# Microcontrollers

## en de taal C

vierde druk

Wim Dolman

Informatie over dit boek en andere uitgaven kunt u verkrijgen bij:
Wim Dolman
info@dolman-wim.nl
http://mic.dolman-wim.nl/

*Microcontrollers en de taal C* is geschreven door Wim Dolman en in eigen beheer gedrukt bij Free Musketeers uitgeverij en producties.

eerste druk, 2008
tweede druk, 2009
derde druk, 2010
vierde druk, 2011

Vormgeving en opmaak is verzorgd door de auteur.
De illustratie op de omslag is een vrije bewerking van de reproductie 'De Visser' van de Chinese landschapsschilder Ma Yuan. Dit schilderij inspireerde Lucebert tot het gedicht 'De visser van Ma Yuan'. Dit gedicht, dat op bladzijde 176 staat, geeft een goed beeld van het geduld, de rust en de kracht die een ontwerper van embedded systemen nodig heeft.

ISBN: 978-90-484-0835-1
NUR: 173/959

# Voorwoord

Microcontrollers is een thema dat gegeven wordt in het eerste jaar van de opleiding E-technology van de Hogeschool van Amsterdam. Dit vak behandelt de ATmega32 van Atmel en de programmeertaal C. Studenten, die dit vak volbracht hebben, kunnen een ontwikkeltraject voor een microcontroller uitzoeken, een demonstratieopstelling maken en met de taal C een programma voor de microcontroller schrijven. Hiervoor is kennis van de taal C nodig en kennis van microcontrollers in het algemeen. Bovendien moet de student weten hoe de ATmega32 is opgebouwd, op de hoogte zijn van de specifieke mogelijkheden van deze microcontroller en kennis hebben van de taal C voor de ATmega32.

Dit boek bevat het studiemateriaal voor het thema Microcontrollers en en bestaat uit drieëntwintig hoofdstukken die te groeperen zijn in zes delen.

- Hoofdstuk 1 is een inleiding over embedded systemen. De microcontroller is de belangrijkste component van een embedded systeem en het is het belangrijkste toepassingsgebied van deze component. Voor de ontwikkeling van embedded systemen wordt de taal C veel gebruikt.
- De hoofdstukken 2 tot en met 9 behandelen de basis van de taal C zonder een microcontroller. In deze hoofdstukken worden alleen applicaties voor de pc gemaakt. Aan bod komen: het compilatietraject, declaraties van variabelen, uitvoer naar het beeldscherm, invoer met het toetsenbord, gebruik van functies, voorwaardelijke opdrachten, herhalingsopdrachten, standaard datatypen en de opmaak van de code.
- Hoofdstuk 10 bespreekt een aantal eigenschappen van de ATmega32. Deels is dit een samenvatting van de datasheet van de ATmega32, anderzijds is het juist een aanvulling op wat in de datasheet staat. Tevens wordt in dit hoofdstuk het ontwikkeltraject voor de ATmega32 besproken.
- De hoofdstukken 11 tot en met 13 behandelen de taal C, zoals deze gebruikt wordt bij de ATmega32. Er worden diverse voorbeelden behandeld: een knipperende led, de dotmatrix, het 7-segmentsdisplay, de externe interrupts en de tellers. Van alle onderwerpen wordt een demonstratievoorbeeld besproken. Naast de uitleg van de code wordt de interne opbouw van ATmega32 verder verklaard en is er aandacht voor de hardware van de demonstratieschakelingen, zoals de aansturing van leds, het ontkoppelen van de ATmega32 en het bestrijden van contactdender.
- In de hoofdstukken 14 tot en met 17 komen de meer geavanceerde onderwerpen van de taal C aan de orde, zoals arrays, pointers, strings, recursie, datastructuren en het lezen van en schrijven naar bestanden.

- Hoofdstuk 18 tot en met 23 bespreken de meer geavanceerde onderwerpen van de ATmega32. Aan de orde komen: de AD-converter, het aansturen van een LCD, de communicatie met de USART, de SPI en de $I^2C$-interface en het gebruik van de interne en een externe EEPROM, de toepassing van de timers bij pulsbreedtemodulatie, de analoge comparator, de slaapstanden, de watchdog en brownoutdetectie. Deze hoofdstukken geven over de besproken features veel achtergrondinformatie en tonen meerdere manieren waarop deze gebruikt kunnen worden.

Om dit boek zo concreet mogelijk te maken is het geschreven rond een bepaalde microcontroller. Er is gekozen voor de ATmega32 van Atmel. Dat is een populaire microcontroller uit de ATmega-reeks, die zeer geschikt is voor de beginnende ontwerper van embedded systemen. In 2008 heeft Atmel de ATmega32 vervangen door de ATmega32A, die alleen elektrisch anders is. Overal in dit boek mag in plaats van ATmega32 ook ATmega32A gelezen worden. Veel van wat in dit boek staat, geldt overigens ook voor de andere ATmega's.

Dit boek is geschreven in LaTeX. Teksten met programmacode kunnen veel beter en eenvoudiger gemaakt worden met LaTeX, dan met een programma als Word. De gebruikte compiler is pdfeTeX, die standaard bij de Cygwin-omgeving zit. De hoofdtekst is gezet in Garamond; voor de wiskundige formules is Math Design en voor de programmacode is Bera Mono gebruikt. Dit laatste font is de TeX-variant van Bitstream Vera Mono. De opmaak van de code is gedaan met de *lstlisting*-omgeving. De gereserveerde namen zijn in de code en in de tekst vet gedrukt. De meeste tekeningen zijn gemaakt met Mayura Draw en als PDF-bestand toegevoegd.
Het boek bevat meer dan vijfhonderd figuren, tabellen en complete programma's. Daarnaast bevat het boek ook nog honderden codefragmenten. Om de leesbaarheid zo groot mogelijk te maken, hebben de toelichtingen bij de programmacodes in de marge een lichtgrijs vlak met daarin een regelnummer en het onderwerp dat aan de orde is. De marge wordt ook gebruikt voor korte losse opmerkingen en voor paginabrede figuren, tabellen en codes. Het boek bevat een groot aantal complete programmacodes, die zonder enige aanpassing direct gecompileerd en uitgevoerd kunnen worden.

Alle codes in het boek zijn grondig getest. De algemene C is getest met de GNU C-Compiler versie 3.4.4, die bij de Cygwin-omgeving versie 1.5.24 hoort. De C voor de ATmega32 is getest met AVRstudio versie 4.16.638 met de plug-in WinAVR versie 20090313. Deze plug-in gebruikt de GNU C-Compiler voor de AVR versie 4.3.2 en de AVR C-bibliotheek versie 1.6.6.

Bij dit boek hoort een internetsite: `http://mic.dolman-wim.nl/` met een studiewijzer, alle voorbeeldprogramma's, oefenmateriaal, practicumopdrachten en informatie over de gebruikte software.
Het studieonderdeel Microcontrollers behandelt en toetst niet alles wat in dit boek besproken wordt. Het boek is ook bedoeld voor verdere zelfstudie bij andere vakken en projecten. Op de internetsite staat een studiewijzer met de onderwerpen die wel en niet bij de stof horen.

# Inhoud

# Bijlagen

# 1

# De Microcontroller

Doelstelling

In dit hoofdstuk leer je wat een microcontroller is, hoe deze is opgebouwd en wat een embedded systeem is waarbinnen de microcontroller toegepast wordt.

Onderwerpen

De behandelde onderwerpen zijn:

- De toepassing en de commerciële ontwikkeling van embedded systemen.
- Het verschil tussen een microcontroller en een microprocessor.
- De opbouw van een microcontroller en met name de geheugenorganisatie: de Harvard- en de Von Neumann-architectuur.
- Het verschil tussen CISC- en RISC-processoren.
- De criteria voor het kiezen van een microcontroller.

Microcontrollers spelen een belangrijke rol in zogenoemde embedded systemen. In dit hoofdstuk wordt verteld wat embedded systemen zijn en wat een microcontroller is, hoe deze component is opgebouwd en waarom deze zo belangrijk is voor de kleinere embedded systemen.


**Figuur 1.1 : Voorbeelden van embedded systemen.**

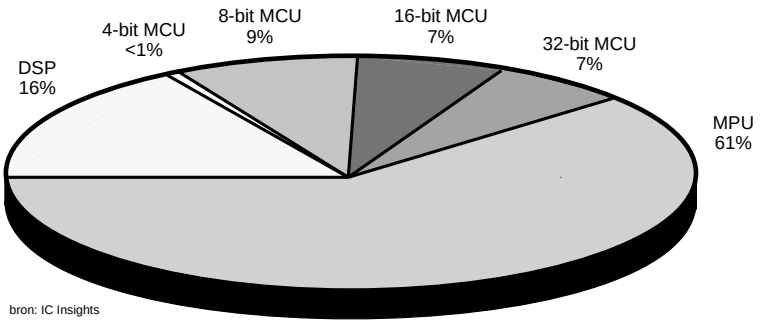
## 1.1 Embedded Systemen

*Embedded systems* betekent letterlijk ingebedde systemen.

De meeste mensen realiseren zich niet dat er in hun huis tientallen computersystemen aanwezig zijn. Dit soort 'computers' noemt men *embedded systems* en zijn te vinden in allerlei apparaten, zoals in: televisies, magnetrons, videospelletjes, fototoestellen, kamerthermostaten, scheerapparaten, wekkerradio's, polshorloges en alarmsystemen. Embedded systemen kom je niet alleen thuis, maar overal tegen. In auto's zitten bijvoorbeeld ook tientallen embedded systemen; zelfs een — op het eerste gezicht — eenvoudig onderdeel als een schokbreker bevat soms een embedded systeem.

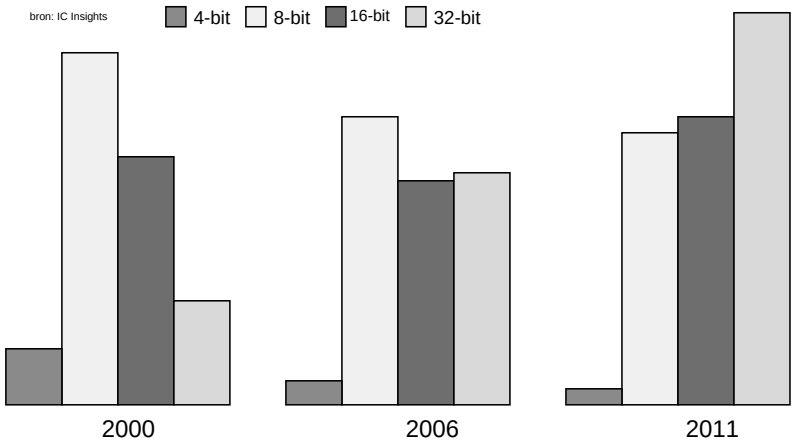
Er zijn tientallen verschillende definities van een embedded systeem. Een hele fraaie definitie is deze: 'Een embedded systeem is een intelligent systeem dat er niet uit ziet als een computer! Dus zonder muis, beeldscherm en toetsenbord.'.

De computers in deze apparaten hoeven slechts een beperkt aantal taken uit te voeren. De consument kan ook maar een beperkt aantal instellingen opgeven. Je kan bijvoorbeeld de tijd opgeven wanneer je gewekt wil worden. De software, die er dan voor zorgt dat je op tijd gewekt wordt, is al door de ontwerper van het apparaat geschreven en maakt deel uit het van het embedded systeem. Bij het realiseren van een embedded systeem krijgt men dus niet alleen te maken met de hardware, maar ook met de software van het systeem. Men spreekt ook wel van *embedded software*.
Embedded software geeft een grotere functionaliteit en betere betrouwbaarheid aan het apparaat; het verhoogt de intelligentie van het systeem. De toepassing ervan kan bovendien leiden tot flexibelere producten, lagere kosten door eenvoudigere aanpassingen en een hogere toegevoegde waarde door het realiseren van functionaliteiten, die via hardware niet zijn aan te brengen. Embedded systemen kom je daarom op steeds meer plaatsen tegen.




**Figuur 1.2** : **De wereldwijde omzet van processoren in 2006.**
In het diagram staat de relatieve omzet van microcontrollers (MCU), digitale signaalprocessoren (DSP) en gewone processoren (MPU).

De productie van embedded processoren is — in volume — meer dan een factor honderd groter dan de productie van processoren voor desktop-pc's en laptops. Bij een vergelijking in omzet ligt dat natuurlijk anders, zoals in figuur 1.2 is te zien.
In een pc is de processor de component die het systeem intelligent maakt. Bij embedded systemen is het hart van het systeem vaak een microcontroller. De verschillen en de overeenkomsten tussen microprocessoren en microcontrollers worden in paragraaf 1.2 besproken.
Microcontrollers worden in enorme aantallen geproduceerd. Figuur 1.3 geeft de ontwikkeling en de prognose van de wereldwijde omzet van deze componenten. Veelal zitten deze in massaproducten, zoals bijvoorbeeld: elektrische tandenborstels en i-pods. Maar ze zitten ook in artikelen met kleine productievolumes. Denk bijvoorbeeld aan een tomatensorteermachine. Embedded systemen hebben altijd eigen speciale software nodig. Ze moeten immers een specifieke taak uitvoeren. Juist omdat het vaak unieke systemen zijn, moet er veel software voor worden geschreven. Het ontwikkelen van embedded software is en wordt in komende jaren een enorme markt. Nu is de hoeveelheid embedded software al groter dan de hoeveelheid software voor pc's en mainframes.
Een embedded systeem bevat vaak meerdere microcontrollers en het hart van een embedded systeem hoeft niet per se een microcontroller te zijn. Tabel 1.1 geeft




**Figuur 1.3 : De wereldwijde omzet van microcontrollers.**
De diagrammen geven de omzet van microcontrollers van 4-, 8-, 16-, en 32-bits in 2000 en 2006 en een voorspelling voor 2011.

een aantal andere mogelijkheden. Afhankelijk van de toepassing, de prijs, het afzetvolume, de snelheid, het vermogen en andere technische specificaties zal er voor een bepaalde component gekozen worden. Dit boek behandelt de microcontroller en gaat daarom over de kleinere embedded systemen.

**Tabel 1.1 : De componenten die het hart van een embedded systeem kunnen vormen.** De rechter kolom geeft de vakcode van het vak bij de opleiding E-technology waar deze component behandeld wordt.

| component | omschrijving | vakcode |
|---|---|---|
| MPU | 32-bits of 64-bits microprocessor unit van standaard pc tot een embedded module, zoals een PC104 | CA |
| MCU | 4-bits-, 8-bits, 16-bits, of 32-bits microcontroller unit | MIC |
| DSP | Digital Signal Processor | DS |
| PLC[1] | Programmable Logic Circuit voor industriële toepassingen | BT |
| discreet[1] | Schakeling opgebouwd uit discrete componenten, bijvoorbeeld met de 7400-serie of CMOS 4000-serie | DT |
| ASIC | Application Specific Integrated Circuit | DI |
| PLD | Programmable Logical Device, bijvoorbeeld een FPGA (Field Programmable Gate Array), een PAL (Programmable Array Logic) of een CPLD (Complex Programmable Logical Device) | DI |

[1] Strikt genomen zijn dit geen embedded systemen. PLC's zijn intelligent, maar zeker niet klein en zijn niet ingebed. Een systeem met discrete componenten zal beperkt zijn qua omvang en dus ook qua intelligentie.

Interessant is om een pc te vergelijken met een eenvoudige 8-bit microcontroller. Tabel 1.2 vergelijkt een gewone desktop-pc met een PIC16F628 microcontroller van Microchip. De verschillen zijn enorm groot.

Assembly is een programmeertaal die zeer dicht staat bij de machinetaal. Een assembler vertaalt deze taal naar de machinecodes die de processor kan uitvoeren.

Omdat dit boek over de microcontroller en de kleinere embedded systemen gaat, is de programmeertaal die gebruikt wordt C. Zelfs bij de kleinere microcontrollers wordt Assembly weinig gebruikt en is er een tendens om C++ te gebruiken. Bij grotere processoren wordt vooral C++ gebruikt. Maar C# en Java worden ook toegepast. Het voordeel van C en C++ is dat beide talen geschikt zijn voor zogenoemde hard real time systemen.

**Tabel 1.2 : Een vergelijking tussen een pc (uit 2003) en een microcontroller.**

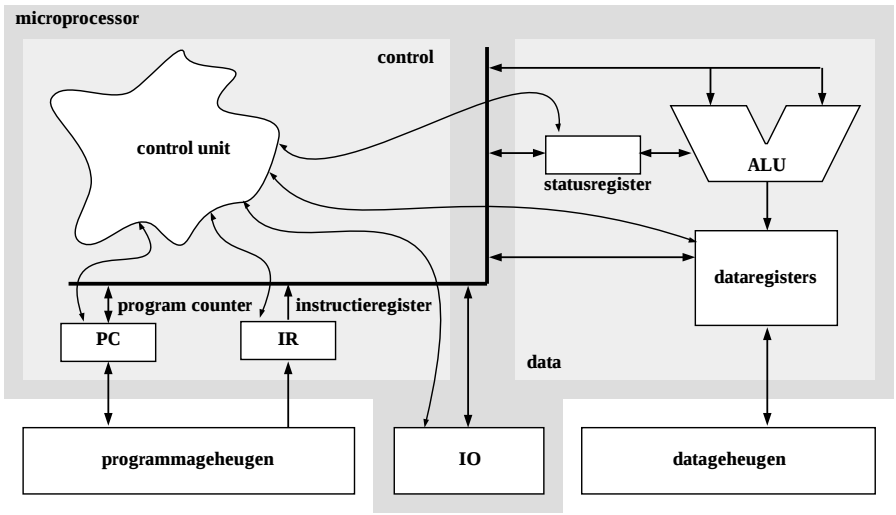
| symbool | pc met Pentium V | PIC16F628 |
|---|---|---|
| **programmageheugen** | 2 MB instructies in cache, 320 GB harde schijf | 3k6 bytes flash |
| **kloksnelheid** | 1,8 GHz en bussnelheid 1066 MHz | 20 kHz |
| **rekengeheugen** | 1 GB RAM | 224 byte RAM, 128 bytes data EEPROM |
| **processor** | 64 bits, 2,6 GHz, 20000 MIPS, 14000 MFLOPS | 8 bits, 20 MHz, 5 MIPS, 2 kFLOPS |
| **randapparatuur** | monitor, toetsenbord, muis, USB, parallelle poorten, seriële poorten, netwerk, joystick, MIDI, luidsprekers, microfoon | 16 IO-pinnen |
| **stroomverbruik** | 150 W zonder beeldscherm (processor alleen al 60 W) | 0,5 W |
| **omvang** | 40 × 40 × 20 cm | 15,8 × 10 × 5 mm |
| **gewicht** | 13 kg | 1 g |
| **prijs** | € 500 | € 1,5 |

MIPS betekent *Mega Instruction Per Seconde* en is het aantal miljoenen instructies dat de processor in een seconde uit kan voeren.

FLOPS betekent *FLoating point Operations Per Seconde* en is het aantal bewerkingen met gebroken getallen dat de processor in een seconde kan berekenen.

## 1.2 De architectuur van de microprocessor en de microcontroller

Een microprocessor en een microcontroller zijn twee verschillende componenten. Om deze verschillen goed te begrijpen, moet er eerst iets worden verteld over de architectuur van een microprocessor.




**Figuur 1.4 : De architectuur van een microprocessor.** De besturing (*control*) van de microprocessor zet een instructie uit het programmageheugen naar het instructieregister (IR). Het adres waar de besturing de instructie vandaan moet halen staat in de *program counter* (PC). Daarna voert de besturing de instructie uit. Er worden bijvoorbeeld data vanuit het datageheugen in de dataregisters gezet, de ALU voert daarna een berekening uit en het resultaat wordt naar het dataregister geschreven. In plaats van het dataregister kan er ook naar de IO (*input/output*) of naar de andere registers worden geschreven of van de IO of uit de andere registers worden gelezen.

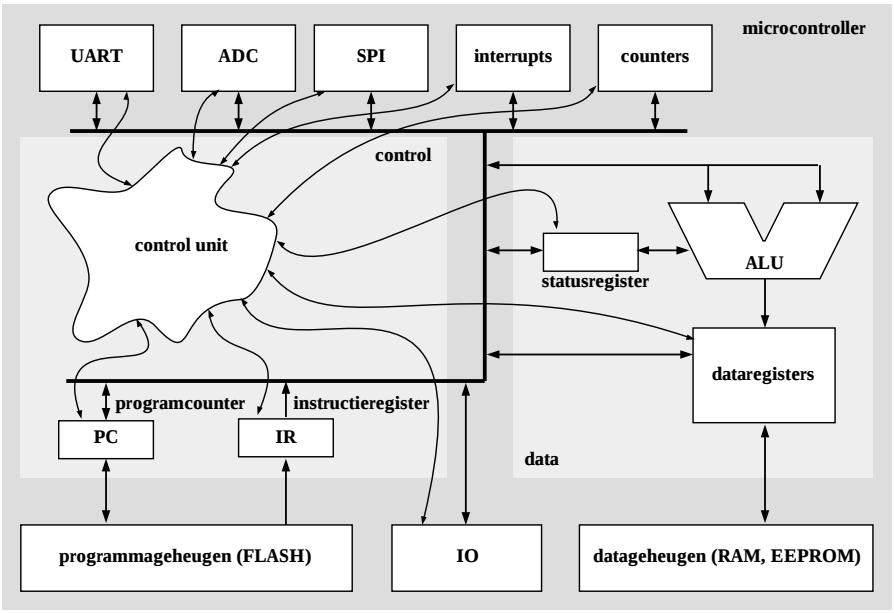
Figuur 1.4 toont de architectuur van een microprocessor. Het hart van de microprocessor is de centrale rekeneenheid — ALU (*Arithmetic Logic Unit*) — die

de rekenkundige (*arithmetic*) en logische (*logic*) bewerkingen uitvoert. Andere onderdelen van de processor zijn algemene dataregisters, een statusregister, een instructieregister, een programmateller (*program counter*) en een stuk besturing (*control unit*), die de processor bestuurt.

Bij een pc-applicatie wordt het hele programma vanaf de harde schijf in RAM geladen. Er wordt in het RAM ook ruimte gemaakt voor de variabelen. Het programmageheugen en het datageheugen zijn dan dus beide RAM. Moderne pc's hebben overigens vaak processoren met een intern RAM, de zogenoemde cache, om de snelheid te verbeteren.

