21

EEPROM en seriële communicatie

Dit hoofdstuk leert je hoe het EEPROM van de ATmega32 kunt benaderen en leer je hoe je serieel met een extern EEPROM en met andere componenten kunt communiceren.

De behandelde onderwerpen zijn:

- Het gebruik bij de ATmega32 van het interne EEPROM, *Electrical Erasable Read-Only Memory*.
- De SPI, *Serial Peripheral Interface*.
- Het benaderen van een extern EEPROM via de SPI.
- De I^2C-interface of TWI-interface, *Two-Wire serial Interface*.
- Het benaderen van een *real time clock* via I^2C.

De voorbeelden tonen:

- Het schrijven naar en het lezen uit het interne EEPROM.
- Het declareren van variabelen in het EEPROM.
- Het initialiseren van het EEPROM met een `eep`-bestand.
- De overeenkomsten en de verschillen tussen het programmeren van het flash en het EEPROM.
- Het schrijven naar en het lezen uit een extern EEPROM via de SPI.
- De beschrijving van een bibliotheek voor het lezen en schrijven van I^2C.
- Het schrijven naar en het lezen uit de *real time clock DS1307* via I^2C.

Een microcontroller gebruikt het RAM voor het bewaren van gegevens en wordt daarom datageheugen genoemd. De compiler reserveert in het datageheugen ruimte voor variabelen en gebruikt een deel van dit geheugen als stack. Het nadeel van het RAM is dat de gegevens verloren gaan als de spanning wegvalt. Voor situaties dat dit niet gewenst is, heeft een microcontroller EEPROM. EEPROM houdt, als de spanning wegvalt, de gegevens wel vast.

Naast het interne EEPROM kan de ontwerper besluiten een extern EEPROM toe te passen. Seriële communicatie is zeer geschikt voor de communicatie met een extern EEPROM en andere componenten. Tegenwoordig zijn er zeer veel componenten beschikbaar met een SPI- of een I^2C-interface. Een modern ontwerp van een PCB bevat vaak een I^2C-bus om efficiënt de diverse componenten te kunnen benaderen.
Dit hoofdstuk bespreekt eerst de communicatie met het interne EEPROM, dan de SPI met als voorbeeld het benaderen van een extern EEPROM en bespreekt tenslotte I^2C met als voorbeeld de interfacing met een DS1307.

21.1 EEPROM van de ATmega32

De meeste microcontrollers hebben drie soorten geheugens: flash voor het programma, RAM voor vluchtige gegevens en EEPROM voor de opslag van nietvluchtige gegevens. Het aantal keer dat er naar een EEPROM geschreven wordt, is beperkt. Het EEPROM van de ATmega32 mag maximaal 100.000 keer beschreven worden. Voor een ontwerp dat tien jaar moet functioneren, betekent dit dat er hooguit een keer per uur naar het EEPROM geschreven mag worden. Anders gezegd als er elke seconde gegevens naar het EEPROM geschreven worden, is de gegarandeerde maximale levensduur 27,7 uur.

De gegevens worden per byte in het EEPROM gezet. De beperking van 100.000 keer schrijven, kan aanzienlijk worden verbeterd door het hele EEPROM te gebruiken. Dit kan door de gegevens in een circulaire buffer te zetten. Afhankelijk van de omvang van de gegevens kan dit de levensduur ruwweg met een factor 100 verlengen.

Het grote verschil met het RAM-geheugen is dat het EEPROM niet direct toegankelijk is. Om waarden uit het EEPROM te halen of er naar toe te schrijven zijn speciale registers nodig. Het is dus niet mogelijk om direct iets naar het EEPROM te schrijven of er uit te halen. In figuur 1.9 en in figuur 10.2 zijn er dan ook geen adreslijnen naar het EEPROM getekend; alle gegevens gaan via de databus. De oplossing om het EEPROM te benaderen, zal bij elke C-compiler anders zijn. De *avr-libc*-bibliotheek heeft een aantal voorgedefinieerde functies om het EEPROM te benaderen. Een overzicht van deze functies staat in tabel 21.1 en zijn gedefinieerd in het headerbestand `eeprom.h`.

Tabel 21.1 : De lees- en schrijffuncties uit eeprom.h.

functie	omschrijving
`eeprom_read_byte`	`uint8_t eeprom_read_byte (const uint8_t *addr)` Deze functie leest een byte van het EEPROM-adres `addr`.
`eeprom_read_word`	`uint8_t eeprom_read_word (const uint16_t *addr)` Deze functie leest een *word* (twee bytes) van het EEPROM-adres `addr`.
`eeprom_read_block`	`void eeprom_read_block (void *ram_addr, const void *eeprom_addr, size_t n)` Deze functie leest een blok van `n` bytes van het EEPROM-adres `eeprom_addr` en plaatst deze in het RAM op adres `ram_addr`.
`eeprom_write_byte`	`void eeprom_write_byte (uint8_t *addr, uint8_t value)` Deze functie schrijft een byte `value` naar het EEPROM-adres `addr`.
`eeprom_write_word`	`void eeprom_write_word (uint16_t *addr, uint16_t value)` Deze functie schrijft een *word* `value` naar het EEPROM-adres `addr`.
`eeprom_write_block`	`void eeprom_write_block (const void *ram_addr, void *eeprom_addr, size_t n)` Deze functie schrijft een blok van `n` bytes van het adres `ram_addr` uit het RAM naar het EEPROM-adres `eeprom_addr`.

Code 21.1 plaatst vanaf adres `0x00` in het totaal tien bytes in het geheugen en leest iedere halve seconde een van deze waarden en zet deze op de uitgangen van poort B. De tien bytes vormen de tafel van veertien.

Het sleutelwoord **volatile** op regel 7 is overbodig. Het is toegevoegd voor het debuggen. De optimalisatie -Os is zo vergaand dat de variabele i niet meer in de assemblercode voorkomt. Zonder **volatile** is i niet zichtbaar in AVRstudio. De versie met **volatile** is groter — 254 tegenover 198 bytes — en iets trager.

Code 21.1 : Het schrijven naar en lezen uit het EEPROM.

```
1  #include <avr/io.h>
2  #include <avr/eeprom.h>
3  #include <util/delay.h>
4
5  int main(void)
6  {
7    volatile int  i;
8
9    DDRB = 0xFF;
10
11   for (i=0; i<10; i++) {
12     eeprom_write_byte((uint8_t *)i,14*(i+1));
13   }
14
15   while (1) {
16     if (i==10) {
17       i=0;
18     }
19     PORTB = eeprom_read_byte((uint8_t *)i);
20     _delay_ms(500);
21     i++;
22   }
23 }
```

Op regel 2 is het headerbestand eeprom.h ingesloten. Op regel 12 schrijft de functie eeprom_write_byte de bytes naar het EEPROM. De eerste variabele van de functie is het adres van de locatie in het EEPROM en de tweede is de waarde van de byte. In dit geval is het adres de lusvariabele i van de **for**-lus en de waarde is 14*(i+1). De lusvariabele is een integer en wordt getypecast naar een pointer die naar een byte, oftewel een acht-bits unsigned integer, wijst.
De functie eeprom_read_byte op regel 19 leest de byte van adres i. De typecasting is overbodig als in plaats van een integer i een pointervariabele p van het type uint_8* gebruikt zou zijn.

De functies uit eeprom.h zijn geoptimaliseerd en grondig getest. Gebruik daarom altijd de functies uit eeprom.h. De datasheet van de ATmega32 geeft voor het schrijven en het lezen van een byte een implementatie in C. In code 21.2 en 21.3 staan twee functies eeprom_write_byte en eeprom_read_byte die hierop geïnspireerd zijn. Hoewel het beter is de functies uit eeprom.h te gebruiken, geven deze functies inzicht in de wijze waarop een EEPROM toegepast wordt.

Code 21.2 : De functie eeprom_write_byte.

```
1  void eeprom_write_byte(const uint8_t *a, uint8_t d)
2  {
3    while(EECR & _BV(EEWE));
4    EEAR  = (uint16_t) a;
5    EEDR  = d;
6    EECR |= _BV(EEMWE);
7    EECR |= _BV(EEWE);
8  }
```

Code 21.3 : De functie eeprom_read_byte.

```
1  uint8_t eeprom_read_byte(uint8_t *a)
2  {
3    while(EECR  & _BV(EEWE));
4    EEAR = (uint16_t) a;
5    EECR |= _BV(EERE);
6    return EEDR;
7  }
8
```

Voor het benaderen van het EEPROM zijn drie registers nodig: het `EEAR`-, het `EEDR`-, en het `EECR`-register. Het 16-bits `EEAR`-register bevat het adres van de locatie binnen het EEPROM dat wordt benaderd. In het `EEDR`-register staat de databyte die naar deze locatie wordt geschreven of van deze locatie wordt gelezen. Vier bits uit het `EECR`-register regelen de acties voor het lezen en het schrijven.

