoer is geel weergegeven.

14.2.4 Details

Details over de in de include file *time.h* gedeclareerde types en functies kun je vinden op: `http://en.cppreference.com/w/c/chrono`.

15 Les 13

Centraal in deze les staan de verschillende geheugens die de ATmega32 heeft en ook hoe deze te gebruiken zijn vanuit software.

15.1 Leerstof

- 8.6 uit [MNN13] t/m Table 7.
- PDF die te vinden is op: `http://www.microdigitaled.com/AVR/Articles/UsingFlashDataSpaceToStoreData_v1.pdf`.
- 9.3 uit [MNN13].
- 9.4 uit [MNN13].
- Paragraaf 4.17 uit [KP09].

15.2 Opmerkingen over leerstof

In de slides zijn functies te zien waarmee je kan schrijven en lezen in het EEPROM geheugen. Deze zijn niet terug te vinden in de bovenstaande leerstof. Het gebruik van de EEPROM functies is echter vergelijkbaar met die van de functies voor het benaderen van het flash geheugen, dus het is niet echt nodig om hier apart nog iets voor te bestuderen naast de slides.

9.4 uit [MNN13]: Waarschijnlijk zal het je al opgevallen zijn dat in het practicum bij het STK500 bordje de tweede manier (ISP) wordt gebruikt en dan specifiek de JTAG interface.

16 Les 14

In deze les komen de verschillende power save modes aan bod die kunnen worden gebruikt op de ATmega32 en de ATmega32A.

16.1 Leerstof

- Paragraaf 9.1 t/m 9.7 uit [Cor] (datasheet ATmega32A)
- Paragraaf 9.9 uit [Cor]
- Paragraaf 17.11.4 uit [Cor]

16.2 Opmerking over 9.9

De bitjes uit het MCUCR register die je ziet bij paragraaf 9.9 hoef je zelf niet direct in te stellen als je C gebruikt. De bitjes worden namelijk geset door de functie *set_sleep_mode* die te zien is in de slides die horen bij deze les.

Referenties

[Cor] Atmel Corporation. *ATmega32A[DATASHEET]*. http://www.eduweb.hhs.nl/ jzbroeders/micprg/pdf/ATmega32A.pdf (zie pagina 48).

[KP09] Al Kelley en Ira Pohl. *De programmeertaal C*. 4e vernieuwde editie. Pearson, 2009. ISBN: 9789043016698 (zie pagina's 22, 32, 47).

[MNN13] Muhammad Ali Mazidi, Sarmad Naimi en Sepehr Naimi. *AVR Microcontroller and Embedded Systems: Using Assembly and C*. International Edition. Pearson, 2013. ISBN: 978-1-292-04256-5 (zie pagina's 4, 5, 6, 19, 20, 23, 28, 29, 30, 47, 48).