De rekeneenheid kan eenvoudige rekenkundige en logische bewerking uitvoeren op de getallen die in de registers staan. De getallen, die voor een berekening nodig zijn, worden uit het datageheugen gehaald en het resultaat wordt naar dit geheugen weggeschreven. Bij de berekeningen kan de ALU bepaalde bits in het statusregister lezen en zetten, bijvoorbeeld of de berekening overflow geeft. De processor voert instructies (opdrachten) uit die in het programmageheugen staan. Elke keer als een instructie uitgevoerd is, haalt de processor de volgende instructie uit het programmageheugen en zet deze in het instructieregister. De enen en nullen van het instructieregister zijn opdrachten aan de besturing om de juiste handelingen uit te voeren. In de *program counter* staat steeds het adres van de volgende instructie. Het programmageheugen en het geheugen, waar de data worden bewaard, zitten niet in de processor, maar zijn externe geheugens. Gegevens kunnen uit alle andere registers van en naar het dataregister worden geschreven. Ook kunnen er via de IO (*input/output*) gegevens naar binnen of naar buiten worden gebracht.

RAM staat voor *Random Access Memory*.




**Figuur 1.5** : **De architectuur van een microcontroller.** De microcontroller bevat een intern programmageheugen en interne datageheugens. Daarnaast zijn er een groot aantal speciale blokken met extra mogelijkheden, zoals een of meer UART's, ADC's, een SPI, een $I^2C$-interface, interrupts en tellers.

Een microcontroller is een microprocessor met het programmageheugen en de andere datageheugens in één behuizing. Verder heeft een microcontroller een breed scala aan extra in- en uitvoermogelijkheden en andere faciliteiten.

Figuur 1.5 toont de architectuur van een microcontroller. Voorbeelden van extra IO's zijn een UART (*Universal Asynchronous Receiver and Transmitter*), een ADC (*Analog-to-Digital Converter*), een SPI (*Serial Peripheral Interface*). Andere faciliteiten zijn bijvoorbeeld interrupts en tellers (*counters/timers*).

## 1.3 Geheugens en geheugenstructuur

EEPROM staat voor *Electrical Erasable Programmable Read Only Memory* en kan elektrisch worden gewist.

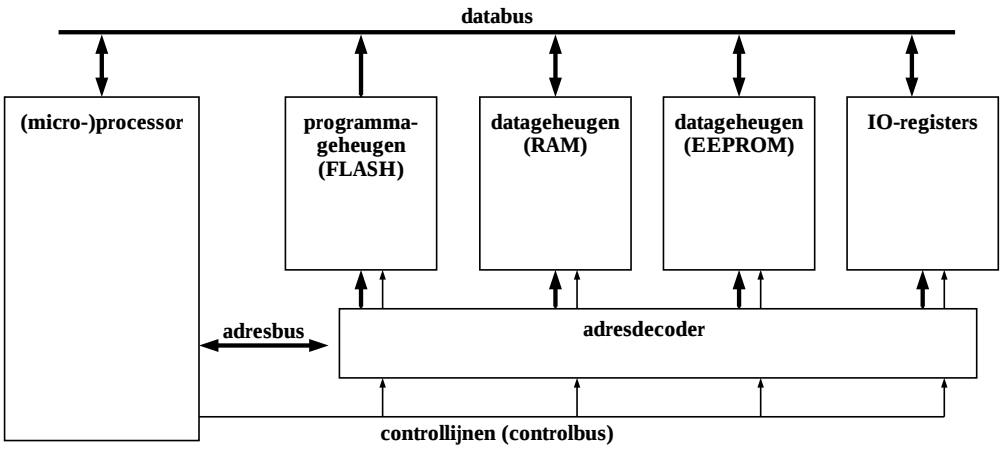
ROM staat voor *Read Only Memory*. Dat is geheugen dat alleen uitgelezen kan worden. De ontwikkelaar kan daar niet schrijven. De data worden in de chipfabriek er ingezet. Een PROM (*Programmable Read Only Memory* is een ROM die de productontwikkelaar met een programmer eenmalig kan programmeren. Een EPROM (*Erasable Programmable Read Only Memory* is een PROM die de ontwikkelaar kan met behulp van uv-licht kan wissen. Deze chips hebben daarvoor een venster en zijn daarom goed herkenbaar.

Het programmageheugen van een microcontroller wordt meestal uitgevoerd in flash en er zijn meestal twee soorten datageheugens: een stuk RAM en een stuk EEPROM. Vroeger — voor dat de flashtechnologie beschikbaar was — hadden microcontrollers een programmageheugen in EEPROM of EPROM. Flash is een verbeterde versie van EEPROM: de toegangssnelheid is hoger, het programmeren gaat sneller. Moderne ROM-geheugens zijn meestal flashgeheugens. Andere verschillen met EEPROM en flash zijn, dat flash minder vaak te herprogrammeren is en dat er grote datablokken tegelijkertijd worden geschreven.
Het datageheugen bestaat in principe uit RAM. Dat is geheugen dat eenvoudig toegankelijk is en dus snel te lezen en te schrijven is. Alleen is RAM altijd vluchtig (*volatile*). Dat betekent dat als de spanning wegvalt alle informatie verdwenen is. Daarom heeft het geen zin om RAM als programmageheugen te gebruiken. Om dezelfde reden heeft een microcontroller ook altijd een stuk EEPROM als datageheugen. Dit geheugen is bestemd voor variabelen en andere gegevens, die bewaard moeten blijven als de spanning uitvalt.

**Tabel 1.3 : De afmeting van de geheugens bij een 8-bits Microcontroller.**

| geheugen | doel | PIC18F2520 | ATmega32 |
|---|---|---|---|
| flash | programma | 32 kB | 32 kB |
| RAM | vluchtige data | 1536 bytes | 2048 bytes |
| EEPROM | niet-vluchtige data | 256 bytes | 1024 bytes |

De afmetingen van geheugens bij een microcontroller zijn direct gerelateerd aan de functionaliteit. Het flash is het grootst omdat in dit geheugen het programma staat. De hoeveelheid instructies in een programma zal veel groter zijn dan de hoeveelheid variabelen die gebruikt worden. Voor de meeste datagegevens van het programma zal spanningsuitval geen probleem zijn. Daarom is in een microcontroller het RAM groter dan het EEPROM. Tabel 1.3 toont de grootte van de geheugens bij twee populaire 8-bit microcontrollers.

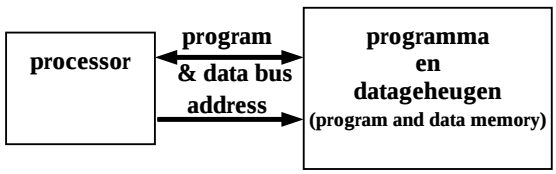



**Figuur 1.6 : Een algemeen schema voor de adressering.** Er zijn diverse geheugens en registers die geadresseerd worden via een adresbus. De adresdecoder zet het adres om naar een selectie van het juiste geheugendeel. De gegevens gaan via de databus van en naar de processor. De geheugens en de adresdecoder worden via controllijnen, de controlbus, aangestuurd.

Geheugens, en ook alle registers, moeten een adres hebben. Elke geheugenplaats heeft in feite een nummer. Net zo als alle huizen een adres — land, stad, straat en huisnummer — hebben, hebben ook alle geheugenplaatsen een adres (*address*). Bij microcontrollers en microprocessoren is dat gewoon een getal. Het maakt niet uit of het een byte uit een statusregister is of dat het een byte uit een programma- of datageheugen is. Alle geheugenplaatsen hebben een uniek adres.
In de figuur 1.6 staat een algemeen schema voor de adressering. De processor zet een adres op de adresbus. De adresdecoder zorgt ervoor dat het betreffende geheugendeel toegang krijgt tot de databus. De controllijnen uit controlbus geven aan of er gegevens op of van de databus gezet of gehaald moeten worden.
Belangrijk is te weten dat de adressering van alle geheugenplaatsen op dezelfde manier gaat.

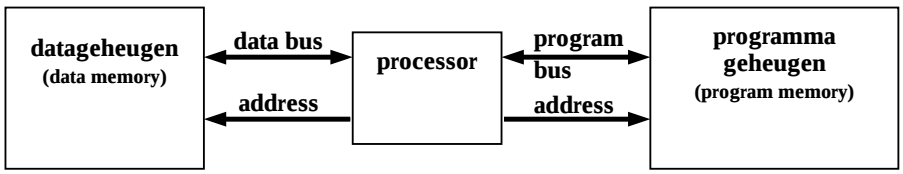
## 1.4 Harvard-architectuur

Er bestaan twee fundamentele architecturen voor een computer: de Princeton- of Von Neumann-architectuur en de Harvard-architectuur. De Von Neumann-architectuur staat in figuur 1.7 en is bedacht door John von Neumann. Deze architectuur heeft een gecombineerde programma- en databus. De Von Neumann-architectuur heeft door de gecombineerde bus een fundamenteel probleem in zich: de zogenoemde Von Neumann-bottleneck. Toch wordt deze structuur — of varianten hierop — nog steeds toegepast.




**Figuur 1.7 : De Von Neumann-architectuur met een gecombineerde programma- en databus.**

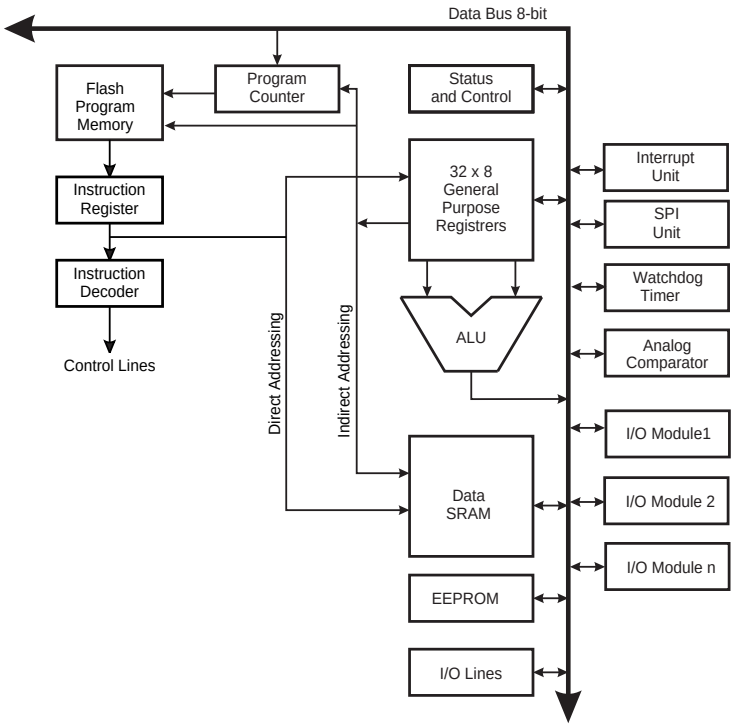
De Harvard-architectuur is getekend in figuur 1.8 en is ontwikkeld op de universiteit van Harvard. Deze architectuur heeft een aparte programmabus en aparte databus.




**Figuur 1.8 : De Harvard-architectuur met gescheiden programma- en databussen.**

Een voorbeeld van *pipelining* is de was. Als de eerste was uit de droger komt om gestreken te worden, kan de tweede was in de droger en kan de derde was in de wasmachine. Als de eerste was klaar is, kan de tweede was worden gestreken, de derde in de droger en de vierde in de wasmachine. Zo wordt er continu gewassen, gedroogd en gestreken.

Het grote voordeel van de Harvard-architectuur is dat, terwijl een instructie uitgevoerd wordt, de volgende instructie al opgehaald kan worden. Deze architectuur leent zich voor *pipelining* en dat maakt sneller systemen mogelijk. Bij microcontrollers is de Harvard architectuur heel populair. Figuur 1.9 is een overdruk van het blokschema uit de datasheet van de ATmega32. Het programmageheugen en de datageheugens worden hier apart geadresseerd.




**Figuur 1.9 : De architectuur van de ATmega32.**

## 1.5 RISC en CISC

De instructieset, die bij een processor hoort, is verschillend voor elk type microprocessor en microcontroller. Er zijn twee soorten processoren: de CISC, *Complex Instruction Set Computer*, die vaak gebruikt wordt bij de Von Neumann-architectuur en de RISC, *Reduced Instruction Set Computer*, die meestal toegepast wordt bij de Harvard-architectuur. RISC-processoren zijn eenvoudiger, kleiner en sneller, maar hebben minder mogelijkheden. Microcontrollers hebben bijna altijd een RISC-processor. Tabel 1.4 geeft een overzicht van de verschillende eigenschappen.

**Tabel 1.4 : Vergelijking tussen RISC en CISC.**

| Reduced Instruction Set Computer (RISC) | Complex Instruction Set Computer (CISC) |
|---|---|
| Eenvoudige instructies van één cyclus | Complexe instructies van meer dan 1 cyclus |
| Alleen LOAD en STORE hebben contact met het geheugen | Elke instructie kan contact met het geheugen hebben |
| Hoge mate van pipelining | Geen of weinig pipelining |
| Instructies uitgevoerd door hardware | Instructies geïnterpreteerd door microprogramma |
| Vast formaat voor instructies | Variabel formaat voor instructies |
| Weinig instructies en modes | Veel instructies en modes |
| Complexiteit zit in de compiler | Complexiteit zit in de microcode |
| Meerdere stellen registers | Eén stel registers |

## 1.6 De keuze voor een microcontroller

Er zijn veel microcontrollerfabrikanten en al deze fabrikanten hebben een enorm breed assortiment. De keuze van een microcontroller kan lastig zijn en wordt bepaald door de volgende aspecten:

- de beschikbaarheid;
- de prijs;
- het gebruikersgemak, sommige devices zijn bijvoorbeeld *in system programmable*;
- de kwaliteit en de prijs van de ontwikkelomgeving, zoals compilers, debuggers en programmers;
- de ondersteuning door vrienden, forums, en dergelijk;
- de beschikbaarheid van application notes, voorbeeld ontwerpen en webpagina's van hobbyisten.
- de eigenschappen van het device, zoals bijvoorbeeld: de snelheid, de geheugengrootte, de dissipatie, het aantal IO-pinnen en de aanwezigheid van: sleepmodes, interrupts, UART's, ADC's en tellers;
- de overstapmogelijkheden naar kleiner — dus goedkopere — of naar grotere — dus meer capabele — devices.

In 2008 heeft Atmel de ATmega32 en de ATmega32L vervangen door de ATmega32A. Deze nieuwe component verschilt alleen elektrisch van de ATmega32. De dissipatie is bijvoorbeeld veel lager. Alles in dit boek geldt ook voor de nieuwe ATmega32A.

Dit boek is gebaseerd de ATmega32 van Atmel. De architectuur van deze 8-bits microcontroller wordt aangeduid met AVR en is in 1992 bedacht door twee studenten van het Norwegian Institute of Technology (Norges Tekniske Høgskole). Deze studenten, Alf-Egil Bogen en Vegard Wollan, werken nu bij Atmel Norway.

De naam ATmega32 is in dit boek gehandhaafd, omdat het een wijd verspreid begrip is en de ATmega32A ook zo genoemd wordt.

De redenen, dat er voor een component uit de AVR-serie van Atmel is gekozen, zijn vooral de kwaliteit van de ontwikkelomgeving en de beschikbaarheid van application notes en voorbeeld ontwerpen. AVRstudio van Atmel — in combinatie met de GNU C-Compiler WinAVR — is een gratis ontwikkelomgeving van hoog niveau, die zeer geschikt is voor het ontwikkelen van embedded systemen met de taal C.

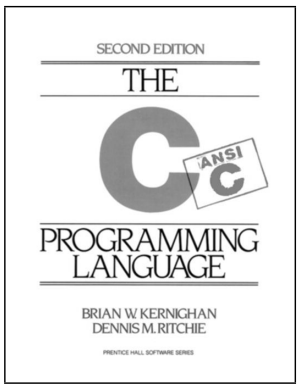
De ATmega16 is pincompatibel met de ATmega32 en heeft dezelfde mogelijkheden. De ATmega8 heeft minder pinnen en minder mogelijkheden. De ATmega64 en de ATmega128 hebben juist meer pinnen en meer mogelijkheden.

Varianten op de ATmega32 zijn: de ATmega32U4 met een USB-aansluiting en de ATmega324P met een zeer lage dissipatie en extra slaapstandmogelijkheden.

# 2

# De taal C

| | |
|---|---|
| Doelstelling | In dit hoofdstuk leer je wat het belang van de taal C is, maak je kennis met een eerste voorbeeld in C en leer je hoe het compilatietraject is opgebouwd. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ Het belang van de taal C.<br>▪ Het voorbeeld 'Hello World' van Kernighan en Ritchie.<br>▪ Het compilatietraject.<br>▪ Een overzicht van de beschikbare (GNU) C-compilers.<br>Voorbeelden van programma's in C zijn:<br>▪ Hello World.<br>▪ Hello World voor Dev-C++.<br>▪ De niet-ANSI versie van Hello World. |




**Figuur 2.1** : **Het beroemde boek van Kernighan en Ritchie:** *The C Programming Language, Second Edition*.

De taal C is een belangrijke programmeertaal. Bij technische toepassingen met computers en microprocessoren is deze taal onmisbaar. De taal C hoort bij Unix. Grote delen van Unix zijn in C geschreven. Ook kunnen bijna alle microprocessoren en microcontrollers in C worden geprogrammeerd. Voor elektrotechnici en technische informatici is deze taal onontbeerlijk. Vele andere talen zijn van C afgeleid of hebben deels dezelfde syntax: C++, C#, PHP, Java, Javascript en Verilog.

C is een algemene, procedurele, sequentiële programmeertaal. C is net als Pascal een procedurele taal. Een programma in C is altijd opgebouwd uit een hoofdroutine met daarbij eventueel meerdere subroutines. De taal C is geen object georiënteerde programmeertaal, zoals Java. Het is een sequentiële taal en kent geen parallellisme, zoals VHDL of Verilog.

C is geen hoog niveau programmeertaal en staat dichtbij de hardware. De taal is niet enorm omvangrijk. C is niet voor een typische toepassing bestemd en kan in veel situaties worden gebruikt. Door de veelzijdigheid en omdat de taal nauwelijks beperkingen kent, is C zeer effectief en snel.

C kent ook nadelen. Juist doordat de taal geen echte beperkingen kent, kan er code worden geschreven die onvoorspelbare resultaten geeft. C kent nauwelijks constructies die de programmeur beschermen tegen het schrijven van niet goed functionerende C. De *type checking* is niet streng of kan eenvoudig worden omzeild.

**Code 2.1 : Hello World.**

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    printf("Hello World!");
6
7    return 0;
8  }
```

In navolging van Kernighan en Ritchie is het gebruikelijk om 'Hello World!' als demonstratievoorbeeld te geven. Op dit internetadres staat voor meer dan 300 verschillende talen het 'Hello World!' voorbeeld: http://www.roesler-ac.de/wolfram/hello.htm

In de praktijk worden belangrijke onderdelen van softwaresystemen, bijvoorbeeld de *kernel* van een simulator, in C geschreven en wordt de grafische gebruikersinterface in Java of Tcl/Tk geschreven.
C is in de jaren zeventig ontwikkeld bij de AT&T Bell Laboratories door Kennith Thomson en Dennis Ritchie. Samen met Brian Kernighan heeft Ritchie het boek *The C Programming Language* geschreven. Begin jaren tachtig heeft het American National Standards Institute (ANSI) een standaardisatie voor C opgesteld. Deze standaard staat bekend als ANSI C en wordt ook aangeduid met ISO C90. Later zijn daar nog aanvullingen op gekomen. Standaard is de compiler meestal ingesteld op ISO C90. In 1988 hebben Brian Kernighan and Dennis Ritchie een nieuwe versie van hun boek uitgebracht: *The C Programming Language, Second Edition*. Deze versie is aangepast voor ANSI C en is ook goed bruikbaar bij de latere versies van C. Inmiddels zijn er honderden boeken over C verschenen. Ook zijn er verschillende Nederlandstalige of in het Nederlands vertaalde boeken verkrijgbaar. Voor iedereen moet er daarom een geschikt boek te vinden zijn.

## 2.1 Hello World

Het eerste voorbeeld uit de tweede druk van het boek van Kernighan en Ritchie staat in code 2.1. Dit is het meest eenvoudige programma en drukt op het scherm de tekst *Hello World!* af, zie ook figuur 2.2.

Met Dev-C++ lijkt het alsof er bij het runnen niets gebeurt. Dev-C++ opent een commandovenster, voert het programma uit en sluit het commandovenster. Dit gaat zo snel dat de gebruiker het niet kan waarnemen. De oplossing is om het programma te laten wachten. Op bladzijde 13 wordt dat uitgelegd.


**Figuur 2.2 : De compilatie van *hello.c* en de uitvoer van het programma *hello*.**

**Uitleg code 2.1 regel 1**
**`#include <stdio.h>`**

De eerste regel in code 2.1 is een zogenoemde preprocessoropdracht. Het bestand `stdio.h` bevat de typen, variabelen, macro's en functiedeclaraties, die nodig zijn om de standaard in- en uitvoer te gebruiken. Dit programma gebruikt de standaard uitvoerfunctie `printf` om tekst naar het scherm te schrijven.
Tussen < en > staat de naam van de headerbestand. De naam van standaard headerbestanden wordt altijd tussen < en > gezet. De compiler zoekt dan naar deze bestanden op de gangbare plaatsen, bijvoorbeeld in `/usr/include`. Eigen headerbe-

standanden worden tussen dubbele aanhalingstekens gezet. De compiler zoekt dan naar de headerbestanden in de werkdirectory bij de andere c-bestanden.

**Regel 3**
```
int main(void)
{
}
```
Functies worden in C soms routines genoemd. Elk C-programma heeft een hoofdroutine, die `main` heet. Dit is de eerste routine, die uitgevoerd wordt, wanneer het programma start. Het type **`int`** geeft aan dat de hoofdroutine een integer retourneert. Tussen de ronde haken staan de (ingangs-)parameters. Het woord **`void`** betekent dat deze routine geen parameters heeft. De statements van de routine staan tussen een accolade openen `{` en een accolade sluiten `}`.

**Regel 5**
```
printf("Hello World!");
```
Met `printf` wordt tekst naar het scherm geschreven. In dit geval is dat de string *Hello World!*. De dubbele aanhalingstekens (`" "`) geven aan dat het een string is. De routine `printf` kent heel veel mogelijkheden en varianten. Op bladzijde 20 bij de uitleg van code 3.1 worden deze besproken.

**Regel 7**
```
return 0;
```
Met **`return`** geeft de functie `main` een resultaat terug. In dit geval wordt een integer `0` geretourneerd. Een retourwaarde is handig om informatie over de status van het programma door te geven aan volgend programma. In (Unix-)scripts wordt hier vaak gebruik van gemaakt.

**Regel 3**
```
int
```
Variabelen van het type **`int`** zijn integers, gehele getallen. Het bereik van **`int`** is — in tegenstelling tot bijvoorbeeld de integers van Java — compiler- en systeem afhankelijk. Meestal worden daar vier bytes (32 bits) voor gebruikt en is het bereik `-2147483648` tot en met `+2147483647`. Bij eenvoudige microcontrollers worden er vaak maar twee bytes gebruikt en is het bereik `-32768` tot `32767`.

**Regel 3**
```
void
```
Het woord **`void`** betekent leeg. Dit wordt gebruikt om expliciet aan te geven dat een routine geen waarde terug geeft of dat de functie geen parameters nodig heeft. In dit geval heeft de hoofdroutine `main` geen parameters nodig.

**Regel 5**
```
;
```
De puntkomma (;) geeft net als bij veel andere programmeertalen het einde van een opdracht (*statement*) aan.

In de kantlijn bij figuur 2.2 is opgemerkt dat er bij Dev-C++ het programma een wachtopdracht moet krijgen, omdat na het uitvoeren het commandovenster direct gesloten wordt. In code 2.2 is op regel 8 voor de return een systeemaanroep geplaatst met de opdracht om te pauzeren. Het programma wacht hier tot de gebruiker op een toets drukt.

**Code 2.2 : Hello World voor Dev-C++.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    printf("Hello World!");
7
8    system("Pause");
9    return 0;
10 }
```

**Uitleg code 2.2 regel 8**
```
int system(char *)
```
De functie `system` voert een *system call* uit. De code `system("dir")` drukt bijvoorbeeld een listing af van de folder van waaruit het programma is gestart. Met `system("Pause")` wacht het programma totdat de gebruiker op een toets gedrukt heeft. *System calls* zijn systeemafhankelijk en kunnen op een ander systeem (*operating system*) niet of anders functioneren.