De functies `eeprom_write_byte` en `eeprom_read_byte` wachten allebei totdat het `EEWE`-bit laag is en zetten eerst de gewenste locatie uit het EEPROM in het `EEAR`-register. De functie `eeprom_write_byte` plaatst daarna de te schrijven byte `d` in het `EEDR`-register. Vervolgens wordt eerst het `EEMWE`-bit hoog gemaakt en tenslotte wordt het `EEWE`-bit hoog gemaakt. De functie `eeprom_read_byte` maakt nadat de locatie in het adres is gezet, het `EERE`-bit hoog en geeft de waarde van het `EEDR`-register terug.

Een schrijfactie naar het EEPROM duurt lang. De datasheet geeft 8,5 ms als karakteristieke waarde.
Tussen het hoog maken van het `EEMWE`-bit en het `EEWE`-bit mag geen interrupt zijn.
De functie `eeprom_write_byte` uit code 21.2 kan daardoor falen.
De schrijffuncties uit `eeprom.h` handelen dit wel correct af.

Code 21.4 : Het schrijven naar en lezen uit een array in het EEPROM.

```
1  #include <avr/io.h>
2  #include <avr/eeprom.h>
3  #include <util/delay.h>
4
5  uint8_t EEMEM v[10];
6
7  int main(void)
8  {
9    int  i;
10
11   DDRB = 0xFF;
12
13   for (i=0; i<10; i++) {
14     eeprom_write_byte(&v[i],14*(i+1));
15   }
16   while (1) {
17     if (i==10) {
18       i=0;
19     }
20     PORTB = eeprom_read_byte(&v[i]);
21     _delay_ms(500);
22     i++;
23   }
24 }
```

Een nadeel van de oplossing van code 21.1 is dat er expliciete adressen worden gebruikt. Vooral bij grote ontwerpen kan het lastig worden om steeds het juiste adres te gebruiken. In plaats daarvan kunnen variabelen in het EEPROM gedeclareerd worden. Het attribuut `EEMEM` bij de declaratie van een variabele geeft aan dat de variabele in het EEPROM staat.

In code 21.4 is op regel 5 een array `v` met dit attribuut gedeclareerd. Deze code is functioneel identiek met code 21.1. Alleen bepaalt de compiler nu de plaats waar `v` in het EEPROM komt te staan. De adreslocatie die op regel 14 en 20 aan de functies `eeprom_write_byte` en `eeprom_read_byte` wordt meegegeven, is het adres van variabele `v[i]`.

De compiler vertaalt code 21.4 naar twee bestanden: een `hex`-bestand met de assemblercode van het programma en een `eep`-bestand met de gegevens waarmee

het EEPROM geprogrammeerd kan worden. Omdat bij de declaratie van variabele v geen beginwaarden staan, worden de tien bytes ingevuld met nullen. Het eep-bestand gebruikt hetzelfde Intel Hex-formaat als het hex-bestand en bevat deze tekst:

```
:0A00000000000000000000000000000000F6
:00000001FF
```

De vetgedrukte nullen zijn de tien bytes. Figuur 21.1 laat de eerste bytes van het EEPROM zien direct nadat het eep-bestand in AVRstudio is geladen.

```
Memory
EEPROM   8/16  abc.  Address: 0x00   Cols: 16
000000 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Figuur 21.1 : Het EEPROM na het laden.

```
Memory
EEPROM   8/16  abc.  Address: 0x00   Cols: 16
000000 0E 1C 2A 38 46 54 62 70 7E 8C FF FF FF FF FF FF
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Figuur 21.2 : Het EEPROM nadat het gevuld is.

Nadat het programma de **for**-lus van regel 13 heeft doorlopen zijn deze nullen overschreven met de hexadecimale waarden van de tafel van veertien, zoals figuur 21.2 laat zien.
Overigens hoeft in dit voorbeeld het eep-bestand niet geladen te worden, omdat de gegevens uit dit bestand niet worden gebruikt en direct worden overschreven met de tafel van veertien.

Code 21.5 : Het initialiseren van en het lezen uit het EEPROM.

```
1  #include <avr/io.h>
2  #include <avr/eeprom.h>
3  #include <util/delay.h>
4
5  uint8_t EEMEM v[10] = {
6      0x0E, 0x1C, 0x2A, 0x38, 0x46,
7      0x54, 0x62, 0x70, 0x7E, 0x8C };
8
9  int main(void)
10 {
11   int i=0;
12
13   DDRB = 0xFF;
14
15   while (1) {
16     if (i==10) {
17       i=0;
18     }
19     PORTB = eeprom_read_byte(&v[i]);
20     _delay_ms(500);
21     i++;
22   }
23 }
```

In plaats van een **for**-lus, die het EEPROM initialiseert, kan er een eep-bestand gemaakt worden met gegevens. In code 21.5 worden bij de declaratie van array v tegelijkertijd de gegevens toegekend. Na compilatie is er naast het hex-bestand

met het programma ook een eep-bestand met deze gegevens:

```
:0A0000000E1C2A38465462707E8CF4
:00000001FF
```

Nadat de beide bestanden in de microcontroller geprogrammeerd zijn en het programma loopt, verschijnt er elke halve seconde een andere waarde op poort B.

De voorbeelden uit deze paragraaf zijn alleen bedoeld als demonstratie van het schrijven naar en het lezen uit het EEPROM. Het is over het algemeen niet praktisch om gegevens, die niet veranderen in het EEPROM te plaatsen. Het is beter om hiervoor het flash-geheugen te gebruiken. Code 11.6 uit paragraaf 11.9 geeft een voorbeeld van het gebruik van het flashgeheugen.

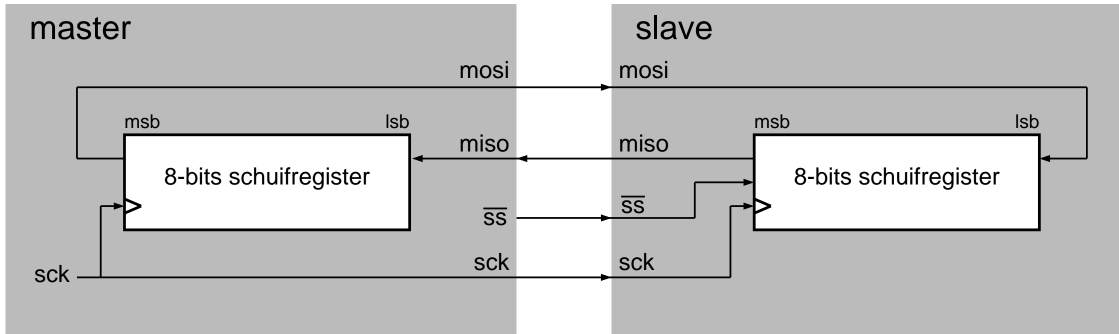
Code 21.6 : Het initialiseren van en het lezen uit het flashgeheugen.

```
1  #include <avr/io.h>
2  #include <avr/pgmspace.h>
3  #include <util/delay.h>
4
5  uint8_t PROGMEM v[10] = {
6      0x0E, 0x1C, 0x2A, 0x38, 0x46,
7      0x54, 0x62, 0x70, 0x7E, 0x8C };
8
9  int main(void)
10 {
11   int i=0;
12
13   DDRB = 0xFF;
14
15   while (1) {
16     if (i==10) {
17       i=0;
18     }
19     PORTB = pgm_read_byte(&v[i]);
20     _delay_ms(500);
21     i++;
22   }
23 }
```

Code 21.6 bevat het voorbeeld van code 21.5, maar nu met de gegevens in het flashgeheugen in plaats van in het EEPROM. Op regel 2 is `eeprom.h` vervangen door het headerbestand `pgmspace.h`. Op regel 5 is het attribuut `EEMEM` bij de declaratie van variabele `v` veranderd in `PROGMEM` . Voor het lezen van de bytes wordt op regel 19 in dit geval de functie `pgm_read_byte` gebruikt in plaats van de functie `eeprom_read_byte`.