**Regel 2**
```
#include <stdlib.h>
```

Het headerbestand `stdlib.h` hoort bij de *standard library* en bevat onder andere het prototype van `system()`. Tabel F.2 uit bijlage F geeft een overzicht van de functies uit `stdlib.h`.
Code 2.1 is volgens de ISO C90 standaard opgeschreven. Dit kan nog eenvoudiger worden opgeschreven. Code 2.3 geeft de code van *Hello World!* uit de eerste druk van het boek van Kernighan en Ritchie.

**Code 2.3 : Oorspronkelijke niet-ANSI versie van Hello World.**

```
1  main()
2  {
3    printf("Hello World!");
4  }
```

De routine `printf` is een standaard routine die bij alle C-compilers bekend is. De include-regel is daarom niet per se nodig. Ook mag de **`return`** weggelaten worden. Bij `main` mag het returntype en net als **`void`** in de parameterlijst worden weggelaten.

De GNU C-Compiler `gcc` kent zeer veel opties. In de *manual page* worden al deze opties zeer uitgebreid besproken.

Dit boek gebruikt altijd de GNU89 standaard, dat is de ISO C90 norm met een aantal GNU uitbreidingen. De verouderde schrijfwijze van code 2.3 wordt verder niet meer gebruikt. Bij de voorbeelden worden ook altijd alle include-bestanden genoemd.

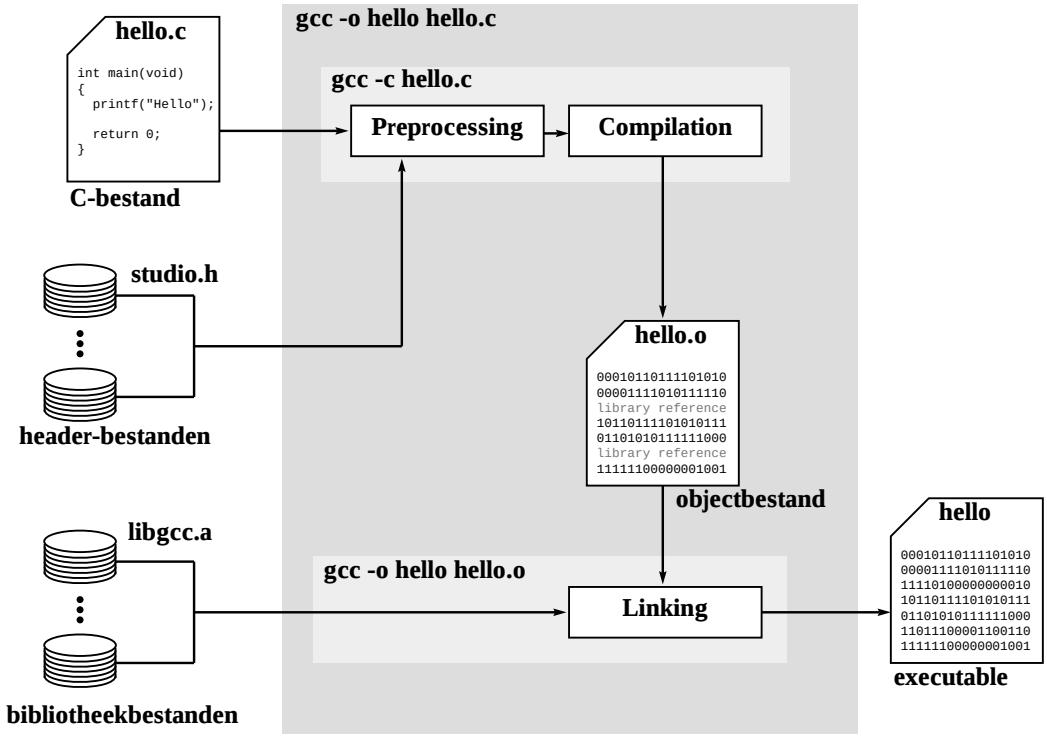
```
/cc/hello
/cc/hello $ gcc -o hello hello.c -Wall
hello.c:2:  warning:  return type defaults to 'int'
hello.c:  In function 'main':
hello.c:3:  warning:  implicit declaration of function 'printf'
hello.c:4:  warning:  control reaches end of non-void function

/cc/hello $ hello
Hello World!
/cc/hello $
```

**Figuur 2.3 : Alle waarschuwingen bij compilatie van code 2.3 en de uitvoer van het programma hello.**

Met de compiler-optie `-Wall` worden bij het compileren alle waarschuwingen afgedrukt op het scherm. Figuur 2.3 toont de waarschuwingen die de compiler geeft bij de verouderde schrijfwijze van code 2.3. De compiler veronderstelt dat het returntype van `main` een **`int`** is, waarschuwt dat er geen retourwaarde is en meldt dat de declaratie van `printf` impliciet is. Dat laatste betekent dat de compiler niet kan checken of de functie op de juiste wijze gebruikt wordt.
Om mogelijke problemen en slordigheden te voorkomen, is het verstandig altijd de optie `-Wall` te gebruiken en de ANSI standaard te volgen. De code is dan wel iets uitgebreider, maar de kans op elementaire fouten is dan veel kleiner.

## 2.2 Het compilatietraject

De compiler vertaalt de C-code in objectcode, voegt bij deze code de objectcode uit standaardbibliotheken en maakt er een uitvoerbaar programma (*executable*) van.
Het hele compilatietraject is te verdelen in drie stappen: de *preprocessing*, de compilatie en het linken. In figuur 2.4 is dit traject getekend. De preprocessor doet een stuk voorbewerking. Het voegt informatie uit de headerbestanden toe aan




**Figuur 2.4 : Het compilatietraject bestaat uit stappen: de preprocessing, de compilatie en het linken.** Met de optie `-c` wordt bij *gcc* alleen de objectcode gemaakt. Als aan de *gcc* de al eerder gecompileerde objectcode (*hello.o*) wordt meegegeven, wordt alleen de linking gedaan.

Objectcode is binaire code die door de *linker* wordt gebruikt om er een uitvoerbaar bestand *executable* van te maken. Objectcode heeft dus niets te maken met object georiënteerd programmeren.

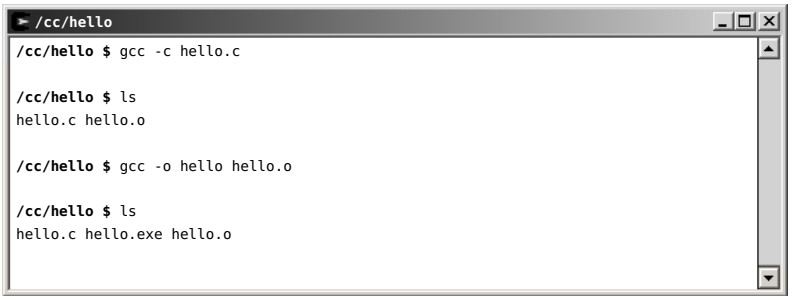
de code die gecompileerd moet worden en voert alle preprocessoropdrachten uit. Dit geheel vertaalt de compiler in objectcode. Deze objectcode bevat alle machinecode die gevormd kan worden uit de broncode. De objectcodes uit de bibliotheken ontbreken nog. De *linker* voegt deze objectcodes uit bibliotheken toe aan de objectcode en maakt er een uitvoerbaar programma van.

Bij complexe programma's wordt de code verdeeld over meerdere c-bestanden. Deze bestanden worden dan apart gecompileerd en later aan elkaar gelinkt. Bij eenvoudige programma's voert men de compilatie en het linken vaak in een keer uit. In figuur 2.4 is dit met de donkergrijze achtergrond aangegeven. De compilatie en het linken kunnen ook apart uitgevoerd worden. Met de compiler-optie `-c` maakt de compiler alleen de objectcode. Door aan de compiler alleen het objectbestand met de objectcode mee te geven, wordt de compilatie overgeslagen en wordt dit objectbestand gelinkt met de bibliotheekbestanden tot een uitvoerbaar programma. In figuur 2.4 zijn deze twee stappen met een lichtgrijze achtergrond aangegeven. Figuur 2.5 toont de twee stappen zoals de programmeur deze uitvoert. Na de compilatie is er een bestand `hello.o` gemaakt en na het linken is er een executable `hello.exe` gemaakt.

Het Unix-commando `ls` geeft een lijst met de bestanden in de huidige folder.

## 2.3 Compilers

Bij de voor- en nadelen van de C-compilers en ontwikkelomgevingen voor C wordt vaak over Unix gesproken. Dat is niet zo vreemd, omdat Unix grotendeels geschreven is in C. C en Unix horen bij elkaar. Bij elke Unix- en Linux-




**Figuur 2.5 : Het compileren van code 2.1 en het linken wordt hier apart uitgevoerd.**

De meeste C++-compilers kunnen ook C compileren en kunnen dus als C-compiler gebruikt worden. Ook de GNU C-Compiler is een C- en C++ Compiler.

distributie zit standaard een C-compiler. Unix- en Linux-gebruikers hebben dus altijd de beschikking over een C-compiler. Windows-gebruikers hebben dat niet. Voor Windows bestaan heel veel C-compilers. Er zijn commerciële pakketten, zoals Microsoft Visual C++, maar ook freeware oplossingen.
Een belangrijke C-compiler, die zowel voor Linux als voor Windows beschikbaar is, is de GNU-compiler. GNU is een project, dat gestart is door Richard Stallman, met als doel een volledig licentievrij besturingssysteem voor computers te maken. De verschillende onderdelen van GNU worden apart aangeboden en worden in allerlei Unix- en Linux-distributies gebruikt. Het meest belangrijke onderdeel uit dit project is de GNU C-Compiler.

GNU staat voor *GNU's Not Unix*. Dit is een recursief acroniem (letterwoord). Recursie wordt in paragraaf17.2 behandeld.

Er zijn verschillende Windows-implementaties van de GNU C-Compiler te vinden. De belangrijkste zijn:

- de Cygwin-omgeving
- MinGW, Minimal GNU for Windows;
- MinGW in combinatie met Msys;
- MinGW in combinatie met een grafische shell;
- Dev-C++ van Bloodshed.

**Cygwin**

Cygwin is een complete Unix-omgeving binnen Windows. Voor gebruikers, die geen *dual-boot*-installatie met Linux willen, is dit wel een interessant alternatief. Dit boek is voorbereid met de `gcc`-compiler uit de Cygwin-omgeving. Alle voorbeelden zijn gemaakt met deze compiler en alle schermafdrukken zijn afdrukken van de Cygwin-omgeving. Een minimale installatie, waar in ieder geval wel `gcc` bij zit, zal ongeveer 50 Mbytes in beslag nemen. De installatie van Cygwin wordt op de webpagina van dit boek beschreven.

`gcc` staat voor GNU's compiler collection en is dus niet alleen een compiler, maar een compleet pakket met compiler en linker.

De gebruikersinterface is een commandovenster met een *prompt* waarachter de compiler en de *executable* worden aangeroepen. Figuur 2.2 geeft een voorbeeld.

Een *command shell window* is een venster waarin opdrachten (commando's) worden gegeven. Deze worden achter een zogenoemde prompt ingetoetst. Het standaard commandovenster van Windows wordt, ondanks dat MSDOS al jaren niet meer bestaat, nog vaak aangeduid als DOS-venster of DOS-box.

**MinGW, Minimal GNU for Windows**

MinGW, Minimal GNU for Windows is een goed alternatief. Oorspronkelijk was het een klein pakket van een tiental Mbytes. Tegenwoordig is het flink gegroeid en is het dus niet echt minimaal meer. MinGW bevat alleen een compiler met een aantal voor het hele compilatietraject noodzakelijke programma's, zoals een linker. De compiler kan vanuit het standaard commandovenster van Windows (*command shell window*) worden aangeroepen. Om dit kunnen doen, moet eerst de omgevingsvariabele PATH worden aangepast. Op de webpagina van dit boek staat hoe dat moet.

**MinGW in combinatie met Msys**

Veel simulatoren en andere programma's voor het ontwerpen en testen van elektronische of microprocessorsystemen gebruiken een eigen versie van `gcc`. Daarom is het veiliger te werken met een aparte omgeving, waardoor er zekerheid is dat de MinGW GNU compiler en de daarbij horende bibliotheken gebruikt worden. Msys is een Unix-achtig commandovenster dat speciaal gemaakt is voor MinGW. Een installatieprocedure voor MinGW/Msys staat ook op de webpagina van dit boek. Figuur 2.6 toont de gebruikersinterface. Deze interface en de manier van werken lijkt op de Cygwin-omgeving.


**Figuur 2.6 : De Unix-achtige shell van Msys/MinGW.**

**Unix-teksteditor vi**

Zowel bij Cygwin als MinGW zit de standaard Unix-teksteditor `vi`. Bij Cygwin zit ook de editor `emacs`. Op de webpagina van dit boek staat een handleiding voor `vi`. Het nadeel van `emacs` en `vi` is dat deze editors commandogestuurd zijn en dat het moeite kost om er mee te leren werken. In plaats van `vi` kan ook een platte teksteditor van Windows gebruikt worden, zoals `notepad`. Deze editor is erg primitief. De webpagina van dit boek geeft een aantal alternatieven.

Juist omdat `vi` op alle Unix-achtige systemen zit, is het is wel zeer nuttig om met `vi` te kunnen werken.

**Dev-C++ van Bloodshed**

Dev-C++ van Bloodshed is een GNU C-Compiler voor Windows met een grafische interface. Windows-gebruikers vinden dat vaak prettig. Toch is Dev-C++ voor een nieuwe gebruiker ook lastig. Er moeten aan de C-code extra opdrachten worden toegevoegd om de applicatie te kunnen gebruiken. In figuur 2.7 staat de grafische interface van Dev-C++.

De ontwikkeling van Dev-C++ is gestopt. de laatste uitgave is van februari 2005. De variant wxDev-C++, met een speciale wxWidget-bibliotheek voor grafische gebruikersinterfaces, wordt wel verder ontwikkeld en onderhouden.

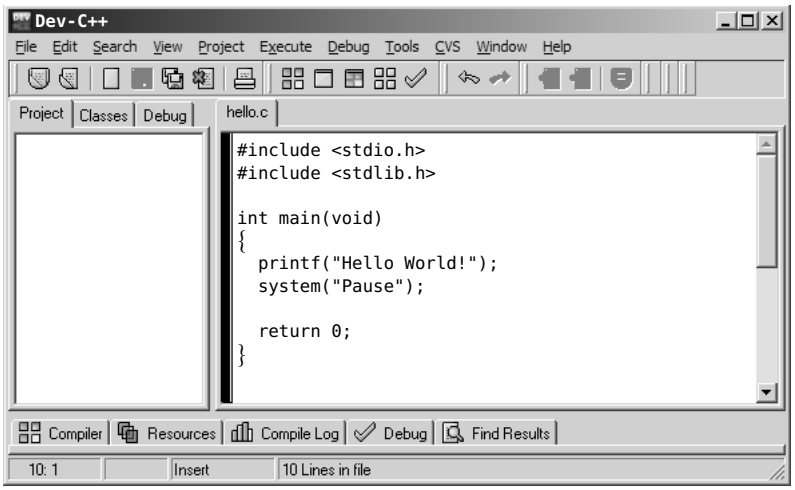
**MinGW in combinatie met Code::Blocks**

Code::Blocks is een *open source* grafische ontwikkelomgeving. Figuur 2.8 toont de interface. De omgeving is geschikt voor vele C/C++-compilers, dus ook voor de GNU-compilers. Op de site van Code::Blocks is een versie met de MinGW-compiler beschikbaar. Code::Blocks heeft geen — zoals Dev-C++ — extra opdrachten nodig om een *console*-applicatie te maken. Net als bij andere grafische interfaces is het gebruik van *command line* parameters omslachtig.

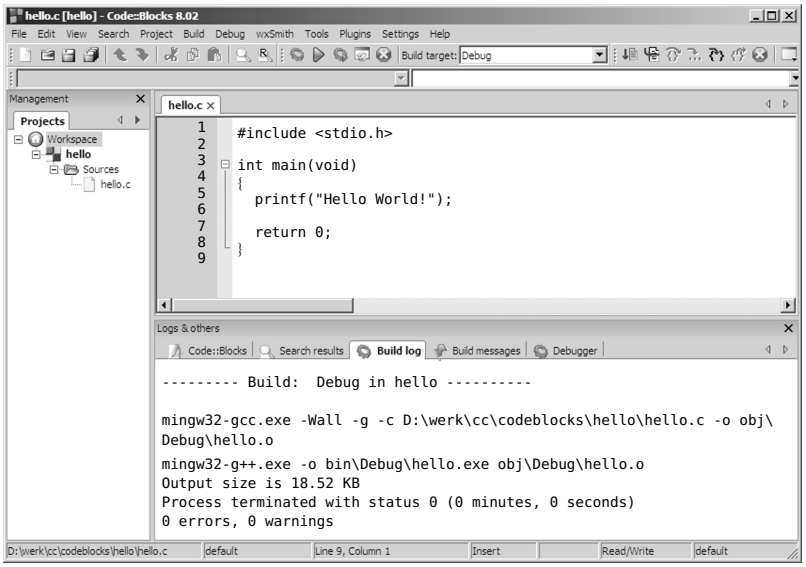
**Native compilers versus crosscompilers**

De besproken compilers zijn *native compilers*; ze zijn bedoeld om C-programma's te maken binnen een Windows-omgeving. Met de GNU C-Compiler kunnen ook applicaties voor andere systemen worden ontwikkeld: bijvoorbeeld voor een ARM-processor. Het maken van een programma op een bepaald systeem (Windows) voor een andere systeem (ARM-processor) heet crosscompilen. Voor het maken van de programma's voor de Atmel AVR devices wordt een aangepaste GNU C-Compiler gebruikt. Deze crosscompiler, WinAVR, wordt apart geïnstalleerd en werkt nauw samen met AVRstudio, de debugger van Atmel.

ARM staat *Advanced RISC Machine*. De architectuur van deze processor is ontwikkeld door ARM Limited en wordt onder licentie door veel processorfabrikanten gebruikt in hun ARM-processoren.




**Figuur 2.7 : De grafische interface van Dev-C++.**




**Figuur 2.8 : De grafische interface van Code::Blocks.**

Er zijn nog vele andere GNU C/C++-compilers en omgevingen voor Windows, zoals: DJGPP, Vide, en Visual-MinGW. Bovendien zijn er ook vele andere gratis oplossingen, bijvoorbeeld: lcc-win32 en Borland-C.

Dit boek gebruikt tot en met hoofdstuk 9 en in de hoofdstukken 14 tot en met 17 Cygwin, MinGW of Dev-C++ voor het maken van Windows-toepassingen. In de overige hoofdstukken wordt AVRstudio in combinatie met WinAVR gebruikt om applicaties voor de Atmel AVR devices te maken.

**Grafische interfaces versus commandovensters**

Er bestaat ook een C/C++-versie van de zeer populaire Java-ontwikkelomgeving Eclipse.

Cygwin, MinGW/Msys, MinGW/Code::Blocks en Dev-C++ zijn vier goede alternatieven om de code van dit boek mee te bestuderen. Elke ontwikkelomgeving heeft zijn voor- en nadelen. Alle voorbeelden in dit boek zijn getest met de GNU-compiler `gcc` versie 3.4.4 uit de Cygwin-omgeving. Van de besproken compilers staat Cygwin het dichtst bij Unix. Voorbeelden met Unix-achtige constructies of toepassingen, zullen bij andere ontwikkelomgevingen niet gedemonstreerd kunnen worden of aangepast moeten worden.

# 3

# Declaraties

| | |
|---|---|
| Doelstelling | In dit hoofdstuk leer je wat een declaratie is, hoe je in C variabelen declareert en zie je wat het effect is van een onjuiste declaratie. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De declaratie van **`int`**, **`char`**, **`float`** en **`double`**.<br>▪ De declaratie van strings.<br>▪ Het geheugengebruik bij strings.<br>▪ Geformatteerd afdrukken met `printf`.<br>▪ Foutmeldingen van de compiler bij verkeerde declaratie.<br>▪ Runtime errors bij verkeerde declaratie.<br>▪ Overschrijven van geheugenruimte bij verkeerde declaratie.<br>Het declareren van variabelen wordt gedemonstreerd met deze voorbeelden:<br>▪ Het afdrukken van naam en leeftijd.<br>▪ Het afdrukken van naam en leeftijd met aparte toewijzingen.<br>▪ Het afdrukken van een te lange naam.<br>▪ Het afdrukken van naam en leeftijd met globale variabelen.<br>▪ Het afdrukken van een te lange naam bij globale variabelen. |

Een C-programma bestaat uit een verzameling functies, variabelen en typedefinities. De C-compiler leest de source code sequentieel van boven naar beneden. Een declaratie van een type, variabele en functie zorgt ervoor dat de compiler de naam kent. Als de compiler een bekende naam tegenkomt, weet de compiler hoe deze toegepast moet worden. Komt de compiler een onbekende naam tegen dan kan dat leiden tot een foutmelding of een waarschuwing en sommige gevallen, bijvoorbeeld bij onbekende functienamen, maakt de compiler een eigen veronderstelling.

Het programma van code 3.1 drukt een naam en een leeftijd af. Er zijn twee variabelen `name` en `age` gedeclareerd, die direct worden geïnitialiseerd met de waarden `"John"` en `29`.

De structuur van dit programma lijkt op die van code 2.1. Alleen de hoofdroutine `main` is nu uitgebreider.