21.2 SPI

De SPI (Serial Peripheral Interface) is een vierdraads seriële verbinding. Net als de TWI of I^{2}C-interface is deze aansluiting bedoeld om met andere componenten op het printed circuit board te communiceren. De SPI bestaat uit een kloklijn, een selectielijn en twee lijnen voor het versturen en het ontvangen van gegevens.



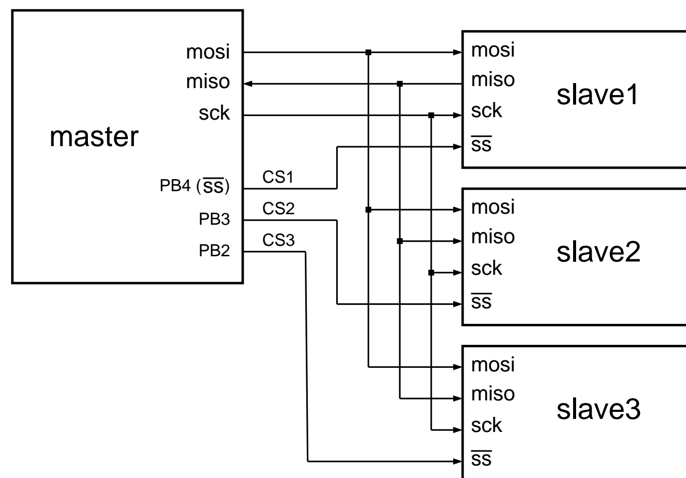
Figuur 21.3 : De SPI met de master en een slave. De `MOSI` van de master is aangesloten aan de `MOSI` van de slave. De `MISO` van de slave is verbonden met de `MISO` van de master. De schuifregisters van de master en de slave vormen samen een roterend schuifregister. De master genereert de klok voor de slave.

De SPI kent een *master mode* en een *slave mode*. In figuur 21.3 staan de verbindingen tussen de *master*, meester, en de *slave*, slaaf. In de *slave mode* is de selectielijn $\overline{\texttt{SS}}$ als ingang geconfigureerd. De kloklijn `SCK` is een uitgang van de master en een ingang van de slave. De master gebruikt dit signaal om waarden in en uit het schuifregister te schuiven. De slave schuift de gegevens als er een kloksignaal is en als bovendien $\overline{\texttt{SS}}$ laag is.

De in- en uitgangen van het schuifregister worden soms ook anders genoemd; bijvoorbeeld `SDI` voor de ingang en `SDO` voor de uitgang. De selectielijn wordt bij sommige bouwstenen aangeduid met `CS` (*chip select*).

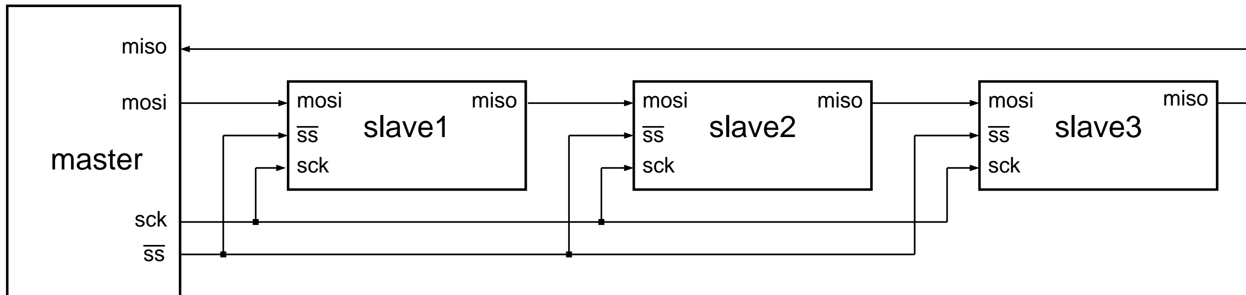
De `MISO`, *Master in Slave Out* van de master is verbonden met de `MISO` van de slave en de `MOSI`, *Master Out Slave In* van de master is verbonden met de `MOSI` van de slave. Hierdoor ontstaat een groot, roterend schuifregister. Na acht klokslagen staat de waarde van de master in het schuifregister van de slave en staat de waarde van de slave in het schuifregister van de master.

Er bestaan ook bouwstenen met een SPI die drie aansluitingen heeft. De temperatuursensor LM74 heeft bijvoorbeeld een gemeenschappelijke `MISO`/`MOSI`. Deze sensor stuurt alleen gegevens. Zodra $\overline{\texttt{SS}}$ laag wordt, schuift de LM74 gegevens naar buiten.



Figuur 21.4 : Een SPI-configuratie met een master en drie slaves. De `MOSI`, `MISO` en `SCK` van de master zijn verbonden met de drie slaves. Er zijn drie selectiesignalen `CS1`, `CS2` en `CS3`. De master selecteert daarmee de slave waarmee gecommuniceerd wordt.

Bij een microcontroller heeft de SPI meestal vier lijnen. In de *master mode* is de selectielijn een gewone uitgang. Elke andere aansluitpin kan eveneens als selectielijn worden gebruikt. In figuur 21.4 staat een master met drie slaves. De `MOSI`, `MISO` en `SCK` van de master zijn verbonden met de slaves. Er is een aparte selectielijn voor iedere slave. Omdat de SPI van de ATmega32 één $\overline{\text{SS}}$ heeft, zijn er nog twee andere pinnen nodig, bijvoorbeeld aansluiting 2 en 3 van poort B.



Figuur 21.5 : De master en drie slaves zijn in een lange seriële ketting geplaatst.

In figuur 21.5 staat een alternatieve methode om de slaves op de microcontroller aan te sluiten. Door de slaves serieel met elkaar te verbinden, ontstaat er een lange ketting van schuifregisters. Er is nu maar een selectielijn nodig. Deze methode wordt gebruikt om het aantal aansluitingen van de microcontroller uit te breiden. Een ketting met drie 74HC595 schuifregisters voegt 24 aansluitingen toe aan de microcontroller.

Omdat nergens exact is vastgelegd aan welke eisen de SPI moet voldoen zijn er vier basisconfiguraties. Tabel 21.2 geeft deze vier modi. De klok kan als er geen klokpulsen zijn, laag of hoog zijn. Bovendien kan er worden geklokt op de opgaande en neergaande klokflank. In modus `0` en in modus 3 worden de gegevens geklokt bij de opgaande klokflank. Er zijn twee bits `CPOL` en `CPHA` nodig om de juiste modus van de SPI in te stellen. Een master en een slave moeten allebei dezelfde modus hebben om correct te functioneren.

Tabel 21.2 : De vier modi voor de SPI.

SPI-modus	CPOL	CPHA	betekenis
0	0	0	clock low, sample at leading edge
1	0	1	clock low, sample at trailing edge
2	1	0	clock high, sample at leading edge
3	1	1	clock high, sample at trailing edge

Tabel 21.3 : De instructies van de AT25128.

instructie	code	betekenis
WREN	6	set write enable latch
WRDI	4	reset write enable latch
RDSR	5	read status register
WRSR	1	write status register
READ	3	read data from memory array
WRITE	2	write data to memory array

De SPI van de ATmega32 gebruikt drie registers: een dataregister `SPDR`, een controlregister `SPCR` en een statusregister `SPSR`. Voor het versturen van een byte hoeft de byte alleen maar in het dataregister gezet te worden. De SPI schuift dan automatisch de bits van de byte naar buiten en maakt de `SPIF`-flag uit het statusregister hoog als alle acht bits verwerkt zijn.