**Code 3.1 : Afdrukken van naam en leeftijd.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    char name[]="John";
 6    int  age=29;
 7
 8    printf("My name is %s and my age is %d", name, age);
 9
10    return 0;
11  }
```

**Uitleg code 3.1 regel 5**
**char**

Het type **char** is een acht bits getal, dat een karakter representeert. In het algemeen zal het de ASCII-waarde zijn. De waarde 65 is bijvoorbeeld het karakter `'A'` en 97 is een `'a'`.

**Regel 5**
*string*
"John"

C kent standaard geen type string. Toch wordt het begrip string veel gebruikt. In C is een string een rij karakters die afgesloten wordt met een speciaal karakter. Dit speciale karakter is de ASCII-waarde nul. Alle acht bits van dit karakter zijn nul en het wordt aangegeven met `'\0'`.

| J | o | h | n | \0 |
|---|---|---|---|---|

**Figuur 3.1 : De string "John" is een array van char afgesloten met een '\0'.**

**Regel 5**
*array*
**char** name[]

Met **char** `name[]` wordt aangegeven dat de variabele `name` een array is. Een array is een verzameling data van dezelfde soort, bijvoorbeeld een rij getallen of karakters. In dit geval is `name` een array van **char**. Als er tussen de rechte haken niets staat, moet er een initiële waarde worden opgegeven. Anders weet de compiler niet hoe lang de array is en hoeveel geheugenruimte er gealloceerd moet worden.

**Regel 8**
**int** printf(**char** *f, ...)
*format specifiers*

De functie `printf` schrijft geformatteerde tekst naar het scherm. Met zogenoemde *format specifiers* worden de waarden van variabelen in de tekst ingevoegd. Hier zijn een aantal voorbeelden:

```
printf("Hello World!");                    // Hello World!
printf("%s %s", "Hello", "World!");        // Hello World!
printf("%s is %d", "John", 29);            // John is 29
printf("char %c is ASCII: %d", 'a', 'a');  // char a is ASCII 97
printf("%d + %d = %d", 4 , 5, 9);          // 4 + 5 = 9
printf("This is %d decimaal", 24+25);      // This is 49 decimaal
printf("This is %x hexadecimaal", 49);     // This is 31 hexadecimaal
printf("This is %o octaal", 49);           // This is 61 octaal
printf("This is %f", 49.2);                // This is 49.200000
printf("This is %e", 49.2);                // This is 4.920000+01
printf("This is %g", 49.2);                // This is 49.2
```

`printf` heeft een variabel aantal argumenten. Het eerste argument van `printf` is de *format string*. De volgende argumenten zijn de variabelen die in de formatregel ingevuld moeten worden. Het eerstvolgende (tweede) argument komt op de plaats van de eerste *format specifier*; het derde argument komt op de plaats van de tweede *specifier* enzovoorts. Bij de uitleg van code 9.3 staat een uitgebreidere uitleg over de *format specifiers*.

Variabelen krijgen bij de declaratie meestal geen waarde. De declaratie maakt dan alleen de naam en het type bekend. De compiler kent dan de naam en weet hoeveel geheugenruimte er nodig is. In de loop van het programma worden aan de variabelen waarden toegekend. In code 3.2 worden op regel 6 en 7 de variabelen `age` en `name` gedeclareerd en op regel 9 en 10 krijgen deze variabelen hun waarden.

**Code 3.2 : Afdrukken van naam en leeftijd met aparte toewijzingen.**

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6    char name[16];
7    int  age;
8
9    age=29;
10   strcpy(name,"John");
11   printf("My name is %s and my age is %d", name, age);
12
13   return 0;
14 }
```

**Uitleg code 3.2 regel 6**
**`char name[16]`**

Omdat de variabele `name` niet geïnitialiseerd wordt, moet de grootte van de array gedefinieerd worden. In dit voorbeeld zijn dat zestien karakters. Dat is inclusief het *null character* `'\0'` dat het einde van de string aangeeft. De variabele `name` mag maximaal vijftien gewone karakters bevatten.

**Regel 9**
**=**

Variabelen van de typen **`int`**, **`char`**, **`float`** en **`double`** krijgen met de toekenningsoperator = een waarde. Hier zijn een aantal voorbeelden:

```
int    i;
char   c;
float  f;
double d;
double p;

i = 2007;
c = 'b';
f = 4.2;
d = 4.2e+03;
p = 3.14;
```

**Regel 10**
**`strcpy(char *d, char *s)`**

Om aan een stringvariabele een waarde toe te kennen, moet de functie `strcpy` worden gebruikt. De toekenningsoperator = mag hier niet worden gebruikt. Dit is dus fout:

```
name = "John";
```

Het levert een foutmelding op dat de toekenning onverenigbare typen bevat:

```
mname.c:9: error: incompatible types in assignment
```

Om dit probleem goed te verklaren moet het begrip pointer bekend zijn. Pointers worden in paragraaf 15 besproken en daar komt dit soort problemen met pointers uitgebreid aan de orde.

**Regel 2**
`#include <string.h>`

Om `strcpy` te gebruiken wordt het includebestand `string.h` toegevoegd. Dit bestand bevat de prototypen van een groot aantal stringbewerkingen. In hoofdstuk 16 worden de strings en de stringfuncties besproken. Dit zijn vier belangrijke bewerkingen:

```
strcpy(s1,s2)    // kopieert de inhoud van s2 naar s1
strcat(s1,s2)    // voegt de inhoud van s2 toe aan s1
strlen(s)        // Geeft de lengte van string s
strcmp(s1,s2)    // vergelijkt de inhoud van s1 met die van s2
```

In hoofdstuk 16 komen de stringfuncties uitgebreider aan bod.

**Code 3.3 : Code 3.2 met een te lange naam.**

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  int main(void)
 5  {
 6     char name[16];
 7     int  age;
 8
 9     age=29;
10     strcpy(name,"John with the very long christian name");
11     printf("My name is %s and my age is %d", name, age);
12
13     return 0;
14  }
```

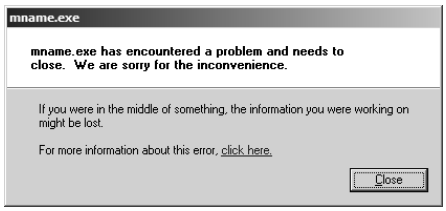
In code 3.2 is voor de variabele `name` geheugenruimte voor slechts zestien karakters gereserveerd. Wanneer de programmeur naar deze variabele een string kopieert die langer is dan zestien karakters, zoals in code 3.3 gebeurt, kunnen er fouten optreden. Wat er precies gebeurt, is vooraf niet altijd te voorspellen. Er zijn drie mogelijkheden:

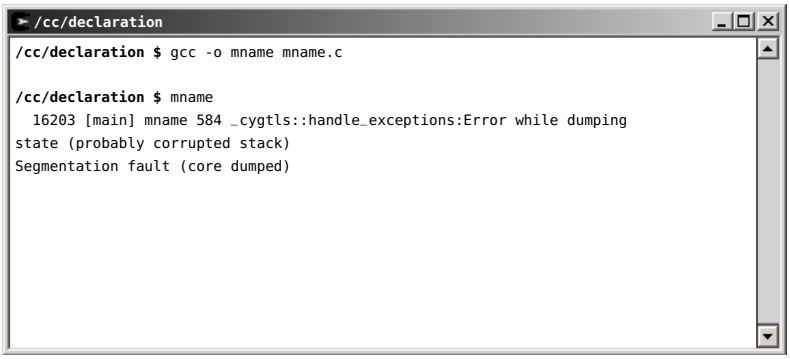
- Het programma werkt en drukt de tekst af op het scherm.
- Het programma doet niets.
- Het programma crasht en geeft een zogenoemde *runtime error*. In dit geval geeft code 3.3 een foutmelding, die in figuur 3.2 getoond wordt. Bovendien wordt er dan een bestand `mname.exe.stackdump` gemaakt met informatie over de actuele situatie van het geheugen toen de executable `mname.exe` crashte.

Met C is het niet moeilijk om een programma te schrijven dat runtimefouten oplevert. De C-programmeur is er verantwoordelijk voor dat deze fouten niet voorkomen.

De variabelen zijn in de voorgaande voorbeelden gedeclareerd in de functie `main`. In plaats daarvan hadden deze ook globaal gedeclareerd kunnen worden, zoals in code 3.4. Globale variabelen worden anders behandeld dan lokale variabelen. Lokale variabelen worden op de *stack* geplaatst en globale variabelen op de *heap*. In dit geval werkt het programma — toevallig — goed, ondanks dat de array te klein is.

Het blauwe scherm met witte karakters bij Windows is ook het gevolg van runtime errors. De uitvoer van figuur 3.2 is daarmee vergelijkbaar. Tegenwoordig vangt Windows deze fouten wat netter af. MinGW en Dev-C++ geven:



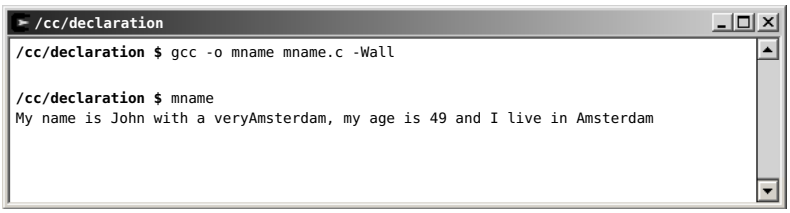




**Figuur 3.2 : Runtime error bij het uitvoeren programma code 3.3.**

**Code 3.4 : Afdrukken van naam en leeftijd met globale variabelen.**

```
1  #include <stdio.h>
2  #include <string.h>
3
4  char name[16];
5  int  age;
6
7  int main(void)
8  {
9     age=49;
10    strcpy(name,"John with the very long christian name");
11    printf("My name is %s and my age is %d", name, age);
12
13    return 0;
14 }
```

De programmeur zou tevreden kunnen zijn; het programma werkt immers naar wens. Toch moet hij proberen dit soort fouten te voorkomen. Op een later moment bij een aanpassing of uitbreiding van het programma kunnen er problemen ontstaan. In code 3.5 is een variabele `place` voor de woonplaats toegevoegd.
De uitvoer van het programma van code 3.5 staat in figuur 3.3. Er treedt nu geen *runtime error* op, maar de uitvoer levert wel een vreemd resultaat op.

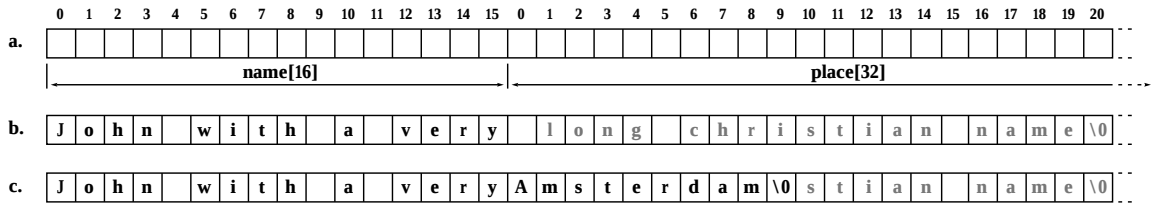



**Figuur 3.3 : De uitvoer van het programma van code 3.5.**

Figuur 3.4 laat zien dat in het geheugen de string `place` gedeeltelijk over string `name` komt te liggen en dat string `name` een combinatie wordt van het begin van string `name` en van string `place`.
Bij een taal als Java is dit soort fouten niet mogelijk. Bij C moet de programmeur

**Code 3.5 : Voorbeeld van effect van te grote string.**

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  char name[16];
 5  char place[32];
 6  int  age;
 7
 8  int main(void)
 9  {
10     age=49;
11     strcpy(name,"John with a very long christian name");
12     strcpy(place,"Amsterdam");
13     printf("My name is %s, my age is %d and I live in %s",
14             name, age, place);
15
16     return 0;
17  }
```




**Figuur 3.4 : Het gebruik van het geheugen bij code 3.5.**
a. De declaraties van regel 4 en 5 reserveren de benodigde geheugenruimte.
b. Regel 11 kopieert de lange string naar de ruimte, die gereserveerd is voor de variabele `name`. Omdat de string te lang is, overschrijft deze ook een deel van `place`.
c. Regel 12 kopieert `Amsterdam` naar de locatie van `place`. String `name` loopt van het begin van `name` tot de eerstvolgende *end-of-string* en luidt daarom `John with a veryAmsterdam`.

er zelf op letten dat er niet per ongeluk buiten een array of string wordt geschreven. De C-programmeur moet de variabelen zorgvuldig declareren en zich ook aan deze declaraties houden. De grootste valkuil is dat een programma soms goed lijkt te werken, terwijl er toch dit soort fouten in de code zit. Een kleine wijziging — in een totaal ander deel van het programma — kan dan plotseling toch *runtime errors* opleveren of aanleiding zijn van andere vreemde effecten.

# 4

# Functies

**Doelstelling**

In dit hoofdstuk leer je wat een functie is en hoe je in C een functie declareert en hoe je een functie aanroept.

**Onderwerpen**

De behandelde onderwerpen zijn:

- De opbouw van een functie: functieheader, functiebody, retourwaarde en parameterlijst.
- Het prototype van een functie.
- De bestandsorganisatie bij gebruik van meerdere c-bestanden.
- Het maken en aanroepen van een eigen include-bestand.
- Formele en actuele parameters.
- De scope van functies en variabelen.
- De *call by reference*-methode en de adresoperator `&`.

De voorbeelden met functies zijn:

- Het afdrukken van leeftijd met een functie `print_age`.
- Het afdrukken van leeftijd met een functie `print_age` met headerbestand `age.h`.
- Een functie `volume` met formele en actuele parameters.
- Een bestand met de verschillende scopes voor zeven variabelen `age`.
- Een functie `get_age1` met retourwaarde en een functie `get_age2`, die de leeftijd teruggeeft via de parameterlijst volgens de *call by reference*-methode.

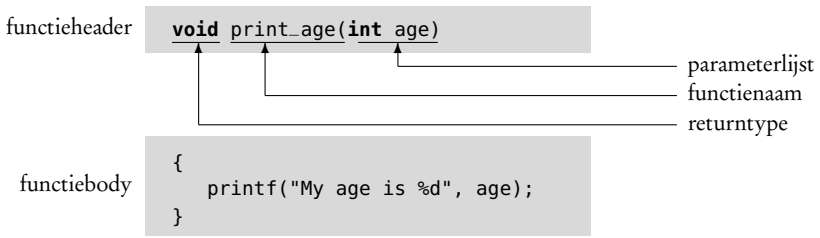
Elke taal kent een of meer mogelijkheden om een stuk code te verdelen in kleinere, hanteerbare stukken. Bij Java zijn dat klassen en methoden, bij Pascal zijn dat procedures en bij C zijn dat functies en deze worden ook wel routines genoemd. Elk C-programma bevat in ieder geval een functie met de naam `main`. Dat is de hoofdroutine waarmee het programma begint. Daarnaast kent C veel bibliotheekfuncties, die gebruikt kunnen worden. De programmeur kan ook zelf functies toevoegen. In code 4.1 is een functie `print_age` gedefinieerd, die een leeftijd afdrukt.

Een functiedefinitie, zie figuur 4.1, bestaat uit een functie-*header* en een functie-*body*. De functieheader heeft returntype, een functienaam en tussen de ronde haken eventueel een lijst met parameters. De functiebody begint met een accolade openen `{` en eindigt met een accolade sluiten `}`. Tussen deze twee accolades staan de lokale declaraties en de statements van de functie.

**Code 4.1 : Afdrukken leeftijd met aparte afdrukfunctie.**

```
 1  #include <stdio.h>
 2  void print_age(int age)
 3  {
 4    printf("My age is %d", age);
 5  }
 6
 7  int  age=24;
 8
 9  int main(void)
10  {
11    print_age(age);
12    return 0;
13  }
```

Een functie mag pas gebruikt worden als de naam, het returntype en de typen van de ingangsparameters van de functie bij de compiler bekend zijn. In code 4.1 staat de functiedefinitie voor de functie `main`; dus voor regel 11 waar de functie aangeroepen wordt. De compiler herkent de functie en weet dat er een ingangsparameter is van het type **`int`** en dat de retourwaarde leeg (**`void`**) is.




**Figuur 4.1 : Een functiedeclaratie bestaat uit een functieheader en een functiebody.** De functieheader bestaat uit een returntype, een functienaam en tussen ronde haken een parameterlijst. De functiebody begint met een `{` en eindigt met een `}`.

**Code 4.2 : Afdrukken leeftijd met prototype.**

```
 1  #include <stdio.h>
 2  void print_age(int age);
 3  int  age=24;
 4
 5  int main(void)
 6  {
 7    print_age(age);
 8    return 0;
 9  }
10
11  void print_age(int age)
12  {
13    printf("My age is %d", age);
14  }
```

In code 4.2 komt de functiedeclaratie na `main`. Bij de aanroep op regel 7 zou de functie dan onbekend zijn en een waarschuwing geven. Met regel 2 wordt dat voorkomen. Dit is een zogenoemde *prototype* van de functie. Het is de header van

de functie afgesloten met een puntkomma (;). Alle gegevens die de compiler van de functie moet weten, zijn dan bekend: de naam, het type van de retourwaarde en de typen van de eventuele ingangsparameters. De body van de functie hoeft de compiler niet te weten om een functieaanroep te kunnen verwerken.

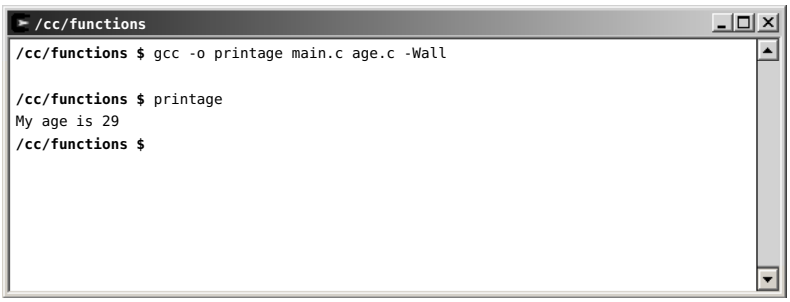
**Code 4.3 : Code bestand main.c.**

```
void print_age(int age);
int  age=29;

int main(void)
{
  print_age(age);
  return 0;
}
```

**Code 4.4 : Code bestand age.c.**

```
#include <stdio.h>

void print_age(int age)
{
  printf("Your age is %d", age);
}
```

De functie `print_age` hoeft ook niet in hetzelfde bestand als `main` te staan. In code 4.3 staat de inhoud van een bestand `main.c`. Op regel 1 staat het prototype van een functie `print_age`. De definitie van `print_age` staat in bestand `age.c`, zie code 4.4. Deze bestanden mogen apart gecompileerd worden tot objectbestanden en daarna samen gelinkt tot een uitvoerbaar programma. Ze kunnen alle twee direct gecompileerd en gelinkt worden, zoals figuur 4.2 is gedaan.




**Figuur 4.2 : De compilatie en de uitvoer van de bestanden main.c en age.c.**

Bij een prototype gaat het niet om de namen van de parameters. Het gaat alleen om de typen. Dit zijn drie geschikte prototypen voor de functie `print_age`:

```
void print_age(int age);
void print_age(int a);
void print_age(int);
```

Het is gebruikelijk om dezelfde namen te gebruiken voor de parameters bij het prototype en bij de header bij de functiedefinitie. Het is dan duidelijk over welke parameters het gaat. Bovendien is het prototype dan gewoon een kopie van de functieheader.
Worden er veel verschillende bestanden gebruikt met veel functies, dan is het vaak lastig om de prototypen goed op te schrijven. De prototypen kunnen ook in een apart headerbestand worden geplaatst, dat hoort bij het c-bestand met de betreffende functies. Code 4.6 uit bestand `age.h` bevat het prototype van de functie `print_age` uit het bestand `age.c`. In code 4.5 staat op regel 1 in plaats van het prototype een **`#include`**-opdracht met het bestand `age.h`.

**Code 4.5 : Code bestand main.c.**

```
#include "age.h"
int  age=29;

int main(void)
{
  print_age(age);
  return 0;
}
```

**Code 4.6 : Code bestand age.h.**

```
void print_age(int age);
```

**Code 4.7 : Code bestand age.c.**

```
#include <stdio.h>

void print_age(int age)
{
  printf("Your age is %d", age);
}
```

**Uitleg code 4.5 regel 1**
`#include "age.h";`

Op regel 1 van code 4.5 staat bij **`#include`** de bestandsnaam tussen dubbele aanhalingstekens in plaats van tussen het < en > teken. De compiler zoekt bij dubbele aanhalingstekens het headerbestand in de werkdirectory bij de andere -bestanden.

In figuur 4.3 zijn `main.c` en `age.c` afzonderlijk met de optie `-c` gecompileerd. Bij het linken is de objectcode `age.o` niet meegenomen. De linker meldt dat er een niet gedefinieerde verwijzing `_print_age` is en kan zo geen uitvoerbaar programma maken.

```
/cc/functions
/cc/functions $ gcc -c main.c -Wall

/cc/functions $ gcc -c age.c -Wall

/cc/functions $ gcc -o printage main.o -Wall
main.o:main.c:(.text+0x33):  undefined reference to `_print_age'
collect2:  ld returned 1 exit status

/cc/functions $
```

**Figuur 4.3 : De afzonderlijke compilatie van main.c en age.c.**

## 4.1 Formele en actuele parameters

De begrippen parameter en variabele worden vaak door elkaar gebruikt. Hier is een parameter een variabele, die gedeclareerd is in de parameterlijst van een functie.

De parameters in de functieheader bij de functiedefinitie worden formele parameters genoemd. In code 4.8 zijn de `l`, `w` en `h` op regel 4 formele parameters. Deze parameters krijgen pas een waarde bij de aanroep van de functie. De parameters die bij de aanroep met de functie worden meegegeven, worden de actuele parameters genoemd. Op regel 14 zijn `length`, `w` en `12` de actuele parameters. De functie `volume` gebruikt de waarden van deze actuele parameters om het volume uit te rekenen. De naam van een actuele en formele parameter mag verschillend zijn. De lengte van het volume heeft als actuele waarde `length` en als formele parameter `l`. De naam van een overeenkomstige actuele en formele parameter mag ook gelijk zijn. In code 4.8 wordt voor de breedte in beide gevallen een `w` gebruikt. Het is vaak handig om dezelfde naam te gebruiken; het gaat immers meestal om een en hetzelfde begrip. Het gaat hier in beide gevallen om de breedte van het volume. Wel is de *scope* van de beide `w`'s verschillend. De formele parameter `w` is gedeclareerd in de functie `volume` en de actuele parameter is in dit geval gedeclareerd in de hoofdroutine `main`.

Scope betekent reikwijdte. Bij programmeertalen is dit het programmadeel waarbinnen een variabele of functie geldig of bereikbaar is.