Code 21.7 : Het headerbestand `spi_eeprom.h` voor het communiceren met een extern EEPROM via de SPI.

```
 1  #include <stddef.h>  // contains definition of size_t
 2
 3  // defines for ATmega32 SPI
 4  #define SPI_PORT    PORTB
 5  #define SPI_DDR     DDRB
 6  #define SPI_SS      PB4
 7  #define SPI_MOSI    PB5
 8  #define SPI_MISO    PB6
 9  #define SPI_SCK     PB7
10
11  // instruction codes for AT25xxx EEPROM with SPI
12  #define WREN        6
13  #define WRDI        4
14  #define RDSR        5
15  #define WRSR        1
16  #define READ        3
17  #define WRITE       2
18
19  // dummy value
20  #define FOO         0
21
22  // function prototypes
23  void    spi_init(void);
24  uint8_t spi_transfer(uint8_t data);
25  uint8_t spi_eeprom_read_byte(uint16_t addr);
26  uint8_t spi_eeprom_read_block(uint8_t *dst, uint16_t addr, size_t n);
27  void    spi_eeprom_write_byte(uint16_t addr, uint8_t data);
28  void    spi_eeprom_write_block(uint8_t *src, uint16_t addr, size_t n);
```

Componenten, die met een SPI leverbaar zijn, zijn bijvoorbeeld: EEPROM, flash, DAC, ADC, temperatuursensor, *real time clock*, LCD en schuifregisters. Deze componenten kunnen meestal alleen als slave functioneren en werken in een bepaalde SPI-modus en met maximale klokfrequentie. Deze paragraaf gebruikt als voorbeeld voor de communicatie met de SPI het schrijven en uitlezen van de AT25128. Dat is een 128 kbytes EEPROM van Atmel met een SPI. De SPI-modus van deze component is `0` en de maximale klokfrequentie is 2 MHz.

De EEPROM is als slave op de microcontroller aangesloten, zoals in figuur 21.3 getekend is. In code 21.7 staat een headerbestand `spi_eeprom.h` met een aantal definities. De SPI-poort zit voor de verschillende typen ATmega's bij andere aansluitingen. Op regel 3 staan de definities voor de SPI van de ATmega32. De AT25128 kent zes instructies voor het schrijven en lezen. De definities van deze instructies of opcodes staan vanaf regel 11 in het headerbestand. Tabel 21.3 geeft een korte omschrijving. De definitie `FOO` van regel 20 is een dummy waarde, die bij het lezen van gegevens wordt gebruikt.

Code 21.8 : Het begin van `spi_eeprom.c` met de functies voor het communiceren met een extern EEPROM via de SPI.

```
1  #include <avr/io.h>
2  #include "spi_eeprom.h"
3
4
5  void spi_init(void) {
6    SPI_PORT |= _BV(SPI_SS);   // set output SS high
7    SPI_DDR  |= _BV(SPI_SS)|   // make SS, MOSI, SCK output
8                _BV(SPI_MOSI)|
9                _BV(SPI_SCK);
10   SPCR      =                // SPIEN,    SPI interrupt ENable : disable
11               _BV(SPE)|      // SPI,      SPI Enable           : enable
12                              // DORD,     Data ORDer           : MSB 1st
13               _BV(MSTR);     // MSTR,     MaSTeR/slave select  : master
14                              // CPOL,     Clock POLarity       : 0, low
15                              // CPHA,     Clock PHAse          : 0, lead
16                              // SPR1,SPR0 SPi clock Rate       : FCPU/4
17 }
18
19
20 uint8_t spi_transfer(uint8_t data)
21 {
22   SPDR = data;
23   while ( !(SPSR & (_BV(SPIF))) ) ;
24
25   return SPDR;
26 }
```

In code 21.8 staat het begin van het bestand `spi_eeprom.c` met de routines voor het lezen en schrijven van gegevens naar het EEPROM via de SPI. De functie `spi_init` initialiseert de SPI. Uitgang $\overline{\texttt{SS}}$ wordt hoog gemaakt. Het EEPROM is dan niet geselecteerd als de SPI wordt ingesteld. De aansluitingen `SS`, `MOSI`, `SCK` zijn gedefinieerd als uitgang. Tenslotte configureert op regel 10 de functie `spi_init` de bits uit het controlregister `SPCR`. De SPI-modus is 0, omdat `CPOL` en `CPHA` beiden laag zijn. De frequentie van de klok `SCK` is 1/4 van de systeemklok. Het meest significante bit wordt eerst verstuurd. Er wordt geen interrupt gebruikt en de SPI wordt aangezet.

De AT25128 heeft een maximale klokfrequentie van 2 MHz. Bij de keuze van 1/4 voor de klokdeling is de maximale systeemklok van de microcontroller 8 MHz.

De functie `spi_transfer` schuift via de `MOSI`-pin de byte `data` naar buiten en geeft tegelijkertijd via de `MISO`-pin de naar binnen geschoven byte terug. Op regel 22 wordt `data` in het dataregister geplaatst. De SPI schuift dan automatisch de bits; na acht keer stopt het schuiven en wordt de `SPIF`-flag in het statusregister hoog. Op regel 23 wordt gewacht totdat deze statusvlag hoog is. De `SPIF`-flag wordt na het lezen automatisch laag gemaakt.

Het headerbestand van code 21.7 geeft de prototypes van de lees- en schrijffuncties. In code 21.9 staan twee functies om een byte te schrijven en te lezen. De schrijffunctie `spi_eeprom_write_byte` heeft twee ingangsparameters, namelijk: de te versturen byte `data` en het adres `addr` van de locatie waar deze byte in het EEPROM moet komen te staan. De functie start op regel 30 met het selecteren van

Code 21.9 : **Functies van `spi_eeprom.c` om een byte via de SPI te lezen en te schrijven uit een extern EEPROM.**

```
28  void spi_eeprom_write_byte(uint16_t addr, uint8_t data)
29  {
30    SPI_PORT &= ~(_BV(SPI_SS));    // select slave
31    spi_transfer(WREN);            // send Write Enable
32    spi_transfer(WRITE);           // send Write
33    spi_transfer(addr>>8);         // send MSB address
34    spi_transfer(addr);            // send LSB address
35    spi_transfer(data);            // send data
36    spi_transfer(WRDI);            // send Write Disable
37    SPI_PORT |= _BV(SPI_SS);       // deselect slave
38  }
39
40  uint8_t spi_eeprom_read_byte(uint16_t addr)
41  {
42    uint8_t data;
43
44    SPI_PORT &= ~(_BV(SPI_SS));    // select slave
45    spi_transfer(READ);            // send Read
46    spi_transfer(addr>>8);         // send MSB address
47    spi_transfer(addr);            // send LSB address
48    data = spi_transfer(FOO);      // get data
49    SPI_PORT |= _BV(SPI_SS);       // deselect slave
50
51    return data;
52  }
```

de slave en het versturen van instructie `WREN`. Daarna worden achtereenvolgens de instructie `WRITE`, het adres `addr` en de byte `data` verstuurd. Het schrijven wordt vanaf regel 36 afgesloten met het sturen van de instructie `WRDI` en het deselecteren van de slave.

De functie `spi_eeprom_read_byte` heeft alleen het adres `addr` als ingangsparameter. De truc om via de SPI een byte te lezen is om een willekeurige byte te versturen. Eerst wordt op regel 45 de instructie `READ` verstuurd en daarna wordt het adres en de willekeurige byte `FOO` verstuurd. Na het versturen staat de byte van het adres `addr` uit EEPROM in het dataregister en wordt deze door de `spi_transfer` teruggegeven en in de variabele `data` geplaatst. Op regel 51 geeft de functie `spi_eeprom_read_byte` deze waarde terug.
De functie `spi_eeprom_read_byte` lijkt sterk op `spi_eeprom_write_byte`; alleen de verstuurde opcodes en de verstuurde byte zijn anders en `spi_eeprom_read_byte` geeft de variabele `data` terug. Andere functies, zoals `spi_eeprom_write_block` en `spi_eeprom_read_block` die een blok van `n` bytes versturen en ontvangen, worden op een zelfde manier beschreven.

Code 21.1 gebruikt, net als code 21.4, het interne EEPROM van de ATmega32 om de tafel van veertien in op te slaan en uit te lezen. In code 21.10 staat een voorbeeld van de communicatie met een externe EEPROM via de SPI. Deze beschrijving gebruikt de functies uit `spi_eeprom.c` om via de SPI gegevens te versturen en te ontvangen. In plaats van het headerbestand `eeprom.h` wordt nu op regel 3

Code 21.10 : Het gebruik van een extern EEPROM via een SPI met de functies uit `spi_eeprom.c`.