**Code 4.8 : Formele en actuele parameters.**


```
1  #include <stdio.h>
2  int length=24;
3
4  int volume(int l, int w, int h)
5  {
6    int v;
7    v = l*w*h;
8    return v;
9  }
10
11 int main(void)
12 {
13   int vol, w=4;
14   volume(length, w, 12);
15
16   printf("The volume is: %d", vol);
17   return 0;
18 }
```

## 4.2 De scope van functies en variabelen

Het deel van het programma waarbinnen een variabele of functie geldig is, wordt de *scope* genoemd. C kent een *file*-, een *function*-, een *block*- en een *function prototype scope*.
Een variabele, die in een functieheader of in de functiebody is gedeclareerd, is alleen geldig in die betreffende functie. Buiten de functie zijn deze lokale variabelen onbekend. Een variabele, die in een prototype wordt gedeclareerd, is buiten het prototype onbekend. Variabelen, die buiten een functie of buiten een functieprototype staan, noemt men globaal.

Code 4.9 toont zeven keer een declaratie van een variabele `age`. Op regel 3 staat een globale declaratie. Deze `age` is geldig vanaf deze regel tot het einde van het bestand. De lokale variabele `age` van de functie `print_age1` is alleen zichtbaar in deze functie. De variabele `age` in het prototype op regel 15 is slechts geldig in de parameterlijst. De functie `main` heeft een lokale variabele `age` die bekend is vanaf de declaratie op regel 19 tot het einde van de functie `main` op regel 33.

In `main` staat tussen 21 en 24 een blok met variabele `age`, die alleen zichtbaar is vanaf de declaratie op regel 22 tot het einde van het blok. De declaratie van `age` in de functieheader van de functie `print_age3` is alleen bekend in deze functie. De functie bevat ook een blok met een declaratie van `age`, die alleen zichtbaar is van regel 39 tot het einde van het blok op regel 41.

De variabele `age` wordt acht keer afgedrukt. Regel 20 ligt in de scope van de lokale declaratie van regel 19 en de globale declaratie van regel 3. Omdat de lokale declaratie *dichter bij ligt*, wordt de waarde 29 afgedrukt. De `printf` van regel 23 ligt ook in de scope van de blokdeclaratie van regel 21 en drukt daarom 14 af. Bij regel 25 wordt weer 29 afgedrukt.

Regel 27 roept de functie `print_age2` aan, die gedeclareerd staat op regel 10. De afdrukfunctie in `print_age2` ziet alleen de globaal gedeclareerde variabele `age` van regel 3 en drukt 24 af.

**Code 4.9 : Bestand scope.c kent zeven verschillende variabelen age.** Een globale declaratie in het bestand, een lokale declaratie bij een prototype, twee lokale declaraties in een blok en drie lokale declaraties in een functie. Met grijstinten is de scope van de verschillende declaraties aangegeven.

```
1   #include <stdio.h>
2
3   int age=24;
4
5   void print_age1(int age)
6   {
7     printf("My age is %d\n", age);
8   }
9
10  void print_age2(void)
11  {
12    printf("My age is %d\n", age);
13  }
14
15  void print_age3(int age);
16
17  int main(void)
18  {
19    int age=29;
20    printf("My age is %d\n", age);    // My age is 29
21    {
22      int age=14;
23      printf("My age is %d\n", age);  // My age is 14
24    }
25    printf("My age is %d\n", age);    // My age is 29
26
27    print_age2();                     // My age is 24
28    print_age1(38);                   // My age is 38
29    print_age3(34);                   // My age is 34
30                                      // My age is 69
31                                      // My age is 34
32    return 0;
33  }
34
35  void print_age3(int age)
36  {
37    printf("My age is %d\n", age);
38    {
39      int age = 69;
40      printf("My age is %d\n", age);
41    }
42    printf("My age is %d\n", age);
43  }
44
45
```

lokaal in functie `print_age1()` (regels 5–8)

lokaal in prototype (regel 15)

lokaal in block (regels 22–24)

lokaal in functie `main()` (regels 17–33)

lokaal in block (regels 39–41)

lokaal in functie `print_age3()` (regels 35–43)

globaal in bestand (regels 1–45)

Bij de aanroep op regel 28 wordt de waarde 38 aan `print_age1` meegegeven. De `printf` in `print_age1` ligt in de scope van de variabele `age` uit de functieheader van `print_age1` en drukt 38 af.

Functie `print_age3` wordt aangeroepen met de waarde 34 en drukt drie keer een leeftijd af. De tweede `printf` van regel 40 staat in een blok, gebruikt de declaratie van regel 39 en drukt 69 af. De eerste en de derde `printf` kijken naar declaratie van regel 35 en drukken 34 af.

**Code 4.10 : Call by reference.**

```
1  #include <stdio.h>
2
3  int get_age1(void)
4  {
5    int a=19;
6    return a;
7  }
8
9  void get_age2(int *ptr_age)
10 {
11   int a=20;
12   *ptr_age= a;
13 }
14
15 int main(void) {
16   int age;
17
18   age = get_age1();
19   printf("My age is %d", age);
20   get_age2(&age);
21   printf("My age is %d", age);
22
23   return 0;
24 }
```

## 4.3 Call by reference

In C bevat de parameterlijst van een functie alleen ingangswaarden en nooit uitgangswaarden Er zijn twee manieren om een functie toch iets terug te laten geven: met een return of met een methode die *call by reference* heet. Met een return wordt er maar één waarde geretourneerd. Moet een functie meer variabelen teruggeven dan kan dat alleen met *call by reference*.

In C worden een aantal tekens op meerdere manieren gebruikt. Afhankelijk van de context is een & het adres of de *bitwise and* of de helft van een logische *and* (&&).
Een * is de inhoud van een adres, een pointer bij een typedeclaratie of gewoon een vermenigvuldiging.

De truc van *call by reference* is om het adres van een variabele mee te geven en de functie het resultaat op dat adres te laten invullen. Code 4.10 bevat een functie `get_age1`, die een leeftijd met een return teruggeeft, en een functie `get_age2`, die een *call by reference* gebruikt. Op regel 9 staat in de parameterlijst van de functie `get_age2` een pointer `int *ptr_age`. Dit is een pointer die naar een variabele van het type `int` wijst. Een pointer is een geheugenplaats waar het adres van een andere geheugenplaats kan staan. Bij de aanroep van de functie op regel 20 staat voor `age` een ampersand (&-teken). Met dit teken wordt in plaats van de waarde `age` het adres van `age` aan de functie `get_age2` meegegeven. De ampersand wordt de adresoperator genoemd.

De betekenis van de asterisk in regel 9 verschilt met die van regel 12. De eerste is een pointerdeclaratie en de laatste staat voor de inhoud van het adres waar de pointer naar wijst.

De ingangsparameter `ptr_age` van de functie `int get_age2` bevat bij de aanroep dus het adres van de variabele `age`. Bij de toewijzing op regel 12 staat voor `ptr_age` een asterisk (*) of sterretje. Dit heeft tot effect dat `a` niet toegekend wordt aan `ptr_age`, maar wordt ingevuld op de geheugenplaats waar `ptr_age` naar wijst.
Pointers en pointerberekeningen zijn in C een belangrijk en krachtig mechanisme. In hoofdstuk 15 komen pointers en het gebruik van pointers uitgebreid aan

bod. Vooralsnog is het belangrijk dat men weet dat het mogelijk is, om met een functie via *call by reference* informatie uit een functie te krijgen. Bovendien is het handig om te weten dat `&` een adresoperator kan zijn; dat `*` voor een typedefinitie een pointer declareert en dat een `*` voor een pointervariabele de inhoud van de geheugenplaats is waar de pointervariabele naar wijst.

# 5

# In- en uitvoer

Doelstelling

In dit hoofdstuk leer je hoe je in C gegevens op het scherm afdrukt en hoe je informatie van het toetsenbord inleest.

Onderwerpen

De behandelde onderwerpen zijn:

- Geformatteerde in- en uitvoer.
- Het gebruik van de adres-operator (&) bij het lezen van variabelen met `scanf`.
- De *escape sequences* voor *format strings*.
- Ongeformatteerde in- en uitvoer.
- Het doorgeven van gegevens aan een programma met behulp van argumenten.

De voorbeelden voor in- en uitvoer zijn:

- Invoeren en afdrukken van naam en leeftijd.
- Ongeformatteerd invoeren en afdrukken.
- Invoeren van naam en leeftijd met behulp van argumenten.
- Robuuste versie van invoeren van naam en leeftijd met behulp van argumenten.
- Het verschil tussen == en =.

Een programma, dat alleen tekst in een commandovenster afdrukt, is over het algemeen niet erg nuttig. Veel interessanter zijn programma's, die invoer van een gebruiker nodig hebben.
C kent standaard een aantal mogelijkheden om tekst naar het scherm te schrijven en om gegevens vanaf een toetsenbord in te lezen. Ook zijn er meerdere methoden om tekst of binaire gegevens uit een bestand te lezen of naar een bestand te schrijven.
Naast het gebruik van een toetsenbord en een beeldscherm of het toepassen van bestanden om informatie in- en uit te voeren, kan er ook via verschillende interfaces worden gecommuniceerd, zoals: de USB-, de seriële of de parallelle poort. Alleen zijn hier geen standaard C-oplossingen voor; bij Windows en Unix gaat dit verschillend.
Bijlage A geeft een uitleg over de communicatie met de seriële poort. Het schrijven naar en het lezen uit bestanden wordt besproken in paragraaf 17.1. Dit hoofdstuk behandelt de invoer van tekst met het toetsenbord en de uitvoer naar het scherm.

**Code 5.1 : Lezen en afdrukken naam en leeftijd.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    char name[16];
 6    int  age;
 7
 8    printf("What is your name?");
 9    scanf("%s", name);
10
11    printf("What is your age?");
12    scanf("%d", &age);
13
14    printf("Welcome\n");
15    printf("\tYour name is %s\n", name);
16    printf("\tYour age is %d", age);
17
18    return 0;
19  }
```

## 5.1 Geformatteerde in- en uitvoer

Code 5.1 vraagt achtereenvolgens om je naam en je leeftijd en drukt vervolgens deze informatie geformatteerd af. Figuur 5.1 toont de compilatie en de aanroep van het programma. Het programma `whoareyou` vraagt om je naam op te geven.

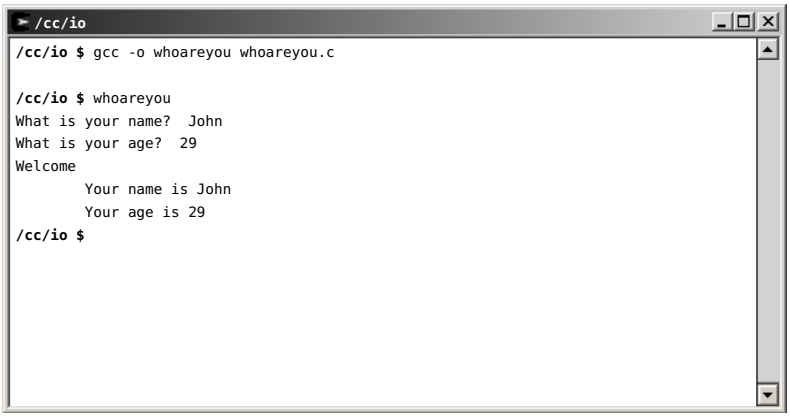

**Figuur 5.1 : De compilatie van code 5.1 en de aanroep het programma `whoareyou`. De gebruiker wordt gevraagd zijn naam in te toetsen.**

Met de MinGW in een Msys-shell, werkt dit niet goed. De uitvoer van tekst verschijnt niet op het juiste moment. Dit komt doordat Msys de uitvoerbuffer niet leegt. Met een `fflush`-opdracht voor de `scanf`'s van regel 9 en regel 12 werkt het programma bij Msys ook goed.

Nadat je je naam hebt opgeven en op de entertoets hebt gedrukt, wordt er om je leeftijd gevraagd, zoals figuur 5.2 laat zien.


**Figuur 5.2 : Na de invoer van je naam vraagt het programma `whoareyou` om je leeftijd.**

Direct nadat je je leeftijd hebt opgegeven, drukt het programma de ingelezen informatie geformatteerd af. Figuur 5.3 laat deze uitvoer zien.




**Figuur 5.3 : Na de invoer van je leeftijd drukt het programma `whoareyou` direct de naam en leeftijd af.**

**Uitleg code 5.1 regel 9**
`int scanf(char *fmt, ...)`

De functie `scanf` leest de met het toetsenbord ingevoerde tekst. De tekst moet voldoen aan het *format* van `scanf`. Het *format* is het eerste argument van de functie `scanf`. In dit geval bevat de *format string* alleen de *format specifier* die aangeeft dat er een string gelezen moet worden, namelijk: `%s`. De functie verwacht dus een tekststring. Deze tekststring wordt ingevuld in het eerstvolgende argument van de functie. Hier is dat de variabele `name`.

**Regel 12**
*adresoperator*
`&`

Zoals in paragraaf 4.3 bij de bespreking van code 4.10 is uitgelegd, kunnen aan een functie alleen ingangswaarden worden doorgegeven. Functies kennen geen uitgangsparameters. Met de *call by reference*-methode wordt het adres van een variabele meegegeven. De functie vult de uitkomst op deze geheugenplaats in. De `&` zorgt ervoor dat aan `scanf` het adres van de variabele `age` wordt meegegeven. De functie `scanf` zet de waarde die ingetoetst wordt op dit adres neer.
Op regel 9 staat voor `name` juist geen `&`. Dit moet ook niet, want de variabele `name` is een array en is in feite het adres van het eerste karakter.

**Tabel 5.1 : Escape sequences voor format strings.**

| escape sequence | hexadecimale ASCII-waarde | betekenis |
|---|---|---|
| `\n` | `0x0A` | newline |
| `\r` | `0x0D` | carriage |
| `\f` | `0x0C` | formfeed |
| `\t` | `0x09` | horizontal tab |
| `\b` | `0x08` | backspace |
| `\v` | `0x0B` | vertical tab |
| `\a` | `0x07` | bell |
| `\\` | `0x5C` | backslash |
| `\"` | `0x22` | double quote |
| `\'` | `0x27` | single quote |
| `\0` | `0x00` | nul |

**Regel 14**
*escape sequence*
`\n`

De *format string* van de `printf`-functie bevat een zogenoemde *escape sequence*, namelijk: `\n`. Dit geeft een nieuwe regel aan. Tabel 5.1 toont een overzicht van de *escape sequences*, die in de *format string* van `printf` en `scanf` kunnen staan.

**Regel 9**
`int fflush(FILE *f)`

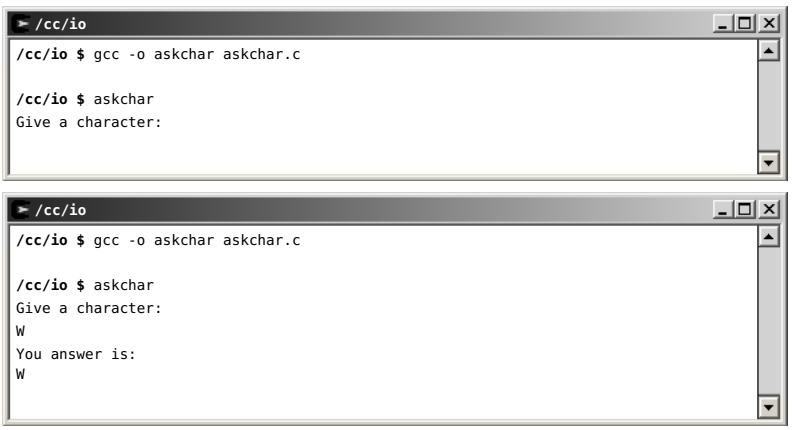
In de kantlijn bij code 5.2 staat dat deze code met MinGW/Msys niet werkt en dat er twee `fflush`-opdrachten moeten worden toegevoegd, zoals hier gedaan is:

```
printf("What is your name?");
fflush(stdout);
scanf("%s", name);
printf("What is your age?");
fflush(stdout);
scanf("%d", &age);
```

Bij Msys werkt het buffermechanisme niet goed. Normaal wordt bij `scanf` de uitvoerbuffer geleegd. Dat gebeurt bij Msys niet. Buffering van de in- en uitvoer is nodig om het aantal systeemaanroepen te beperken. De functie `fflush` leegt de uitvoerbuffer van een bestand. Aan `fflush` wordt daarom de filepointer van het bestand meegegeven, waarvan de buffer geleegd moet worden. Bij de standaarduitvoer naar het beeldscherm is de filepointer `stdout`. Andere bestandsbewerkingen en filepointers worden besproken in paragraaf 17.1.

## 5.2 Ongeformatteerde in- en uitvoer

Naast de geformatteerde in- en uitvoer kent C ook ongeformatteerde in- en uitvoer. Code 5.2 laat een voorbeeld zien. Deze code vraagt om een letter en drukt deze letter vervolgens af. Figuur 5.4 laat de in- en uitvoer van dit programma zien.




**Figuur 5.4 : Het programma van code 5.2 vraagt om een letter (*character*) en nadat de gebruiker een letter (W) heeft opgegeven drukt het deze letter af.**

**Uitleg code 5.2 regel 7**
`int puts(char *s);`

De functie `puts` schrijft een string `s` naar de standaarduitvoer en vervangt de *end-of-string* door een newline. Deze functieaanroep `puts(buf);` is identiek aan `printf("%s\n", buf);`.

**Regel 8**
`int getchar(void);`

De functie `getchar()` leest een karakter van de standaardinvoer. Het returntype is geen **`char`** maar een **`int`**. Deze functie hangt nauw samen met de functies `getc()` en `fgetc()`, die in de paragraaf 17.1 bij het lezen uit bestanden worden besproken.

**Regel 10**
`int putchar(int c);`

De functie `putchar()` schrijft een karakter naar de standaarduitvoer. Het karakter wordt niet als **`char`** maar als **`int`** meegegeven. Deze functie hangt nauw samen of is afgeleid met de functies `putc()` en `fputc()`, die in de paragraaf 17.1 worden besproken.

**Code 5.2 : Ongeformatteerd lezen en afdrukken.**

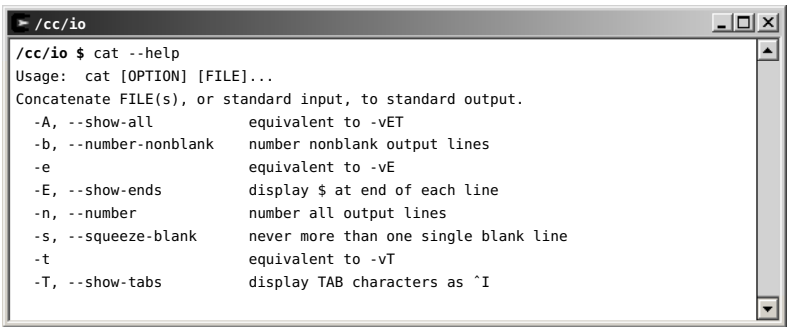
```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int c;
6
7    puts("Give a character:");
8    c = getchar();
9    puts("You answer is:");
10   putchar(c);
11
12   return 0;
13 }
```

In paragraaf 17.1 worden de functies voor het lezen uit en het schrijven naar bestanden uitgebreider besproken. Daarbij komen ook de voor- en nadelen van geformatteerde en ongeformatteerde in- en uitvoer aan de orde.

## 5.3 Argumenten doorgeven aan een programma

Sommigen vinden het doorgeven van informatie via de commandoregel ouderwets. Waarschijnlijk omdat het op tekst gebaseerd is. Toch is het een praktische en veelgebruikte methode. Bij de ontwikkeling en het testen van programma's is het handig. De oude opdrachten kunnen bij de commandoregel met de pijltjestoetsen worden teruggehaald. Bij een programma dat zelf om informatie vraagt, moet dit steeds opnieuw worden ingetoetst.

Naast dat een programma zelf om invoer vraagt of dat het informatie uit een bestand leest, kan de gebruiker ook argumenten of parameters aan een programma meegeven. Bij de aanroep van het programma worden dan een of meer strings achter de programmanaam ingetoetst. Dit is een handige manier om het gedrag van een programma te beïnvloeden. In Unix is gebruikelijk om argumenten aan programma mee te geven. Dit zijn bijvoorbeeld de naam van een invoerbestand of bepaalde opties. Figuur 5.5 toont een deel van de opties van het Unix-commando `cat`. Met `cat` worden bestanden aaneengevoegd afgedrukt op de standaarduitvoer. De aanroep `cat -b getch.c` drukt de inhoud van het bestand `getch.c` af met regelnummers.

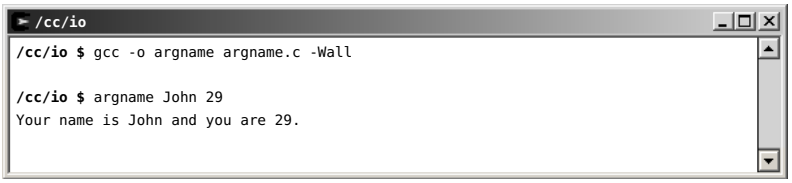



**Figuur 5.5 : Hier staat een deel van de opties van het Unix-commando `cat`. Met `cat` worden bestanden aaneengevoegd.**

In code 5.3 staat een programma dat twee argumenten meekrijgt, namelijk: een naam en een leeftijd. Figuur 5.6 toont de uitvoer van het programma. De parameterlijst van de hoofdroutine `main` is nu niet **`void`**, maar heeft twee parameters `argc` en `argv`. Bij het uitvoeren van het programma staat in `argc` het aantal argumenten en bevat `argv` een array van strings. Deze strings kunnen door het programma worden gebruikt. In dit geval staat in `argv[1]` de naam en in `argv[2]` de leeftijd.

**Code 5.3 : Het afdrukken van naam en leeftijd met argumenten.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6     int age;
7
8     age = atoi(argv[2]);
9
10    printf("Your name is %s and you are %d", argv[1], age);
11
12    return 0;
13 }
```




**Figuur 5.6 : Het programma van 5.3 drukt de argumenten `John` en `29` af.**

**Uitleg code 5.3 regel 4**

```
main(int argc,
     char *argv[])
{
}
```

Aan `main` kan een variabel aantal argumenten worden meegegeven. De parameterlijst van `main` heeft voor het doorgeven van een willekeurig aantal argumenten slechts twee parameters nodig: een integer `argc` en een pointer `char *argv[]`, die naar een lijst met argumenten wijst. De lijst met argumenten wordt genummerd vanaf `0`. `argv[0]` bevat de naam van het programma, `argv[1]` bevat het eerste argument, `argv[2]` bevat het tweede argument, etc.. Het totaal aantal argumenten, inclusief de naam van het programma, staat in `argc`. Bij de aanroep

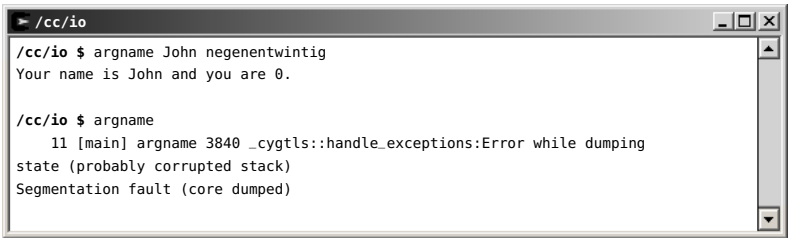
```
argname John 29
```

bevat `argv[0]` de string `"argname"`, `argv[1]` de string `"John"`, `argv[2]` de string `"29"` en is `argc` gelijk aan `3`.