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3  #include "spi_eeprom.h"
4
5  int main(void)
6  {
7    volatile unsigned int  i;
8
9    DDRD = 0xFF;
10   spi_init();
11
12   for (i=0; i<10; i++) {
13     spi_eeprom_write_byte(i+3, 14*(i+1));
14   }
15
16   while (1) {
17     if (i==10) {
18       i=0;
19     }
20     PORTD = spi_eeprom_read_byte(i);
21     _delay_ms(500);
22     i++;
23   }
24 }
```

`spi_eeprom.h` gebruikt. Omdat de SPI bij de ATmega32 op poort B zit, is poort D op regel 9 gedefineerd als uitgang. Op regel 10 is de initialisatie van SPI toegevoegd en op regel 13 en 20 staan nu de schrijf- en leesfuncties uit `spi_eeprom.c`.

Spreek I^2C in het Nederlands uit als i-kwadraat-c en in het Engels als *eye-squared-see*.

Philips Semiconductors is geen onderdeel meer van Philips. Philips Semiconductors is in 2006 zelfstandig verder gegaan onder de naam NXP.

21.3 I^2C

I^2C is — net als SPI — een serieel communicatieprotocol voor verbindingen tussen geïntegreerde schakelingen. Het protocol gebruikt slechts twee lijnen: één voor het versturen en ontvangen van gegevens en één voor het kloksignaal. I^2C is bedacht door Philips Semiconductors. Dit bedrijf voorzag al vroeg dat het gebruik van parallelle bussen op een printed circuit board bij steeds verdere integratie en hogere kloksnelheden grote design- en tijdsproblemen zouden geven. Philips toonde aan dat met een eenvoudige tweedraadsverbinding er toch snel gecommuniceerd kon worden tussen een groot aantal geïntegreerde schakelingen. Het bedrijf bracht zelf componenten op de markt met een I^2C-interface en gaf de mogelijkheid aan andere chipfabrikanten om — in licentie — deze interface toe te passen.

Binnen het I^2C-protocol heeft elke type component een unieke identificatiecode. Philips stelt deze codes in licentie ter beschikking. Als de microcontroller master

is, heeft de I²C-interface geen eigen identificatiecode nodig. Als de microcontroller slave is, heeft het een slave-adres nodig. Microcontrollerfabrikanten laten dit aan de ontwerper over. Deze kan dan zelf een code kiezen. De ATmega32 heeft wel een I²C-interface, maar heeft deze niet aangemeld bij Philips. Vanwege licenties gebruikt Atmel — net als andere microcontrollerfabrikanten — een andere naam voor de I²C-interface, namelijk TWI. Deze afkorting staat voor *Two Wire Interface* of *Two-Wire serial Interface*

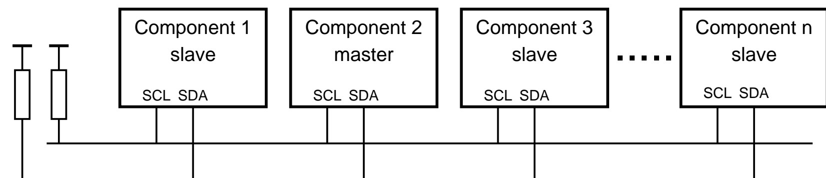
I²C kent voor de maximale kloksnelheid vier standaarden:
- *normal* (100 kHz),
- *fast* (400 kHz),
- *fast plus* (1 MHz),
- *high speed* ((3,4 MHz).

De hogere snelheden worden bereikt doordat er strengere eisen gesteld worden aan het tijdsgedrag. De meeste componenten hebben als maximale frequentie 400 kHz.

SPI en I²C zijn beide seriële communicatieprotocollen. De voordelen van SPI zijn: de kloksnelheid kan hoger zijn dan I²C, er is geen extra hardware nodig en er hoeven geen extra gegevens, zoals de identificatiecode, verstuurd te worden. De voordelen van I²C: er is een officiële standaard, er zijn minder lijnen nodig, in een systeem met veel componenten blijft het aantal lijnen twee en het is geschikt voor een multi-master systeem.

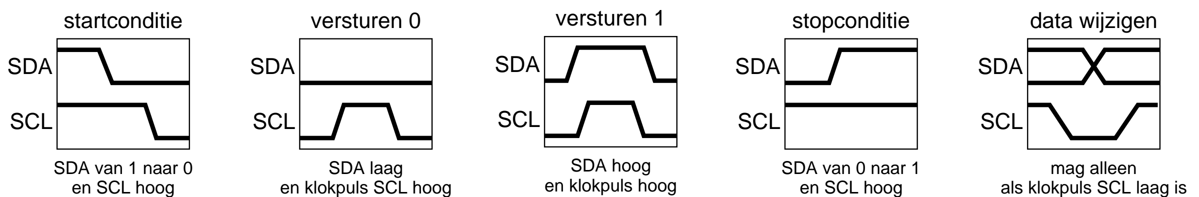
Componenten, die met een I²C-interface verkrijgbaar zijn, zijn bijvoorbeeld: SDRAM, EEPROM, flash, port expander, DAC, ADC, *real time clock*, LCD, temperatuursensor en vele andere sensoren.

De datalijn en de kloklijnen vormen allebei een *wired AND*-functie. De uitgang van de componenten kan laag of hoogimpedant zijn. Iedere component mag de lijn laag maken. Voor een hoog signaal op de lijn moet de uitgang van alle componenten hoogimpedant zijn.



Figuur 21.6 : Een I²C-configuratie met een master en meerdere slaves. De kloklijn SCL en de datalijn SDA zijn met twee weerstanden verbonden met de voeding. De master en de slaves zijn met elkaar verbonden via de SCL- en SDA-lijn.

In figuur 21.6 staat een voorbeeld van een configuratie met een master en een groot aantal slaves. De datalijn, SDA en de kloklijn, SCL, zijn via een pullupweerstand met de voeding verbonden. De signalen op deze lijnen kunnen door de I²C-componenten laag worden gemaakt. Informatie wordt doorgeven door de data- en de kloklijn laag te maken. Als een component een lijn omlaag trekt, zien de andere componenten dat.



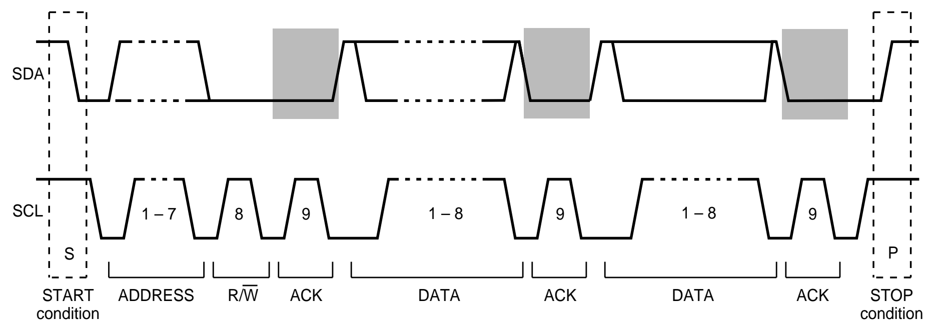
Figuur 21.7 : De betekenis van de verschillende signaalcondities met de SDA- en de SCL-lijn.

In figuur 21.7 staat de betekenis van de condities die met de klok- en datalijn samengesteld kunnen worden. Het I²C-protocol begint met een startconditie en

eindigt met een stopconditie. De master geeft een startconditie door de datalijn omlaag te trekken terwijl de kloklijn hoog is. De master beëindigt het protocol met een stopconditie; de datalijn wordt dan hoog gemaakt terwijl de kloklijn hoog is. De bits van de te versturen en te ontvangen informatie worden gevormd door de enen en nullen op de datalijn bij een positieve klokpuls. De bits op de datalijn mogen alleen veranderen als de klokpuls laag is.

Figuur 21.8 : Het protocol voor het versturen van gegevens door de master naar een slave. Na de startconditie START verstuurt de master het slave-adres ADDRESS, het schrijfbit W en de te versturen gegevens DATA. Na ontvangst van elke byte antwoord de slave door het bevestigingsbit ACK op de bus te zetten. Tenslotte sluit de master de communicatie met de STOP-conditie.

Figuur 21.8 toont het I^{2}C-protocol voor het versturen van twee databytes naar een slave. Als de master gegevens wil versturen, meldt deze zich met de startconditie op de bus. De master stuurt eerst het adres van de slave en stuurt een schrijfbit W met de waarde nul om aan te geven dat de slave gegevens moet ontvangen. Als de slave dit goed ontvangen heeft, antwoordt deze met een ACK-bit. Het bevestigingsbit heeft dan de waarde nul. Hierna kan de master de databytes versturen. Elk correct ontvangen byte bevestigt de slave met een ACK. Na het versturen van de databytes sluit de master de communicatie af met het versturen van de STOP-conditie.



Figuur 21.9 : De signalen voor het versturen van twee databytes door de master. De ACK-bits worden door de slave gegenereerd en hebben in de figuur een grijze achtergrond.

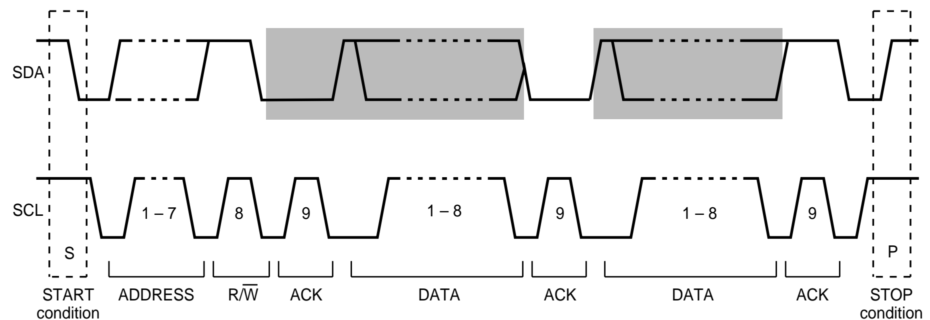
Figuur 21.9 toont de signaalwaarden van het protocol van figuur 21.8. Het kloksignaal wordt door de master gegenereerd. De slave zet de bevestigingsbits op de datalijn. De rest van het datasignaal maakt de master.

Het protocol voor het ontvangen van informatie van de slave door de master lijkt sterk op dat van het versturen van gegevens door de master. In figuur 21.10 staat dit protocol. Nadat het slave-adres is verstuurd, stuurt de master het leesbit R. De slave antwoordt daar op met een ACK en stuurt de databytes. De master bevestigt de ontvangst van elk byte met een ACK. Op het moment dat de master

Figuur 21.10 : Het protocol voor het ontvangen van gegevens door de master naar een slave. Na de startconditie START verstuurt de master het slave-adres ADDRESS, het schrijfbit R. De slave reageert met een ACK en stuurt het eerste databyte. De master bevestigt dat steeds met een ACK. Als de master stopt stuurt het geen ACK maar een NACK ($\overline{\text{ACK}}$). Zo weet de slave dat er geen gegevens meer verstuurd hoeven te worden. De master beëindigt de communicatie met de STOP-conditie.

voldoende informatie ontvangen heeft, reageert de master met een NACK ($\overline{\text{ACK}}$) en sluit de communicatie af met een STOP-conditie.

Figuur 21.11 toont de signaalwaarden van het protocol van figuur 21.10. Het kloksignaal wordt door de master gegenereerd. De master zet het slave-adres en het leesbit op de datalijn. De andere gegevens en het eerste bevestigingsbit zet de slave erop. De andere bevestigingsbits zet de master op de datalijn.



Figuur 21.11 : De signalen bij het ontvangen van twee databytes door de master. De bits, die door de slave gegenereerd worden, hebben een grijze achtergrond.

I²C is bij de microcontroller Atmel geïmplementeerd als TWI, *Two Wire Interface* of *Two-Wire serial Interface* en verschilt niet met een geautoriseerde I²C-bus. Er zijn vier registers bij betrokken: een register `TWBR` voor het instellen van de *bit rate*, een register `TWAR` voor het slave-adres voor het geval dat de microcontroller als slave wordt toegepast, een controlregister `TWCR` en een dataregister `TWDR`.

Code 21.11 bevat een aantal definities voor het maken van een I²C-verbinding. Deze paragraaf behandelt alleen een master-slave systeem met de ATmega32 als master. De klokfrequentie van I²C-interface wordt bepaald door de master. De macro `F_SCL` bevat deze klokfrequentie. Het ingesloten headerbestand `util/twi.h` hoort bij de `avr-libc`-bibliotheek en bevat de definities voor het bepalen van de status van de TWI. Alle namen uit code 21.12 die met `TW_` beginnen zijn in dit bestand gedefinieerd. Met de definities en functies van `i2c.h` en `i2c.c` kan de ontwerper een I²C-verbinding maken.

In code 21.12 staat het bestand `i2c.c` met de functies voor de communicatie met de I²C-interface. Naast een functie voor de initialisatiefunctie zijn er vijf functies voor het schrijven, het lezen en het starten, herstarten en stoppen van de communicatie. Deze functies zetten steeds de bits voor de betreffende actie in het `TWCR`-register en wachten daarna tot de actie is uitgevoerd.

Code 21.11 : Het headerbestand `i2c.h` met definities en prototypes.

```
1  #include <inttypes.h>
2  #include <util/twi.h>
3
4  #define F_SCL       100000UL
5  #define I2C_ACK     0
6  #define I2C_NACK    1
7  #define I2C_READ    1
8  #define I2C_WRITE   0
9
10 void    i2c_init(void);
11 uint8_t i2c_start(void);
12 uint8_t i2c_restart(void);
13 void    i2c_stop(void);
14 uint8_t i2c_write(uint8_t data);
15 uint8_t i2c_read(uint8_t ack);
```

De functie `i2c_init` initialiseert de TWI-interface. Door eerst het controlregister `TWCR` leeg te maken, kan tijdens de initialisatie de communicatie niet per ongeluk gestart worden. De ATmega32 heeft geen identificatiecode. Deze code is alleen van belang als de microcontroller als slave wordt gebruikt. Register `TWAR` wordt hier niet gebruikt en is leeg gemaakt. Het `TWSR` bevat twee bits `TWPS1` en `TWPS0` waarmee de systeemklok geschaald wordt. Samen met de waarde in `TWBR`, het *TWI Bit rate Register*, bepalen deze gegevens de frequentie van de I^2C-klok. Deze frequentie is:

$$f_{\text{scl}} = \frac{f_{\text{cpu}}}{16 + 2(\texttt{TWBR})4^{\texttt{TWPS}}} \tag{21.1}$$

De prescaling is alleen noodzakelijk voor een extreem lage kloksnelheid of bij een hoge frequentie van de systeemklok. Bij een maximale klokfrequentie van 16 MHz is prescaling niet nodig. De klokfrequentie is dan:

$$f_{\text{scl}} = \frac{f_{\text{cpu}}}{16 + 2(\texttt{TWBR})} \tag{21.2}$$

De waarde van `TWBR` voor een gewenste klokfrequentie wordt berekend met:

$$\texttt{TWBR} = \frac{\frac{f_{\text{cpu}}}{f_{\text{scl}}} - 16}{2} \tag{21.3}$$

In code 21.12 wordt op regel 8 met formule 21.3 de waarde berekend en toegekend aan `TWBR`. De datasheet van de ATmega32 adviseert om altijd een waarde groter of gelijk aan `10` te kiezen voor `TWBR`. De voorwaarde op regel 9 zorgt er voor dat dit het geval is.

De functie `i2c_start` zet de bits in het controlregister. De startconditie wordt dan automatisch verstuurd. Na afloop wordt het `TWINT`-bit hoog gemaakt. De functie wacht op regel 17 totdat dit bit hoog is. Als het starten succesvol is, staat in het statusregister de code `TW_START`. Bij een succesvolle start geeft de functie een `0` terug en anders een 1. De functie `i2c_restart` is identiek met `i2c_start`. Het enige verschil is dat bij succes in het `TWSR` een andere code staat.

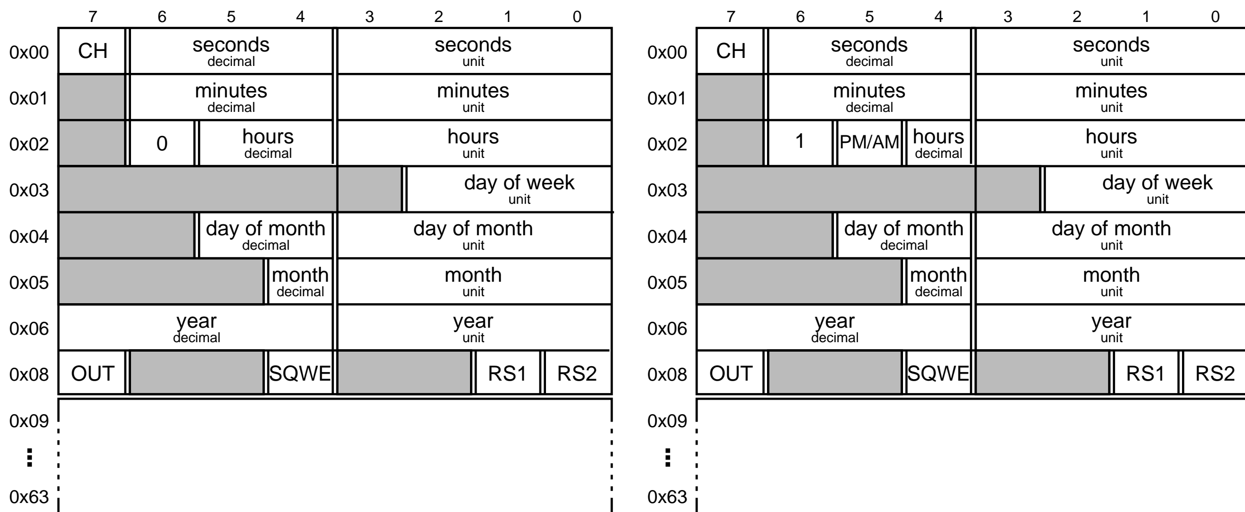
Code 21.12 : De I^2C-functies uit **i2c.c** voor het versturen en ontvangen.