**Regel 8**

```
int atoi(char *s)
```

De routine `atoi` zet een alfanumerieke string om in een integer. In dit voorbeeld wordt de string `"29"` omgezet in het getal `29`. Als `atoi` een niet-alfanumerieke waarde meekrijgt, geeft deze de waarde `0` terug. Het prototype van `atoi` staat in `stdlib.h`.




**Figuur 5.7 : De aanroep van het programma van code 5.3 met verkeerde invoer.**

Het programma uit code 5.3 is niet robuust. De gebruiker kan informatie opgeven waardoor het crasht of onzin oplevert. In figuur 5.7 wordt het programma eerst aangeroepen met als tweede argument een niet-alfanumerieke waarde. Om-

**Code 5.4 : Een robuuste versie van code 5.3.**

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main(int argc, char *argv[])
 5  {
 6     int age;
 7
 8     if ( argc < 3 ) {
 9        printf("usage: %s <name> <age>\n", argv[0]);
10        return 1;
11     }
12
13     if ( (age = atoi(argv[2])) == 0) {
14        printf("usage: %s <naam> <age>\n", argv[0]);
15        printf("       <age> must be a number");
16        return 1;
17     }
18
19     printf("Your name is %s and your age is %d", argv[1], age);
20
21     return 0;
22  }
```

dat `atoi` een `0` retourneert als de ingangswaarde niet-alfanumeriek is, wordt de leeftijd `0` in plaats van 29. Vervolgens wordt het programma zonder argumenten aangeroepen. Het gedrag wordt dan onvoorspelbaar. In dit geval crasht het.
Het is juist bij C heel belangrijk om veel zorg te besteden aan de neveneffecten van een programma. Alle mogelijke fouten moet de programmeur afvangen. Code 5.4 staat een verbeterde versie. Bij minder dan drie argumenten (programmanaam, naam en leeftijd) wordt er een foutmelding gegeven. Zijn er meer dan drie argumenten, dan is dat geen probleem, want deze worden niet gebruikt. Als de leeftijd geen alfanumerieke waarde — dat is het geval als `atoi()` `0` retourneert — is, dan wordt er ook een foutmelding gegeven.

**Uitleg code 5.4 regel 8**
*relationele operatoren*
`<`
`<=`
`==`
`>=`
`>`
`!=`

De `<` van regel 8 is net als `==` op regel 13 een relationele of vergelijkingsoperator. Een relationele operator vergelijkt twee waarden en geeft een 1 terug als deze vergelijking waar is en `0` als deze vergelijking niet waar is. Er zijn zes relationele operatoren:

| | |
|---|---|
| `<` | is kleiner dan |
| `<=` | is kleiner of gelijk aan |
| `==` | is gelijk aan |
| `>=` | is groter of gelijk aan |
| `>` | is groter dan |
| `!=` | is ongelijk aan |

**Regel 13**
`==` *versus* `=`

Let op het verschil tussen `==` en `=`. Het `==`-teken is de gelijkheidsoperator en het vergelijkt twee waarden. Het `=`-teken is een toekenning. Een veel voorkomende fout is een toekenningsoperator (`=`) gebruiken op een plaats waar een de gelijkheidsoperator (`==`) bedoeld is. Voorbeeld 5.5 geeft voor elke `x` deze uitvoer:

```
Will always be executed!
The value of x is 0.
```

**Code 5.5 : Het verschil tussen == en =.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6     int x;
7
8     x = atoi(argv[1]);
9
10    // x can be anything
11    if ( x = 0  ) {
12        printf("Will never be executed!\n");
13    } else {
14        printf("Will always be executed!\n");
15    }
16
17    // x will always be 0 at this point
18    printf("The value of x is %d.\n", x);
19
20    return 0;
21 }
```

**Uitleg code 5.4 regel 8**
**if**

C kent twee conditionele toewijzingen: het **if**-statement en het **switch**-statement. De opdracht na het **if**-statement wordt uitgevoerd als de voorwaarde waar is:

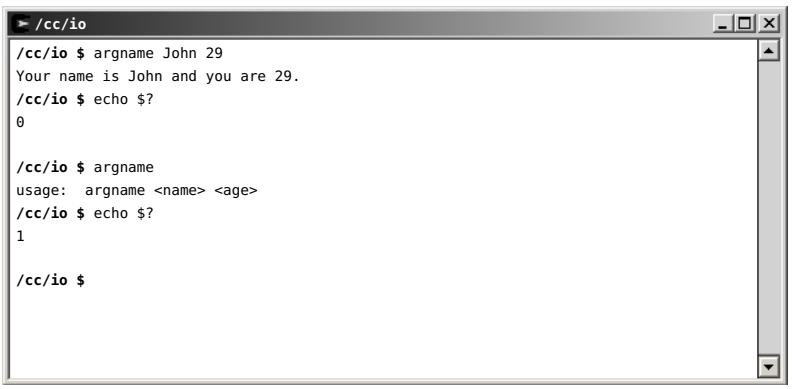
```
if ( conditie  )
        wordt toegekend als conditie waar is;
```

**Regel 9**
`argv[0]`

Het argument `argv[0]` bevat de naam van het programma. Dit is handig bij foutmeldingen om een gebruiksvoorschrift af te drukken.

**Regel 10**
**return**

Met het **return**-statement kan een functie een waarde teruggeven. De hoofdroutine `main` is een gewone functie die dus ook een waarde terug kan geven. Op Unix/Linux-systemen wordt dit in scripts gebruikt om de status van een programma te achterhalen. Het programma van 5.4 geeft een 1 terug als er een fout is opgetreden en een `0` als het correct is uitgevoerd. Het Unix-commando `echo $?` geeft de status van het laatst uitgevoerde programma.




**Figuur 5.8 : Het programma van 5.4 vangt foutieve invoer af. De returncode is `0` bij correcte invoer en 1 bij foutieve invoer.**

# 6

# Voorwaardelijke opdrachten

Doelstelling

In dit hoofdstuk leer je wat voorwaardelijke opdrachten zijn, waarvoor deze opdrachten dienen en hoe je deze gebruikt.

Onderwerpen

De behandelde onderwerpen zijn:

- Het **`if`**-statement met de **`if`**-, **`if-else`**- en de **`if-else-if`**-vorm.
- De bloktoewijzing.
- De geneste **`if`**-statements.
- Het **`switch`**-statement.
- De increment-operator (`++`) en de decrement-operator (`--`).
- De conditionele operator (`?:`).

De demonstratieprogramma's met voorwaardelijke opdrachten zijn:

- Een voorbeeld met **`if`**.
- Een voorbeeld met **`if-else`** en met *block*-statements.
- Een voorbeeld met geneste **`if`**'s.
- De functie `print_ctype` met geneste **`if`**'s zonder else.
- De functie `print_digit` met de **`if-else-if`**-vorm.
- De functie `print_digit` met een **`switch`**.
- Een programma met een **`switch`** dat eigenschappen van cijfers afdrukt.

Een normaal softwaresysteem reageert op informatie van buitenaf of verwerkt de invoergegevens op een bepaalde manier. Sommige functionaliteiten zullen voorwaardelijk uitgevoerd moeten worden en sommige acties zullen moeten worden herhaald. Denk bijvoorbeeld aan een tekstverwerker. De eerste pagina moet bijvoorbeeld geen paginanummer krijgen en de andere pagina's moeten worden genummerd. En zolang er nog regels op een pagina passen, moet de volgende regel aan de pagina worden toegevoegd.
De besturingsopdrachten of *control statements*, die hiervoor nodig zijn, worden ingedeeld in voorwaardelijke of conditionele opdrachten en in iteratieve of herhalingsopdrachten. C kent daarom — net als de meeste programmeertalen — constructies voor het uitvoeren van voorwaardelijke of conditionele opdrachten (*conditional assignments*) en herhalingsopdrachten (*loop assignments*). De herhalingsopdrachten worden in hoofdstuk 7 besproken. Dit hoofdstuk bespreekt de voorwaardelijke opdrachten.

**Code 6.1 : Voorbeeld met if.**

```
 1  #include <stdio.h>
 2  #include <ctype.h>
 3
 4  int main(int argc, char *argv[])
 5  {
 6    int x;
 7
 8    if ( argc < 2 )
 9      return 1;
10
11    x = argv[1][0];
12
13    if ( isdigit(x) )
14      printf("%c is a digit\n", x);
15
16    return 0;
17  }
```

Er zijn twee soorten voorwaardelijke opdrachten: het **if**-statement en het **switch**-statement. Hiervan kent de eerste drie verschillende vormen: de **if**, de **if-else** en de **if-else-if**.

## 6.1 Het if-statement: de if-vorm

Achter het sleutelwoord **if** staat altijd tussen ronde haken een conditionele uitdrukking, die waar of niet waar kan zijn. Het statement dat direct na deze conditionele toekenning staat, wordt uitgevoerd als deze conditie waar is. De syntax van de **if**-vorm is:

```
if ( conditie )
   statement, dat uitgevoerd wordt als conditie waar is;
```

Het programma van code 6.1 bevat twee if-statements. De eerste **if** controleert of er een teken aan het programma is meegegeven. Als het aantal argumenten kleiner is dan twee, wordt het programma afgebroken. De tweede **if** kijkt of het teken een cijfer is. Als het een cijfer is, wordt er een tekst afgedrukt, anders wordt er niets afgedrukt.

**Uitleg code 6.1 regel 11**
`argv[1][0]`

Het argument `argv[1]` is een string. In dit geval zal dat bijvoorbeeld `"5"` zijn. Aan de variabele `x` moet dan het karakter `'5'` worden toegekend. Het karakter `'5'` is het eerste karakter van de string `"5"`. Het eerste karakter van een string `s` is `s[0]`. Op dezelfde manier is `argv[1][0]` het eerste karakter van de string `argv[1]`. Alle invoer, die met een vijf begint, maakt dat `x` de waarde `'5'` krijgt, dus `"5"`, `"555"`, `"5.03"` en `"5ap49s"` leveren allemaal een `'5'` op.

**Regel 13**
`int isdigit(int c)`

De functie `isdigit` retourneert 1 (waar) als het karakter `c` een cijfer is, anders retourneert het `0` (niet waar). Het prototype van `isdigit` staat in `ctype.h`.

**Regel 2**
`#include <ctype.h>`

Het headerbestand `ctype.h` bevat de prototypen en macro's van eenvoudige tests en bewerkingen op karakters. Dit bestand is nodig voor de functie `isdigit()`. Tabel 6.1 geeft een overzicht van de macro's uit `ctype.h`.

**Tabel 6.1 : Overzicht van tests en bewerkingen op karakters.**

| prototype | functionaliteit |
|---|---|
| `int isalnum(int c)` | test of `c` alfanumeriek is |
| `int isalpha(int c)` | test of `c` een hoofd- of kleine letter is |
| `int islower(int c)` | test of `c` een kleine letter is |
| `int isupper(int c)` | test of `c` een hoofdletter is |
| `int isdigit(int c)` | test of `c` een cijfer is |
| `int isxdigit(int c)` | test of `c` een hexadecimaal cijfer is |
| `int iscntrl(int c)` | test of `c` een control-karakter is |
| `int ispunct(int c)` | test of `c` interpunctie is |
| `int isgraph(int c)` | test of `c` *printable* en geen *white space* is |
| `int isprint(int c)` | test of `c` *printable* is |
| `int isspace(int c)` | test of `c` een *white space* is |
| `int isblank(int c)` | test of `c` een spatie of een tab is |
| `int isascii(int c)` | test of `c` een ASCII-waarde is |
| `int tolower(int c)` | converteert `c` naar een kleine letter |
| `int toupper(int c)` | converteert `c` naar een hoofdletter |
| `int toascii(int c)` | maakt van `c` een ASCII-waarde |

## 6.2 De bloktoewijzing

Als er bij een **`if`** meer dan een opdracht uitgevoerd moet worden, moeten deze opdrachten tussen accolades gezet worden. Een stuk code tussen een accolade openen `{` en een accolade sluiten `}` wordt een blok (*block*) of bloktoewijzing (*block statement*) genoemd.

```
if ( condition ) {
   assignment1;
   assignment2;
   ...
}
```

Het is verstandig om bij alle besturingsopdrachten de toewijzingen altijd in een blok te plaatsen. Dat voorkomt misverstanden.

```
if ( x == 0 )
   printf("The first line.\n");
   printf("The second line.\n");
```

```
if ( x == 0 ) {
   printf("The first line.\n");
   printf("The second line.\n");
}
```

**Figuur 6.1 : Het effect van een bloktoewijzing.** Bij de linker **`if`** wordt de eerste `printf` voorwaardelijk afgedrukt. De tweede `printf` komt na dit **`if`**-statement en wordt altijd afgedrukt. Bij de rechter **`if`** — met de bloktoewijzing — worden de beide `printf`'s voorwaardelijk afgedrukt. De scope van beide **`if`**'s is met grijs aangegeven.

Bij de linker **`if`** uit figuur 6.1 valt alleen de eerste `printf` binnen de scope van de **`if`**. Hoewel, door het inspringen van de tweede `printf`, het lijkt dat deze bij het **`if`**-statement hoort, valt deze buiten de scope. Daarom wordt deze opdracht altijd uitgevoerd. Bij de rechter **`if`** liggen de `printf`-opdrachten allebei binnen de scope van de **`if`** en worden beide alleen uitgevoerd als `x` de waarde nul heeft.

Code 6.2 is een verbeterde versie van het programma uit code 6.1. Bij de **`if`** op regel 8 is een afdrukregel toegevoegd en is een blok noodzakelijk. De toewijzing bij de **`if`** op regel 16 is in een blok geplaatst om misverstanden te voorkomen. Bovendien is op regel 17 een **`else`** toegevoegd.

**Code 6.2 : Voorbeeld met if-else en block statements.**

```
 1  #include <stdio.h>
 2  #include <ctype.h>
 3
 4  int main(int argc, char *argv[])
 5  {
 6    int x;
 7
 8    if ( argc < 2 ) {
 9      printf("usage: %s <token>\n", argv[0]);
10      return 1;
11    }
12
13    x = argv[1][0];
14
15    if ( isdigit(x) ) {
16      printf("%c is a digit\n", x);
17    } else {
18      printf("%c is not digit\n", x);
19    }
20
21    return 0;
22  }
```

## 6.3 Het if-statement: de if-else vorm

Direct na een **if** mag een **else**-statement komen te staan. Het statement na de **else** wordt alleen uitgevoerd als de conditionele uitdrukking van de voorgaande **if** niet waar is. De syntax van de **if-else**-vorm is:

```
if ( conditie )
   statement, dat uitgevoerd wordt als conditie waar is;
else
   statement, dat uitgevoerd wordt als conditie niet waar is;
```

In voorbeeld 6.2 zorgt de **else** op regel 17 ervoor dat de `printf`-opdracht van regel 18 wordt uitgevoerd. Een **else** hoort altijd bij een **if**. Er bestaan geen losse **else**-statements. Een **else** is niet altijd nodig. De twee codes in figuur 6.2 zorgen er allebei voor dat `y` de waarde `30` als `x` negatief en anders `50` is.

```
if ( x < 0 ) {
  y = 30;
} else {
  y = 50;
}
```

```
y = 50;
if ( x < 0 ) {
  y = 30;
}
```

**Figuur 6.2 : Een else zonder else.** De linker code gebruikt een **else** om `y` `50` te maken. De rechter code maakt `y` `50` en verandert daarna `y` als `x` negatief is in `30`.

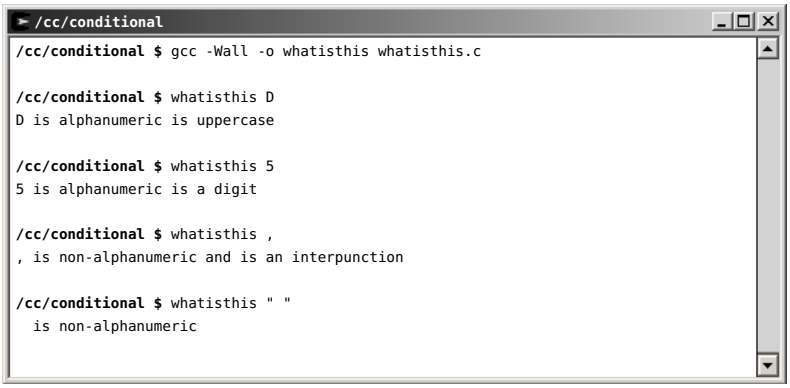
## 6.4 Het nesten van if-statements

De voorwaardelijke statements na een **if** en na een **else** mogen ook weer **if**-statements bevatten. En op hun beurt kunnen deze ook weer voorwaardelijke opdrachten bevatten. Men zegt dan dat **if**-statements mogen worden genest en spreekt dan van geneste **if**'s. Code 6.3 toont een voorbeeld.

**Code 6.3 : Voorbeeld met geneste if's.**

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main(int argc, char *argv[])
5  {
6    int x;
7
8    if ( argc < 2 ) {
9      printf("usage: %s <token>\n", argv[0]);
10     return 1;
11   }
12
13   x = argv[1][0];
14
15
16   if ( isalnum(x) ) {
17     printf("%c is alphanumeric ", x);
18     if ( isdigit(x) ) {
19       printf("is a digit");
20     } else {
21       if ( isupper(x) ) {
22         printf("is uppercase");
23       } else {
24         printf("is lowercase");
25       }
26     }
27   } else {
28     printf("%c is non-alphanumeric ", x);
29     if ( ispunct(x) ) {
30       printf("and is an interpunction");
31     }
32   }
33   printf("\n");
34
35   return 0;
36 }
```

Een nadeel van — diep — geneste **if**'s is dat deze moeilijk te overzien zijn. Het is belangrijk om de layout van de code zorgvuldig vorm te geven, dat wil zeggen het inspringen van de teksten op een consistente wijze te doen en daarbij altijd accolades toe te voegen.
Overigens is het vaak overzichtelijker om geen **else** te gebruiken. Code 6.4 en 6.5 tonen samen een programma met een functie zonder **else**, die dezelfde functionaliteit heeft als code 6.3. In plaats van met een **else** alle mogelijkheden te bekijken, wordt met een **return**-statement de functie verlaten als er geen andere mogelijkheden meer zijn. De evaluatie van de functie `print_ctype` stopt bijvoorbeeld op het moment dat er een cijfer gevonden. De tekst *is alphanumeric and is a digit* is dan al afgedrukt. Figuur 6.3 toont de uitvoer van code 6.3.




**Figuur 6.3 : De uitvoer van het programma van code 6.3.** Dit programma drukt de eigenschappen van een karakter af. Bij de laatste aanroep wordt een spatie getest. Deze moet tussen aanhalingstekens staan, net als alle andere karakters die een speciale betekenis in de *shell* hebben.

**Code 6.4 : Geneste if's zonder else.**

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

void print_ctype(int x);

int main(int argc, char *argv[])
{
  if ( argc < 2 ) {
    printf("usage: %s <token>\n", argv[0]);
    return 1;
  }

  print_ctype(argv[1][0]);

  return 0;
}
```

**Code 6.5 : Functie met geneste if's zonder else.**

```
void print_ctype(int x)
{
  printf("%c ", x);
  if ( isalnum(x) ) {
    printf("is alphanumeric ");
    if ( isdigit(x) ) {
      printf("and is a digit\n");
      return;
    }
    if ( isupper(x) ) {
      printf("and uppercase\n");
      return;
    }
    printf("and lowercase\n");
    return;
  }
  printf("is non-alphanumeric ");
  if ( ispunct(x) ) {
    printf("and is an interpunction\n");
    return;
  }
  printf("\n");
}
```

## 6.5 Het if-statement: de if-else-if vorm

Het **if**-statement kent naast de **if** en de **if-else** ook de **if-else-if**-vorm. De **if-else-if** bevat meerdere **if**'s en test op meerdere condities. De syntax luidt als volgt:

```
if ( conditie_1 )
   statement, als conditie_1 waar is;
else if ( conditie_2 )
   statement, als conditie_2 waar is (en conditie_1 niet waar is);
      ⋮
else if ( conditie_n )
   statement, als conditie_n waar is (en conditie_1 ··· conditie_n-1 niet waar zijn);
else
   statement, als conditie_1 ··· conditie_n niet waar zijn;
```

Code 6.7 geeft een voorbeeld. De uitvoer van dit programma staat in figuur 6.4. Een alternatief voor de **if-else-if** is het **switch**-statement en in het geval van een functie kan ook de truc gebruikt worden om de functie voortijdig te verlaten, zoals eerder bij code 6.5 is toegepast.

## 6.6 Het switch-statement

Het **switch**-statement is naast het **if**-statement de tweede voorwaardelijke opdracht. Het is een variant op de **if-else-if**-vorm van het **if**-statement. Een **switch** is altijd te herschrijven als **if-else-if**. Van een **if-else-if** kan daarentegen niet altijd een **switch** worden gemaakt. De functie `print_digit` uit code 6.7 is in code 6.8 herschreven met een **switch**.