```
1  #include "i2c.h"
2
3  void i2c_init(void)
4  {
5    TWCR = 0x00;
6    TWAR = 0x00;                            // slave address
7    TWSR = 0x00;                            // no prescaling
8    TWBR = ((uint8_t)(F_CPU/F_SCL) - 16)/2;
9    if (TWBR < 10 ) {                       // TWBR must be 10 or more
10     TWBR = 10;
11   }
12 }
13
14 uint8_t i2c_start(void)
15 {
16   TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWSTA);
17   while ( !(TWCR & _BV(TWINT)) );
18
19   if ( TW_STATUS != TW_START ) return 1;
20   return 0;
21 }
22
23 uint8_t i2c_restart(void)
24 {
25   TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWSTA);
26   while ( !(TWCR & _BV(TWINT)) );
27
28   if ( TW_STATUS != TW_REP_START ) return 1;
29   return 0;
30 }
31
32 void i2c_stop(void)
33 {
34   TWCR =  _BV(TWINT)|_BV(TWEN)|_BV(TWSTO);
35   while ( (TWCR & _BV(TWSTO)) );
36 }
37
38 uint8_t i2c_write(uint8_t data)
39 {
40   TWDR = data;
41   TWCR = _BV(TWINT)|_BV(TWEN);
42   while ( !(TWCR & _BV(TWINT)) );
43
44   if( TW_STATUS != TW_MT_DATA_ACK) return 1;
45   return 0;
46 }
47
48 uint8_t i2c_read(uint8_t ack)
49 {
50   TWCR = _BV(TWINT)|_BV(TWEN)|( ( ack == I2C_ACK) ? _BV(TWEA) : 0 );
51   while ( !(TWCR & _BV(TWINT)) );
52
53   return TWDR;
54 }
```

De functie `i2c_stop` maakt het `TWSTO`-bit hoog om de stopconditie te versturen. Na de stop actie is uitgevoerd, wordt het bit automatisch laag gemaakt. De functie wacht hierop.
Voor het schrijven zet de functie `i2c_write` de te verzenden byte in het dataregister en maakt het `TWINT`-bit hoog. De functie wacht, net als de startfuncties, op het laag worden van het `TWINT`-bit. Bij succes geeft `i2c_write` een `0` terug en anders een 1. Voor het lezen wordt ook het `TWINT`-bit hoog gemaakt. Het onderscheid tussen lezen en schrijven wordt gemaakt doordat bij het versturen van het slave adres het `R/W̄`-bit hoog of laag is gemaakt. De functie `i2c_read` geeft na het lezen van de byte automatisch een `ACK`. Als het `TWEA`-bit hoog is, wordt er een `ACK` gegeven en als het bit laag is een `NACK`. Functie geeft tenslotte de gelezen databyte terug.

Dallas Semiconductor is in januari 2001 overgenomen door Maxim.

Een voorbeeld van een I^2C-component is de DS1307 real time clock van Dallas. Deze component bevat een oscillator en houdt de tijd en de datum bij. Deze gegevens worden bewaard in een 64 × 8 NV-RAM, *Non Volatile RAM* en zijn via I^2C van buitenaf bereikbaar.



Figuur 21.12 : De geheugenindeling bij de DS1307 van Dallas. Links staat de indeling voor de 24-uur modus en rechts die voor de 12-uur modus. De grijs gekleurde bits worden niet gebruikt.

In figuur 21.12 staat de geheugenindeling. De eerste acht bytes bevatten de informatie voor tijd en datum. De gegevens zijn BCD gecodeerd. Het hoogste nibble bevat het tiental en het laagste nibble de eenheid; zo betekent `0010 1001` bijvoorbeeld 29. De bits 6 tot met `0` van de eerste byte bevatten het aantal seconden.

Het hoogste bit van de eerste byte is de vlag `CH`, *Clock Halt*, hiermee kan de oscillator aan- en uitgezet worden. Normaal gesproken is dit bit altijd laag.

De bits 6 tot met `0` van de tweede byte bevat de minuten. De uren staan in de derde byte en kunnen in een 12-uurs en 24-uurs modus worden opgeslagen. Voor de 24-uurs notatie is bit 6 laag en stellen de bits 5 tot en met `0` de uren voor. Voor de 24-uurs notatie is bit 6 hoog en stellen de bits 4 tot en met `0` de uren voor. Bit 5 is hoog als het na de middag (pm) is en laag als het voor de middag (am) is.

De volgende vier bytes bevatten de dag van de week, de dag, de maand en het jaar. De dag van de week is een getal 1 tot en met 7. Het jaar bevat alleen een tiental en een eenheid.

Code 21.13 : Het bestand `rtc.h` met definities voor de DS1307 real time clock.

```
1  #define RTC_SLAVE_ADDRESS   0xD0
2
3  #define RTC_SECOND          0x00
4  #define RTC_MINUTE          0x01
5  #define RTC_HOUR            0x02
6  #define RTC_DAY             0x03
7  #define RTC_DATE            0x04
8  #define RTC_MONTH           0x05
9  #define RTC_YEAR            0x06
10 #define RTC_CONTROL         0x07
11
12 void  rtc_set_time(void);
13 void  rtc_set_date(void);
14 void  rtc_get_time(void);
15 void  rtc_get_date(void);
16 char *rtc_time_to_string(char *s);
17 void  string_to_rtc_time(char *s);
18
19 struct rtc_time {
20   uint8_t second;
21   uint8_t minute;
22   uint8_t hour;
23 };
24
25 struct rtc_date {
26   uint8_t day;
27   uint8_t month;
28   uint8_t year;
29 };
```

Code 21.14 : Deel van `rtc.c` met de functies `rtc_set_time` en `rtc_get_date`.

```
1  #include "i2c.h"
2  #include "rtc.h"
3
4  struct rtc_time time;
5  struct rtc_date date;
6
7  void rtc_set_time(void)
8  {
9    i2c_start();
10   i2c_write(RTC_SLAVE_ADDRESS|I2C_WRITE);
11   i2c_write(RTC_SECOND);
12   i2c_write(time.second);
13   i2c_write(time.minute);
14   i2c_write(time.hour);
15   i2c_stop();
16 }
17
18 void rtc_get_date(void)
19 {
20   i2c_start();
21   i2c_write(RTC_SLAVE_ADDRESS|I2C_WRITE);
22   i2c_write(RTC_DATE);
23   i2c_restart();
24   i2c_write(RTC_SLAVE_ADDRESS|I2C_READ);
25   date.day   = i2c_read(I2C_ACK);
26   date.month = i2c_read(I2C_ACK);
27   date.year  = i2c_read(I2C_NACK);
28   i2c_stop();
29 }
```

Bij de verwerking van de datum en tijd is het handig om de gegevens van de DS1307 eerst in een datastructuur te plaatsen. Dat kan bijvoorbeeld een array van zeven bytes zijn:

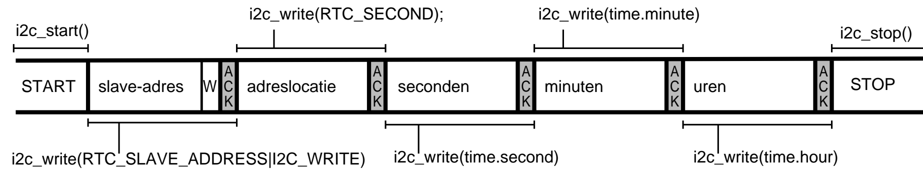
```
uint8_t ds1307[7];
```

Het eerste byte `ds1307[0]` is dan het aantal seconden en het zevende byte `ds1307[6]` is dan het jaar.

In code 21.13 en code 21.14 is gekozen om de tijd en de datum op te slaan in twee datastructuren `time` en `date`. Deze datastructuren zijn gedeclareerd in het headerbestand `rtc.h`. Het voordeel is dat de verwijzingen naar de verschillende velden zeer goed leesbaar zijn: `time.hour` geeft het uur.

De dag van de week is in dit voorbeeld weggelaten. Dit veld kan worden toegevoegd aan de structuur `date`.

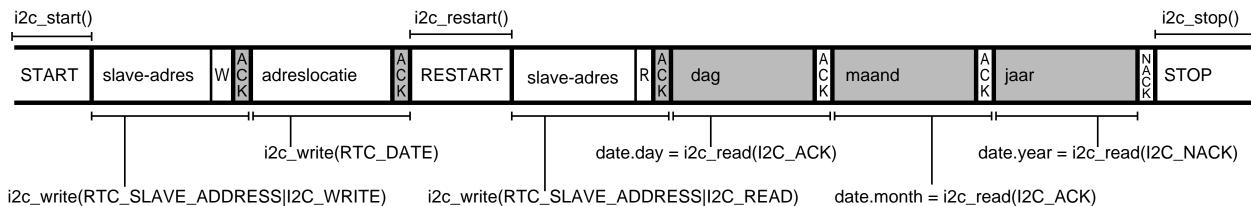
Voor het instellen van een DS1307 real time clock wordt na het slave-adres en het schrijfbit eerst de adreslocatie verstuurd van waar af de bytes ingesteld moet worden. Na het adres van de geheugenlocatie volgen de bytes.
De functie `rtc_set_time` uit code 21.14 stelt de tijd van de DS1307 in. Na het slave-adres volgt eerst het adres `RTC_SECOND` van de locatie waar de seconden staan.



Figuur 21.13 : Het dataformaat voor het instellen van de tijd bij de DS1307.

Daarna volgen de drie bytes met de nieuwe waarden uit de structuur `time` met de seconden, minuten en uren. Figuur 21.13 toont het dataformaat dat verstuurd wordt samen met de toewijzingen uit de functie `rtc_set_time`.

Voor het uitlezen van een DS1307 moet eerst de adreslocatie worden verstuurd van waar de gegevens gelezen moeten worden. Achtereenvolgens wordt eerst het slave-adres, het schrijfbit en de adreslocatie verstuurd. Nadat de communicatie opnieuw gestart is, worden opnieuw het slave-adres met het leesbit verstuurd en worden de bytes gelezen.



Figuur 21.14 : Het dataformaat voor het lezen van de datum bij de DS1307.

De functie `rtc_get_date` uit code 21.14 leest de datum uit de DS1307. Na het slave-adres met de schrijfbit (`W`) wordt eerst het adres `RTC_DATE`, van de locatie waar de dag staat, verstuurd. Daarna wordt opnieuw het slave-adres, maar nu met het leesbit (`R`), verstuurd en worden de drie bytes met de dag, de maand en het jaar gelezen en in de datastructuur `date` geplaatst. Figuur 21.14 laat het formaat van de communicatie zien, samen met de toewijzingen uit de functie `rtc_get_date`. Het bestand `rtc.c` bevat ook de functies `rtc_get_time` en `rtc_set_date`. Deze lijken sterk op de functies uit code 21.14.

De waarden van de tijd en datum zijn BCD gecodeerd. Om iets met deze waarden te kunnen doen, moeten deze omgezet worden naar afdrukbare karakters of naar een binaire representatie. In code 21.15 staan twee conversiefuncties voor het omzetten van de tijd naar een afdrukbare string en omgekeerd. Het formaat van de string is `HH:MM:SS`. De eerste twee karakters zijn het tiental en de eenheid van de uren. De eenheid wordt toegekend door de functie `rtc_time_to_string` aan `s[1]` en is het lage *nibble* van het veld `time.hour`. Dit is een waarde van `0` tot en met 9.

De ASCII-waarden van de cijfers `0` tot en met 9 zijn hexadecimaal `0x30` tot en met `0x39`. Met behulp van de bitsgewijze OF wordt het lage *nibble* omgezet naar de juiste ASCII-waarde. Het tiental is het hoge *nibble* en wordt toegekend aan `s[0]`. De betreffende bits worden gemaskeerd en vier posities naar rechts geschoven en op dezelfde wijze omgezet naar de ASCII-waarde van het betreffende cijfer.

Code 21.15 : Conversiefuncties uit `rtc.c` voor het omzetten van het `rtc_time`.

```
1  char *rtc_time_to_string(char *s)
2  {
3    s[0] = 0x30 | ((time.hour & 0x30) >> 4);
4    s[1] = 0x30 |  (time.hour & 0x0F);
5
6    s[3] = 0x30 | ((time.minute & 0x70) >> 4);
7    s[4] = 0x30 |  (time.minute & 0x0F);
8
9    s[6] = 0x30 | ((time.second & 0x70) >> 4);
10   s[7] = 0x30 |  (time.second & 0x0F);
11
12   return s;
13 }
```

```
1 void string_to_rtc_time(char *s)
2 {
3   time.hour   = ((s[0] & 0x03)<<4) | (s[1] & 0x0F);
4   time.minute = ((s[3] & 0x07)<<4) | (s[4] & 0x0F);
5   time.second = ((s[6] & 0x07)<<4) | (s[7] & 0x0F);
6 }
```

De karakters `s[2]` en `s[5]` zijn de scheidingstekens en blijven ongewijzigd. De minuten worden toegekend aan `s[3]` en `s[4]` en de seconden aan `s[6]` en `s[7]`. De buffer waar `rtc_time_to_string` naar schrijft, moet een string met het juiste formaat zijn.

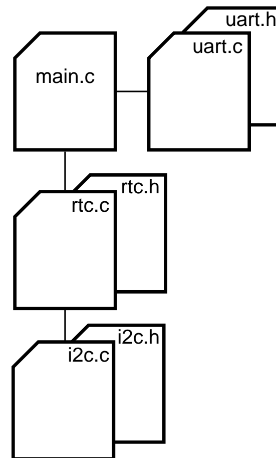
De functie `string_to_rtc_time` doet het omgekeerde. De karakters `s[1]`, `s[4]` en `s[7]` bevatten de eenheden van de uren, minuten en seconden. Het masker `0x0F` geeft alleen het lage *nibble* door. Dit wordt samengevoegd met de relevante bits van de tientallen, die vier posities naar links zijn geschoven.

Code 21.16 : Programma dat de tijd instelt op een vaste waarde en daarna elke seconde een tijdmelding geeft. (f_{cpu} = 8 MHz)

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <util/delay.h>
4  #include "uart.h"
5  #include "i2c.h"
6  #include "rtc.h"
7
8  #define BAUD   38400   // F_CPU is 8000000 Hz
9
10 int main(void)
11 {
12   char t[]="HH:MM:SS";
13
14   i2c_init();
15   uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
16   sei();
17
18   string_to_rtc_time("08:53:38");
19   rtc_set_time();
20   while(1) {
21     rtc_get_time();
22     uart_puts(rtc_time_to_string(t));
23     _delay_ms(1000);
24   }
25 }
```

Het hoofdprogramma uit code 21.16 stelt, na de initialisatie van de I²C-interface en de UART, de tijd van de DS1307 in op `08:53:38`. Vervolgens verstuurt het via de UART elke seconde de actuele tijd.

Het is praktischer om via bijvoorbeeld de UART de tijd in te stellen op een moment dat dat gewenst is. Deze functionaliteit is hier weggelaten, om de uitleg enigszins beperkt te houden.
Het hoofdbestand `main.c` maakt gebruik van `uart.c` en `rtc.c` Het bestand `rtc.c` bevat alleen de functies voor het benaderen van de DS1307. De I²C-functies die daarvoor nodig zijn staan in `i2c.c`. Het bestand `i2c.c` heeft geen kennis van de component waarmee het communiceert, het bevat alleen basale functies om via I²C te communiceren. De kennis over DS1307 zit in bestand `rtc.c`. Figuur 21.15 toont de samenhang tussen de diverse bestanden die voor de communicatie met DS1307 nodig zijn.



Figuur 21.15 : De organisatie van de bestanden bij het lezen van en schrijven naar de DS1307.