**Code 6.6 : Functie print_digit wordt aangeroepen.**

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

void print_digit(int n);

int main(int argc, char *argv[])
{
  if ( argc < 2 ) {
    printf("usage: %s <digit>\n", argv[0]);
    return 1;
  }

  if ( strlen(argv[1]) > 1 ) {
    printf("usage: %s <digit>\n", argv[0]);
    return 2;
  }

  print_number(argv[1][0]-'0');

  return 0;
}
```

**Code 6.7 : Functie print_digit met een if-else-if.**

```
void print_digit(int n)
{
  if ( n==0 ) {
    printf("zero");
  } else if ( n==1 ) {
    printf("one");
  } else if ( n==2 ) {
    printf("two");
  } else if ( n==3 ) {
    printf("three");
  } else if ( n==4 ) {
    printf("four");
  } else if ( n==5 ) {
    printf("five");
  } else if ( n==6 ) {
    printf("six");
  } else if ( n==7 ) {
    printf("seven");
  } else if ( n==8 ) {
    printf("eight");
  } else if ( n==9 ) {
    printf("nine");
  } else {
    printf("it isn't a digit");
  }
}
```

De **switch** evalueert voor de uitdrukking tussen de ronde haken alle mogelijkheden. Eerst wordt getest of `n` 0 is, vervolgens of deze 1 en daarna of deze 2 is. Als `n` twee is, worden de statements achter de betreffende **case** uitgevoerd en wordt dan two afgedrukt en vanwege het **break**-statement wordt de **switch** direct daarna verlaten. De syntax van het **switch**-statement is:

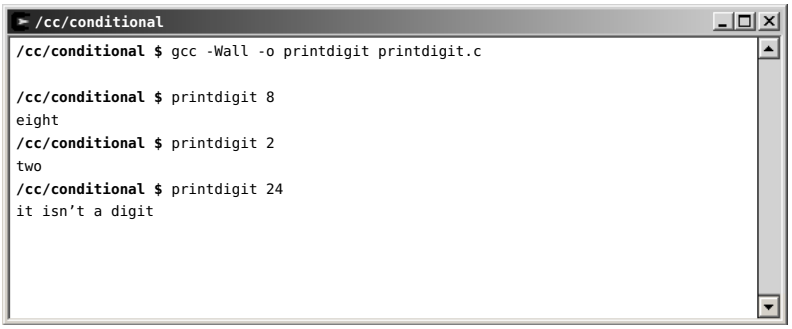
```
switch ( numerieke uitdrukking ) {
case voorwaarde_1:   [statements;]
                     [break;]
case voorwaarde_2:   [statements;]
                     [break;]
      ⋮
case voorwaarde_n:   [statements;]
                     [break;]
[default:]           [statements;]
                     [break;]
}
```

Achter een **case** hoeft niet altijd een statement of een **break** te staan. Staat er geen statement, dan wordt er niets uitgevoerd. Als er geen **break** staat, wordt de **switch** niet verlaten en wordt er verder gegaan bij de volgende **case**. De **default** is optioneel en wordt meestal als laatste keuze van de **switch** gegeven. In principe doet de volgorde van de **case**-statements er niet toe, maar het is gebruikelijk om de **default** als laatste neer te zetten.

**Code 6.8 : Functie print_digit met een switch.**

```
1  void print_digit(int n)
2  {
3    switch (n) {
4      case 0:  printf("zero");
5               break;
6      case 1:  printf("one");
7               break;
8      case 2:  printf("two");
9               break;
10     case 3:  printf("three");
11              break;
12     case 4:  printf("four");
13              break;
14     case 5:  printf("five");
15              break;
16     case 6:  printf("six");
17              break;
18     case 7:  printf("seven");
19              break;
20     case 8:  printf("eight");
21              break;
22     case 9:  printf("nine");
23              break;
24     default: printf("it isn't a digit");
25              break;
26   }
27 }
```

Het programma 6.9 drukt van een getal tussen 0 en 10 de eigenschappen af. In deze code staat niet achter elke **case** een statement of een **break**. Als n gelijk aan 2 is, voert het programma de opdracht van regel 13 uit en drukt af dat het een priemgetal is. Omdat er geen **break** staat, gaat het door met het evalueren van de **switch**. Achter de **case**-statements van regel 14 en 15 staat niets en wordt de opdracht van regel 16 uitgevoerd. Het programma drukt af dat het een even getal is. Op regel 17 staat wel een **break**. Hier is het programma klaar met de evaluatie van de **switch**. In figuur 6.5 staat de uitvoer van het programma voor een aantal cijfers.

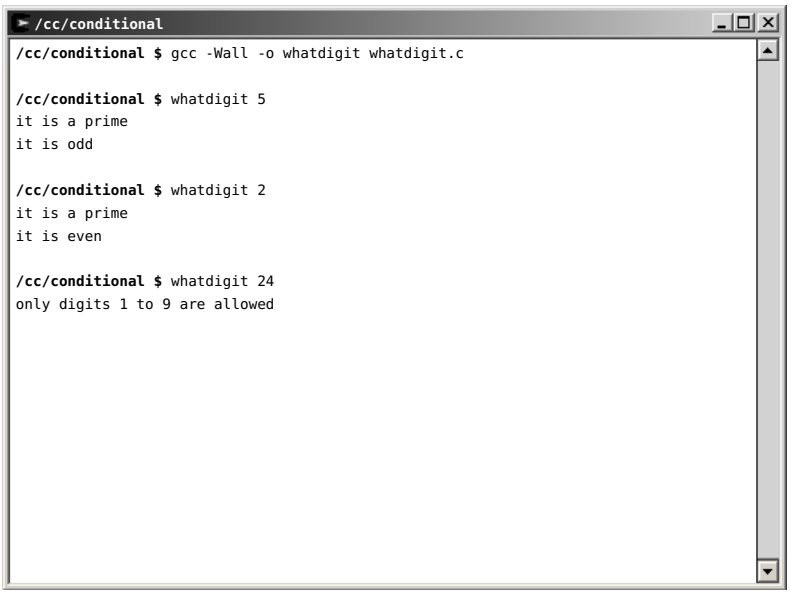



**Figuur 6.4 : De uitvoer van het programma van code 6.6.**

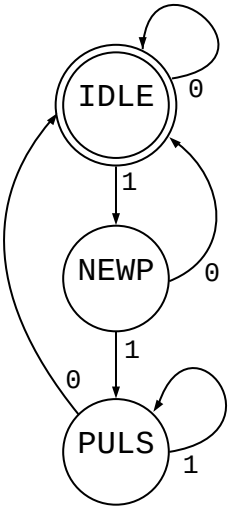
**Code 6.9 : Programma dat eigenschappen van cijfers afdrukt.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6    switch(atoi(argv[1])) {
7      case 1:
8      case 3:
9      case 5:
10     case 7:  printf("it is a prime\n");
11              printf("it is odd\n");
12              break;
13     case 2:  printf("it is a prime\n");
14     case 4:
15     case 6:
16     case 8:  printf("it is even\n");
17              break;
18     case 9:  printf("it is odd\n");
19              break;
20     default: printf("only digits 1 to 9 are allowed\n");
21   }
22
23   return 0;
24 }
```

Sommige programmeurs zweren bij een **switch**, andere gebruiken dit statement bijna nooit. Soms wordt er beweerd dat de **switch** sneller is. Anderen zeggen dat de **if-else-if** beter is. De verschillen zijn klein. In beide gevallen geldt in ieder geval dat een duidelijke opmaak van de code belangrijk is.




**Figuur 6.5 : De uitvoer van het programma van code 6.9.**




**Figuur 6.6 : Het toestandsdiagram van de pulsdetector.**

Men onderscheid twee soorten toestandsmachines: de Moore en de Mealy-machine. Dit is een Moore-machine. Bij een Mealy-machine is er toestand minder nodig.

Overigens had dit ook zonder toestandsmachine beschreven kunnen worden door steeds op deze conditie `(s[i]=='1') && (s[i-1]='0')` te testen.

**Code 6.10 : Voorbeeld van een toestandsmachine**

```
1  #include <stdio.h>
2
3  #define IDLE 0
4  #define NEWP 1
5  #define PULS 2
6
7  char s[]="000111000111011000011111110001000111110111000";
8
9  int main(void)
10 {
11   int i=0;
12   int state=IDLE;
13
14   while (s[i] != '\0') {
15     switch( state ) {
16       case IDLE: if (s[i]=='1') {
17                    state = NEWP;
18                  }
19                  break;
20       case NEWP: printf("At point %d a new puls is found\n", i);
21                  if (s[i]=='1') {
22                    state = PULS;
23                  } else {
24                    state = IDLE;
25                  }
26                  break;
27       case PULS: if (s[i]=='0') {
28                    state = IDLE;
29                  }
30     }
31     i++;
32   }
33
34   return 0;
35 }
```

Een toestandsmachine (*finite state machine*, FSM) is een model om het gedrag van een sequentieel systeem te beschrijven door middel van toestanden, toestandsovergangen en acties.

Het switch-statement wordt vooral gebruikt bij zogenoemde toestandsmachines (*state machines*). Code 6.10 analyseert een rij enen en nullen en detecteert dat de waarde van nul naar één gaat.

Deze code representeert het toestandsdiagram (*state diagram*) uit figuur 6.6. Er is een variabele `state`, die bijhoudt in welke toestand de machine zich bevindt. De pijlen in de figuur zijn de toestandsovergangen (*state transitions*). In code worden deze overgangen beschreven met de **`switch`** en een aantal **`if`**-statements. De machine zal in de toestand (*state*) `IDLE` blijven zolang de waarde van `s[i]` een `'0'` is. Alleen als `s[i]` de waarde `'1'` heeft, gaat de machine naar de volgende toestand `NEWP`. In deze toestand wordt een tekst afgedrukt dat er een nieuwe puls gevonden is. De machine is maar een periode in toestand `NEWP`. Als `s[i]` gelijk aan `'1'` is, gaat de machine naar de toestand `PULS` en anders gaat deze naar de toestand `IDLE`. De machine blijft in `PULS` zolang `s[i]` gelijk aan `'1'` is. Pas als deze `'0'` is, gaat de machine terug naar `IDLE`.

**Uitleg code 6.10 regel 3**
**`#define`**

De preprocessor verandert de code voordat deze gecompileerd wordt. De tekststring direct achter preprocessoropdracht **`#define`** representeert de code die achter de tekststring staat. In dit geval is `IDLE` de tekststring en `0` de code. Overal in deze code vervangt de preprocessor `IDLE` door een `0`, `NEWP` door een `1` en `PULS` door een `2`. Deze preprocessoropdrachten verbeteren de leesbaarheid en de onderhoudbaarheid van de code.
De **`#define`** kent ook parameters. Met een **`#define`** kunnen daarom macro's gemaakt worden. De preprocessor vult overal waar de macronaam staat de macrodefinitie in met op de juiste plaatsen de waarden van de parameters. De macro's `max` en `min` bepalen het maximum en het minimum van twee getallen:

```
#define max(a,b)  ((a)>(b)?(a):(b))
#define min(a,b)  ((a)<(b)?(a):(b))
```

Deze macro's worden als volgt aangeroepen:

```
maximum = max(7,3);
minimum = min(7,3);
```

Na afloop heeft `maximum` de waarde 7 en `minimum` de waarde 3.

**Regel 14**
**`while`**`()`
`{ }`

Het **`while`**-statement is een herhalingsopdracht. De statements tussen de accolades worden uitgevoerd zolang de conditie tussen de ronde haken waar is. In hoofdstuk 7 worden de **`while`**-lus en de andere herhalingsopdrachten besproken.

**Regel 31**
*`increment operator`*
`i++`
*`decrement operator`*
`i--`

De `++` en `--` operatoren verhogen of verlagen de waarde van een variabele met 1. Er zijn twee versies. Bij de *pre*-versie staat de operator voor de variabele en bij de *post*-versie staat de operator achter de variabele. De *post*-versie wordt het meest gebruikt:

```
i=43:
i++;
// i heeft nu de waarde 44
```

```
i=43:
i--;
// i heeft nu de waarde 42
```

De *pre*-versie doet in dit voorbeeld precies hetzelfde:

```
i=43:
++i;
// i heeft nu de waarde 44
```

```
i=43:
--i;
// i heeft nu de waarde 42
```

Bij *post*-versie wordt, in een samengestelde uitdrukking, eerst de uitdrukking uitgewerkt en daarna wordt de variabele verhoogd of verlaagd. Bij een *pre*-versie verandert eerst de variabele en wordt daarna de uitdrukking uitgewerkt.

```
int i = 43;
int s;
s = (i++ + 56);
// i is nu 44 en s is 99
```

```
int i = 43;
int s;
s = (++i + 56);
// i is nu 44 en s is 100
```

De *pre*-versie wordt weinig gebruikt en het is raadzaam om de *post*- en *pre* niet te veel door elkaar te gebruiken. Gebruik altijd de *post* en alleen de *pre*-versie wanneer dat per se noodzakelijk is.

## 6.7 De conditionele operator

Naast de **`if-else`** kent C een conditionele operator die precies hetzelfde doet als de **`if-else`**. Deze ternaire operator heeft drie operanden:

```
conditie ? resultaat als conditie waar is
         : resultaat als conditie niet waar is
```

De eerste operand is een conditie, die waar of niet waar kan zijn. De tweede operand is een uitdrukking, die het resultaat van de bewerking is als de conditie waar is, en de derde operand is een uitdrukking, die het resultaat van de bewerking is als de conditie niet waar is. De conditionele operator is nuttig bij macro's. De macro:

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

komt overeen met de functie:

```
int max(int a, int b)
{
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Een andere toepassing is het conditioneel afdrukken.

```
printf("This text has %d line%s.\n",
            numberoflines, (numberoflines == 1) ? "" : "s");
```

Als de tekst uit een enkele regel bestaat, wordt er dit afgedrukt:

```
This text has 1 line.
```

En als de tekst uit vier regels bestaat, wordt er een `s` achter `line` afgedrukt:

```
This text has 4 lines.
```

# 7

# Herhalingsopdrachten

Doelstelling

In dit hoofdstuk leer je wat herhalingslussen zijn, waar deze lussen voor dienen en hoe je deze gebruikt.

Onderwerpen

De behandelde onderwerpen zijn:

- De **for**-lus.
- De geneste **for**-lussen.
- De komma-operator.
- De **while**-lus.
- De **do**-**while**-lus.
- De **break**-statement en het **continue**-statement.

Voorbeeldprogramma's met herhalingslussen zijn:

- Berekenen tafels van vermenigvuldiging met behulp van een **for**-lus.
- Berekenen meerdere tafels van vermenigvuldiging met behulp van een **while**-lus.
- Bepalen of een getal een priemgetal is of niet met een **while**-lus en een **break**- statement.
- Afdrukken van ASCII-waarden met een **while**-lus en een **continue**-statement:

Naast de conditionele opdrachten kent elk softwaresysteem iteratieve of herhalingsopdrachten (*loop assignments*). Net als de meeste andere software talen onderscheid C drie soorten herhalingsopdrachten: de **for**-lus, de **while**-lus en de **do**-**while**. De **for**-lus wordt gebruikt bij een aftelbaar aantal herhalingen. De **while**-lus wordt gebruikt als het aantal herhalingen niet bekend is. De **do**-**while** wordt gebruikt bij een niet aftelbaar aantal herhalingen, die minimaal één keer uitgevoerd moeten worden.

## 7.1 De for-lus

Iteratie betekent herhaling.

Code 7.1 gebruikt een **for**-lus om een tafel van vermenigvuldiging af te drukken. Aan het programma wordt een getal meegegeven. De **for**-lus vermenigvuldigt dit getal eerst met 1 en drukt het resultaat af. Bij de volgende iteratie wordt de vermenigvuldiger met 1 opgehoogd tot 2, daarna tot 3 en dat gaat zo door totdat de waarde 10 is bereikt. Het programma drukt alleen de tafels van 1 tot en met 99 af. Als de invoer hier niet aan voldoet wordt er een waarschuwing afgedrukt.

**Code 7.1 : Voorbeeld met een for-lus: de tafels van vermenigvuldiging.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX_NUMBER   100
5  #define MSG_USAGE    "usage: %s <number>\n"
6  #define MSG_NUMBER   "       <number> must be greater than 0 and less than %d\n"
7
8  int main(int argc, char *argv[])
9  {
10   int number;
11   int i;
12
13   if ( argc < 2 ) {
14     printf(MSG_USAGE, argv[0]);
15     return 1;
16   }
17
18   if ( ((number = atoi(argv[1])) <= 0) || (number>=MAX_NUMBER) ) {
19     printf(MSG_USAGE, argv[0]);
20     printf(MSG_NUMBER, MAX_NUMBER);
21     return 1;
22   }
23
24   for (i=1; i<=10; i++) {
25     printf("%2d * %d = %3d\n", i, number, i*number);
26   }
27
28   return 0;
29 }
```

De syntax van de for-lus luidt als volgt:

```
for ( startconditie ; eindconditie; iteratie ) {
  statements
}
```

De `for`-lus is aftelbaar. Er is meestal een variabele die een startwaarde krijgt en daarna bij elke volgende stap met een vaste waarde verhoogd of verlaagd wordt totdat de eindconditie is bereikt. Bij elke stap worden de statements precies een keer uitgevoerd. Een voorbeeld van een for-lus is:

```
for (i=0; i<7; i++) {
  printf("The square of %d is %d\n", i, i*i);
}
```

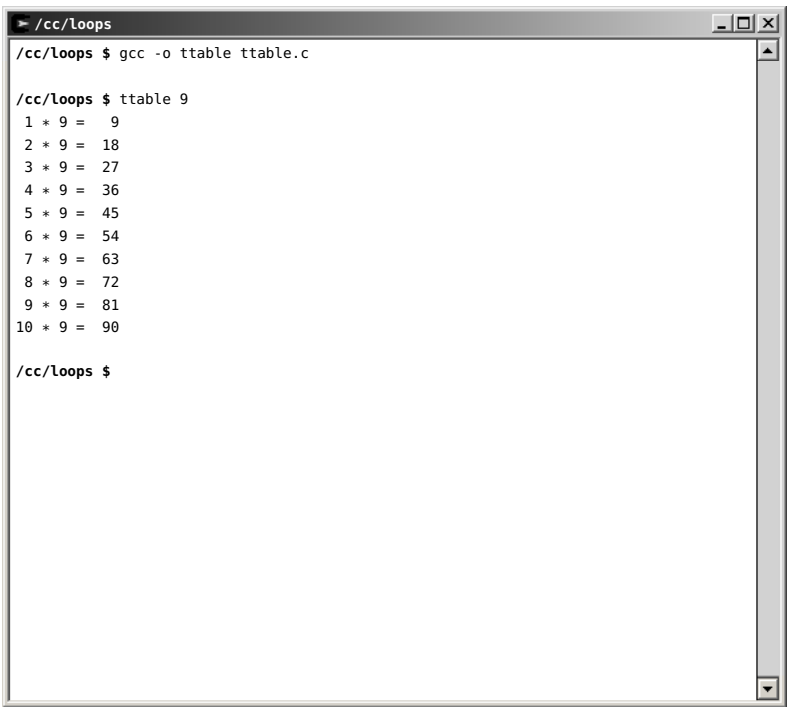
De lusvariabele, die vaak `i` genoemd wordt, start in dit voorbeeld met `0`. Daarna wordt het statement uitgevoerd en wordt `i` met één opgehoogd. Dit wordt herhaald totdat `i` de waarde 7 heeft bereikt. Elke beginwaarde, eindvoorwaarde en herhaling kan gebruikt worden. Dus dit is ook goed, al is het niet erg fraai:

```
for(x = 12 ; (x < f(x)) || (a < 7); b = b(x) ) {
  // statements
}
```

Er mogen ook delen weg worden gelaten. Zelfs alles mag worden weggelaten.

```
for (;;) {
  statements
}
```

Dit levert een oneindige lus op. Het programma blijft de statements uit de lus voortdurend uitvoeren.
In code 7.1 is `i` de lusvariabele. De beginwaarde van `i` is 1, de eindwaarde 10 en de stapgrootte 1. Het resultaat voor de tafel van negen staat in figuur 7.1.




**Figuur 7.1 : het programma van code 7.1 drukt de tafel van negen af.**

De lusvariabele moet natuurlijk gedeclareerd worden. Het is gebruikelijk daar `i` of een andere letter voor te gebruiken. Bij een **`for`** in een andere **`for`** wordt de lusvariabele van buiten naar binnen vaak aangeduid met achtereenvolgens `i`, `j`, `k`, `m` en `n`.

```
for (int i=1; i<10; i++) {
  printf("The times table of %d is:\n", i);
  for (int j=1; j<=10; j++) {
    printf("\t%d * %d = %d\n", i, j,  i*j);
  }
}
```

In bovenstaand fragment zijn de lusvariabelen, net als bij C++ en Java, lokaal gedeclareerd in de **`for`**-lussen. De compiler geeft — bij de instelling op GNU89 — een foutmelding. Met de compileroptie `-std=gnu99` worden lokaal gedeclareerde variabelen wel geaccepteerd. In oudere C-standaarden moeten variabelen altijd aan het begin van een functie worden gedeclareerd. Zolang GNU99 nog niet de standaardinstelling is, worden lokale lusvariabelen in C weinig gebruikt. Dit boek volgt GNU89 en declareert dus altijd alle lokale variabelen aan het begin van de betreffende functie.

## 7.2 De komma-operator

De onderstaande code rekent de som van `n` getallen uit:

```
sum=0;
for (i=0; i<n; i++) {
  sum = sum + i;
}
```

Dit kan met de komma-operator (,) geschreven worden als:

```
for ( sum-0,i=0; i<n; i++) sum=sum+i;
```

De initialisatie van de som is bij initialisatie van de lusvariabele geplaatst. Bovendien mag de sommatie ook in de **`for`** voor de iteratie worden neergezet. Als bovendien de verkorte schrijfwijze `sum += i` voor `sum = sum + i` wordt gebruikt, levert dat een zeer beknopt stuk code op:

```
for ( sum-0,i=0; i<n; sum+=i,i++) ;
```

Het statement achter de **`for`** is leeg. De leesbaarheid van dit soort constructies is vaak slecht. Daarom wordt dit soort constructies weinig gebruikt.

**Code 7.2 : Voorbeeld met een while-lus: meerdere tafels van vermenigvuldiging.**

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    int number;
6    int i;
7
8    printf("Enter number for times table or enter 0 to stop:");
9    scanf("%d", &number);
10
11   while ( number > 0 ) {
12     for (i=1; i<=10; i++) {
13       printf("%2d * %d = %2d\n", i, number, i*number);
14     }
15     printf("Next number or enter 0 to stop:");
16     scanf("%d", &number);
17   }
18
19   return 0;
20 }
```

## 7.3 De while-lus

In deze syntaxformulering staat achter de **`while`** een block statement. Dit mag ook een enkelvoudig statement zijn.

```
while (conditie)
  statement;
```

Voor de leesbaarheid en voor de duidelijkheid is het beter om altijd accolades te plaatsen.

De **`while`**-lus is geschikt om een taak een onbekend aantal keren uit te voeren. Code 7.2 vraagt steeds om een getal en drukt van dit getal de tafel van vermenigvuldiging af en blijft dit herhalen totdat de gebruiker `0` invoert. Van te voren is niet bekend hoe vaak de gebruiker dit zal doen. De **`while`**-lus is daarom heel geschikt voor deze herhaling. De syntax van de **`while`** is:

```
while ( conditie ) {
  statements
}
```

Zolang de conditie waar is, worden de statements tussen de accolades uitgevoerd.

Elke **for**-lus kan altijd als een **while**-lus worden geschreven. De syntax van de **for**-lus komt overeen met deze constructie:

```
startconditie
while ( eindconditie ) {
   statements
   iteratie
}
```

Het voorbeeld bij de uitleg over de **for**-lus luidt met een **while**-lus als volgt:

```
i=0;
while (i<7) {
  printf("The square of %d is %d\n", i, i*i);
  i++;
}
```

Ook bij de **while**-lus mogen onderdelen worden weggelaten. De oneindige **while**-lus ziet er zo uit:

```
while(1) {
   statements
}
```

Een microcontroller moet zijn taak meestal voortdurend blijven uitvoeren. In het hoofdprogramma van een microcontroller staat daarom meestal een oneindige **while**-lus. Alleen in het geval dat de spanning wegvalt of als er een hardwarematige reset of interrupt is, wordt het hoofdprogramma onderbroken.

## 7.4 De do-while-lus of do-lus

De **do**-**while** of **do** is een variant op de **while**. In plaats van te testen aan het begin van de lus, wordt bij de **do**-**while** de test aan het eind gedaan.

```
do {
  statements
} while ( conditie );
```

Zolang de conditie waar is worden de statements uitgevoerd. De **do**-**while** wordt gebruikt als het aantal herhalingen onbepaald is en de statements minimaal een keer moeten worden uitgevoerd. Net zoals de **for**-lus in een **while** kan worden omgezet, is een **for** ook te schrijven met een **do**-**while**:

De accolades zijn bij de **do**-**while** niet nodig. De **do** impliceert een bloktoewijzing en de **while** sluit deze impliciet af.
De accolades staan er voor de leesbaarheid.

```
startconditie
do {
  statements
  iteratie
} while ( conditie );
```

Het voorbeeld bij de uitleg over de **for**-lus is hier met een **do**-**while** opgelost:

```
i=0;
do {
   printf("The square of %d is %d\n", i, i*i);
   i++;
} while (i<7) ;
```

De **do**-**while** wordt veel minder vaak gebruikt dan de **while** en de **for**. De **for** is aantrekkelijk bij aftelbare situaties. De meeste programmeurs gebruiken meestal een **while** in plaats van een **do**-**while**. Waarschijnlijk omdat de **while** eerst test op de conditie en dan de statements uitvoert in plaats van andersom.

## 7.5 Het break-statement en het continue-statement

Het **break**-statement is al eerder besproken bij de **switch**, maar kan deze ook bij de **for**, **while** en **do-while** gebruikt worden om de lus voortijdig te verlaten. Code 7.3 drukt van een getal af of het een priemgetal is of niet. Het programma vraagt om

**Code 7.3 : Voorbeeld break-statement: afdrukken priemgetal of niet.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    int i, n, prime=1;
 6
 7    printf("Give a number : ");
 8    scanf("%d", &n);
 9    for( i=2; i<=n/2; i++) {
10      if( n % i == 0 ) {
11        prime = 0;
12        break;
13      }
14    }
15
16    printf("%d %s a prime.\n", n, prime ? "is" : "isn't" );
17
18    return 0;
19  }
```

Een priemgetal (*prime*) is een getal dat alleen door zichzelf en door 1 deelbaar is.

een getal en probeert dit getal te delen door alle getallen van 2 tot en met $\frac{1}{2}$n. Alle mogelijke priemgetallen liggen in dat bereik. Als een deeltal gevonden is, wordt met een **break** het zoeken gestaakt en wordt er afgedrukt dat het een priemgetal is. Als er geen deeltal is gevonden wordt er afgedrukt dat het geen priemgetal is. De `printf` gebruikt een conditionele operator om `is` of `isn't` af te drukken.

**Uitleg code 7.3 regel 10**
*modulo operator*
%

De *modulus operator* berekent de modulus van twee getallen. Bij C is dat de rest van de deling. Als de modulus nul is, is het deeltal deelbaar door de deler. Bij bijna alle computertalen wijkt de definitie van de modulus af van de wiskundige definitie. In veel gevallen — zo ook in C — is het de rest een geheeltallige deling. De functie had daarom beter de *remainder* in plaats van *modulo* genoemd kunnen worden. Deze macro rekent de echte modulus uit:

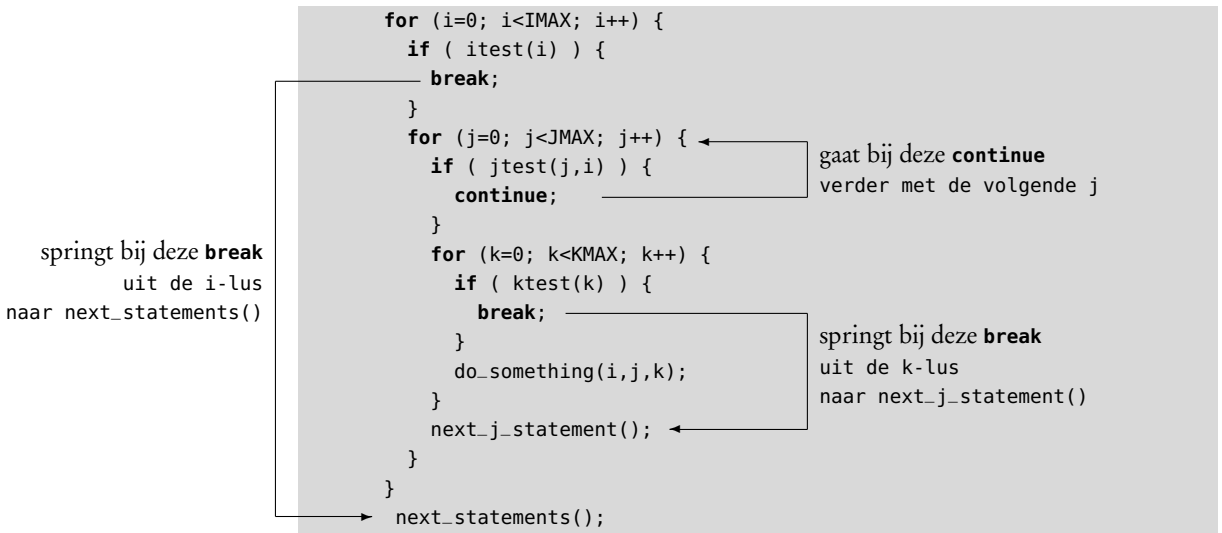
```
#define mod(x,y)   ( ((x)%(y)==0)?0:(((x)/(y))<0 )?((x)%(y))+(y):(x)%(y) )
```

Omdat voor positieve waarden van deler en deeltal de rest en de modulus hetzelfde zijn, is het meestal geen probleem om deze functie te gebruiken.

Het **continue**-statement lijkt op het **break**-statement; alleen stopt **continue** niet met de lus, maar breekt het de huidige iteratie af en gaat door met de volgende iteratie. Code 7.4 drukt de afdrukbare ASCII-waarden af op het scherm met het bijbehorende karakter. Als de waarde een controlteken is, wordt de huidige iteratie niet verder uitgevoerd. De `printf` wordt dan overgeslagen en er wordt verder gegaan met de volgende waarde.
Het voorwaardelijk afdrukken bij code 7.4 had net zo goed met een **if-else** gedaan kunnen worden. De vorm met de **continue** is te prefereren boven de **if-else** wanneer er meer statements staan dan een enkele afdrukopdracht.

**Code 7.4 : Voorbeeld continue-statement: afdrukken ASCII-waarden.**

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main (void)
5  {
6    int i;
7
8    for (i=0; i<128; i++) {
9      if ( iscntrl(i) ) {
10       continue;
11     }
12     printf("Character %c is ASCII value %d\n", i, i);
13   }
14
15   return 0;
16 }
```




**Figuur 7.2 : break en continue bij geneste lussen**

Figuur 7.2 laat zien dat bij geneste lussen de **continue** of **break** op de binnenste lus slaat waarbinnen deze gebruikt wordt. Moet er uit een diep geneste lus in één keer teruggesprongen worden, gebruik dan een functie met een **return** op de wijze zoals dat bij de functie `print_ctype` in code 6.5 is gedaan.

# 8

# Structuur en Opmaak

| | |
|---|---|
| Doelstelling | In dit hoofdstuk leer je hoe je leesbare C-code schrijft, op welke manieren je de code van commentaar kan voorzien, wanneer je commentaar gebruikt en wat goede namen voor *identifiers* zijn. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De algemene structuur van een programma.<br>▪ Het gebruik van commentaar: `/*`, `*/` en `//`.<br>▪ De opmaak van toewijzingen.<br>▪ Het gebruik van haakjes<br>▪ Het gebruik van spaties en lege regels.<br>▪ Het opmaakprogramma `indent`.<br>▪ De naamgeving van typen, constanten, variabelen en functies. |

Een programmeur moet de software zodanig coderen dat het voor hemzelf en voor andere programmeurs duidelijk is. Bij C is dit — vanwege de vele mogelijkheden en slimme constructies — nog belangrijker dan bij andere talen.
Bij grote projecten is het verstandig de code over verschillende bestanden te verdelen. Functies die bij elkaar horen, worden in een enkel bestand geplaatst. Er zijn bijvoorbeeld aparte bestanden: voor alle functies die bewerkingen doen op een bepaalde datastructuur, voor alle leesfuncties, voor alle schrijffuncties, voor een aantal generieke functies en voor de hoofdroutine. De code in de bestanden moet ook gestructureerd worden. De volgorde van de statements is meestal:

- een stuk commentaar met een korte omschrijving van het bestand
- de **`#include`**-opdrachten
- de **`#define`**-opdrachten
- de typedefinities
- de globale declaraties
- de prototypen
- de hoofdroutine `main`, althans als deze in dit bestand staat.
- de overige functies

## 8.1 Commentaar

Sommige programmeurs zetten heel veel commentaar in hun code. Anderen zijn daar zeer voorzichtig mee. Er is een aantal redenen om zuinig te zijn met commentaar. Ten eerste zal — mits de code goed geschreven is en de variabelen en functies goede, betekenisvolle namen hebben — de code zelfverklarend zijn. Ten tweede is de kans groot dat de uitleg van het commentaar afwijkt — of na een revisie gaat afwijken — van de feitelijke code. Onjuist commentaar is zeer verwarrend. Tenslotte verstoort commentaar de opmaak van de code en dat verslechtert de leesbaarheid.

Norm Schryer van AT&T Research, heeft ooit gesteld: "When the code and the comments disagree, both are probably wrong.".

Commentaar mag in geen geval gebruikt worden om slecht geschreven code te verklaren. Het is beter om in plaats daarvan de code te herschrijven zodat deze wel begrijpelijk is. Vaak is het al voldoende sommige delen een eigen functie te geven en betekenisvolle namen te gebruiken.

Commentaar hoort wel aan het begin van een bestand om uit te leggen wat de status is van het bestand. Voor een ingewikkelde functie kan met een stuk commentaar worden uitgelegd wat de functie doet, wat de parameters doen en wat de retourwaarde is. In een functie is commentaar vaak overbodig. Een enkele keer is het zinvol om een bepaalde bewerking of beslissing kort toe te lichten. In ieder geval is dit soort commentaar onzinnig:

```
i=i+1;              // add one to i

if ( test(i) ) {
  action();         /* does action() if test(i) is true */
}

y = sin(2*x)        //  y is sin(2x)
```

Oorspronkelijk kende C alleen `/*` en `*/` als commentaarblok. Later is daar, omdat C++ deze ook kent, `//` als commentaarregel bijgekomen. In ieder geval is `//` bij GNU89 toegestaan.

Het vorige voorbeeld toont twee verschillende soorten commentaar. Alles wat tussen de tekens `/*` en `*/` staat en alles wat tussen `//` en het einde van de regel staat, wordt door de compiler genegeerd. De `/*` en `*/` zijn handig om bijvoorbeeld aan het begin van een bestand een commentaarblok op te nemen:

```
/*
 *  Project    : mtb
 *  File       : port.c
 *  Version    : 1.40
 *  Author     : Wim Dolman
 *  Date       : september 2001
 *  Contents   : Contains all routines to manipulate the PORT list.
 */

#include <string.h>
#include <ctype.h>
#include "port.h"
     ⋮
```

Het `//`-commentaar is handig bij het becommentariëren van statements. Verder is het `//`-commentaar zeer praktisch tijdens het debuggen. Regels kunnen — tijdelijk — worden weggecommentarieerd, zodat de programmeur kan onderzoeken wat er gebeurt als een statement of functie weggelaten wordt.

## 8.2 Opmaak

Er zijn verschillende typografische mogelijkheden om de code beter leesbaar te maken, onder andere: het inspringen van tekst, het gebruik van extra spaties en het invoegen van witte regels en het beperken van de regellengte.
Door de tekst te laten inspringen, wordt de code duidelijker. In code 7.3 is met de gekozen opmaak direct zichtbaar dat de regels 11 en 12 bij de **`if`** van regel 10 horen. Voor het inspringen bestaat geen vaste norm. Een inspringing (*indentation*) van twee karakters levert een prima resultaat op.

Gebruik nooit de tabulatortoets. Programma's verwerken de tabulatie verschillend. Vroeg of laat levert dat problemen op.

Met lege regels worden belangrijke stukken code gescheiden. Een lege regel kan komen na de `include`'s, de `define`'s, de declaraties en na elke functie. In een functie kan een lege regel komen na de lokale declaratie, tussen **`while`**- en **`if`**-statements, en voor de laatste **`return`**.

Gebruik witte regels om stukken code te groeperen.

Een regel bevat nooit meer dan een toewijzing. In plaats van:

Plaats nooit meer dan een toewijzing op een regel.

```
a++; b++;
z = (c=a+b) + (d * e);
x1 = (-b + sqrt(D=b*b-4*a*c))/(2*a);
x2 = (-b - sqrt(D))/(2*a);
```

is het beter om dit op te schrijven:

```
a++;
b++;
c = a + b;
z = c + (d * e);
D = b*b-4*a*c;
x1 = (-b + sqrt(D))/(2*a);
x2 = (-b - sqrt(D))/(2*a);
```

Binnen een regel kunnen spaties de tekst op een logische wijze ordenen.

Maak regels niet te lang.

Te lange regels zijn niet goed leesbaar. Bovendien geven lange regels altijd problemen bij het afdrukken en bij het verder verwerken van stukken code in verslagen en andere documentatie. In dit document passen in de tekst maximaal 78 karakters en bij code met regelnummers zijn dat er slechts 72. Bij een volle paginabreedte is dat respectievelijk 106 en 100. Aanbevolen wordt regels niet langer te maken dan 72 karakters.

Breek regels op een logische plaats af, die past bij de betekenis.

Een regel moet worden afgebroken achter een komma of een operator. Een hoog niveau afbreking is beter dan een laag niveau afbreking. De tekst op de nieuwe regel komt direct onder een uitdrukking van hetzelfde niveau te staan. Als de tekst dan teveel aan de rechter kant komt, wordt er gewoon ingesprongen. Eventueel wordt de uitdrukking over meerdere regels verdeeld.

```
void function(int variable1, int variable2, int variable3,
                             int variable4, int variable5);

void function_with_a_really_long_long_name(int variable1,
       int variable2, int variable3, int variable4, int variable5);

if ( m && ((m[0] == '\0') ||
           (m[1] == '\0' && ((m[0] == '0') || (m[0] == '*')))) ) {

if ( longname &&
     ( (longname[0] == '\0') ||
       (longname[1] == '\0' &&
        ((longname[0] == '0') || (longname[0] == '*'))) ) ) {
```

Maak royaal gebruik van ronde haken bij bewerkingen.

Logische en rekenkundige bewerkingen kennen voorrangsregels (*precedence*). De vermenigvuldiging en de deling gaat voor de optelling en deze gaat weer voor de gelijkheidsoperator. Ronde haken hebben de hoogste prioriteit en maken de voorrang expliciet.

```
b = -a - 6*b < z + 4;           // without parenthesises
b = (((-a) - (6*b)) < (z + 4)); // equivalent, parenthesises showing the precedence
b = (-a - 6*b) < (z + 4);       // equivalent, with a few parenthesises to improve readability
```

De volgorde waarin operatoren geëvalueerd worden, ligt vast met de prioriteits- en associativiteitsregels. Deze regels staan in tabel 9.9 van paragraaf 9.13. Maar de volgorde waarin de operanden geëvalueerd worden, ligt niet vast. Er treden zogenoemde neveneffecten (*side effects*) op. Meer informatie hierover staat in paragraaf 9.13. Verschillende compilers kunnen daarom een ander resultaat geven. Hier staan twee voorbeelden waarbij het mis kan gaan:

```
z = f() + g();

printf("%d %d\n", ++n, x(n));
```

Ingewikkelde, slimme constructies hebben vaak vervelende neveneffecten.

Als het resultaat van de bewerking van functie `g()` afhangt van de uitkomst van functie `f()`, kan de volgorde ertoe doen. De compiler die eerst `f()` evalueert, geeft dan een ander resultaat dan een compiler die eerst `g()` evalueert. Ook de volgorde waarin de argumenten van een functie worden berekend ligt niet vast. Dit is een alternatief waarin beide problemen zich niet voordoen:

```
h1 = f();
h2 = g();
z  = h1 + h2;

++n;
printf("%d %d\n", n, x(n));
```

De functie `f` wordt nu altijd geëvalueerd voor de functie `g` en `n` wordt eerst opgehoogd en daarna wordt `x(n)` berekend.

Gebruik bij voorwaardelijke opdrachten en bij herhalingsopdrachten altijd accolades.

De accolades van blokstatements moeten op een logische, consistente en consequente manier worden geplaatst. Figuur 8.1 toont een aantal verschillende oplossingen. Het is zeer verwarrend als er meerdere schrijfwijzen door elkaar worden gebruikt.

Gebruik altijd een vaste opmaak.

Veel bedrijven hebben eigen regels voor het schrijven van code. Ook bij grote *open source* projecten zijn er vaak strenge richtlijnen. Bij GNU-projecten is dat vaak de GNU-stijl, die ook in figuur 8.1 te zien is. Het nadeel van deze stijl is dat er veel witte ruimte in de code staat. Voor het voorbeeld van figuur 8.1 zijn bij de GNU-stijl 22 regels en bij de andere twee stijlen 17 regels nodig. Er zijn ook programma's die de code automatisch de juiste opmaak geven. Bij de meeste GCC-distributies zit het programma `indent`. Het programma `indent` kent heel veel opties en mogelijkheden. De GNU-stijl uit figuur 8.1 is met dit programma met de optie `-gnu` gegenereerd. Code 8.1 is bijna onleesbaar. Met `indent` en een negental opties is de code omgezet naar de code 8.2, die wel een goed leesbare opmaak heeft.

```c
int
number_of_e (char *s)
{
  int i = 0;
  int n = 0;
  int m = 0;

  while (s[i] != '\0')
    {
      if (s[i] == 'e')
        {
          n++;
        }
      else if (s[i] == 'E')
        {
          m++;
        }
        i++;
    }

  return n + m;
}
```

```c
int number_of_e (char *s) {
  int  i = 0;
  int  n = 0;
  int  m = 0;

  while (s[i] != '\0') {
    if (s[i] == 'e') {
      n++;
    }
    else if (s[i] == 'E') {
      m++;
    }
    i++;
  }

  return n + m;
}
```

```c
int number_of_e(char *s)
{
  int i = 0;
  int n = 0;
  int m = 0;

  while (s[i] != '\0') {
    if (s[i] == 'e') {
      n++;
    } else if (s[i] == 'E') {
      m++;
    }
    i++;
  }

  return n + m;
}
```

**Figuur 8.1 :** **Dezelfde functie met drie verschillende layouts.** Links is de zogenoemde GNU-stijl gebruikt. In het midden staat een variant op de Kernighan&Ritchie-stijl. Rechts staat de stijl, die in dit boek is gebruikt. Deze stijl is compacter dan de GNU-stijl. Bij de functie staat de { op een nieuwe regel. Bij de **while** en de **if** staat de { er direct achter. De **else-if** staat direct achter de } van de voorafgaande **if**. Er staat geen spatie bij de functienaam en de parameterlijst en het returntype staat voor de functienaam.

## 8.3 Naamgeving

Definieer constanten met een **#define**. Bijvoorbeeld:

```c
#define PHI   1.618
#define EULER 2.71828
```

Typen, constanten, variabelen en functies moeten duidelijke namen krijgen. In C bestaat een naam van een identifier uit letters en cijfers. Het liggende streepje _ telt als letter. C maakt onderscheid tussen hoofd- en kleine letters. Een identifier begint altijd met een letter. Identifiers die met een _ beginnen, hebben een speciale betekenis binnen het compilersysteem. De lengte van de identifier is onbeperkt, met dien verstande dat vaak alleen de eerste 31 karakters significant zijn.
Constanten worden altijd met hoofdletters geschreven. Typedefinities beginnen vaak met een hoofdletter of bestaan helemaal uit hoofdletters. Korte variabele- en functienamen worden altijd met kleine letters geschreven. Er zijn twee conventies om lange variabele- en functienamen visueel op te splitsen: de delen van de naam beginnen met een hoofdletter of worden gescheiden door een liggend streepje:

```c
char *thisIsALongName;
char *this_is_a_long_name;
char *shortname;
```

Het nadeel van te lange namen is dat deze de opmaak van de code kunnen verstoren. Vooral bij indices van arrays is een korte naam — zelfs een naam met een enkele letter — beter.

```c
result = calculate(data1[i], data2[i], data3[i], data4[i], data5[i]);

result = calculate(data1[measure_index],
                   data2[measure_index], data3[measure_index],
                   data4[measure_index], data5[measure_index]);
```

**Code 8.1 : Code uit bestand v.c zonder opmaak.**

```
#include <stdio.h>

int main(int argc,
char *argv[])
{
int x=argv[1][0];
int a=0;int c;

while (x != 'c')
{
if (a%2==1) {c = 3;} else {         c=18;
}printf("%d\nNext: ", c);
scanf("%d\n", &x); a++;
}                return 0;  }
```

```
indent -i2 -br -ce -npsl -npcs -l75 -nut -sob -bap v.c
```

**Code 8.2 : Automatische opmaak met indent.**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int x = argv[1][0];
  int a = 0;
  int c;

  while (x != 'c') {
    if (a % 2 == 1) {
      c = 3;
    } else {
      c = 18;
    }
    printf("%d\nNext: ", c);
    scanf("%d\n", &x);
    a++;
  }
  return 0;
}
```

Er bestaan veel systemen voor documentatie van software. Het populaire programma `doxygen` is geschikt voor C++, C, Java en vele andere talen. Net als `Javadoc` extraheert het documentatie uit het commentaar van de broncode en verwerkt dat tot HTML, CHM, RTF, PDF of LaTeX.

Een aantal namen is gereserveerd en mag niet worden gebruikt als een eigen naam. Tabel 8.1 geeft een overzicht van deze gereserveerde namen (*keywords*). Bij sommige compilersystemen zijn er nog meer namen verboden.
Bij het schrijven van de code is het handig om een platte teksteditor te gebruiken, die de gereserveerde namen herkent en deze bijvoorbeeld een andere kleur geeft of automatisch vet afdrukt. Er zijn ook hulpmiddelen om code automatisch op te maken met de juiste syntaxkleuren of lettertypen. Met `source-highlite` kunnen allerlei talen — dus ook C — de juiste opmaak krijgen als de code wordt vertaald naar bijvoorbeeld HTML. Het tekstverwerkingspakket LaTeX kan de broncode automatisch vormgeven. Dit boek is gemaakt met LaTeX en voor de code is het style-bestand `listings` gebruikt. Alle gereserveerde namen zijn automatisch vet gezet.

**Tabel 8.1 : De gereserveerde namen in C.**

| | | |
|---|---|---|
| **auto** | **extern** | **switch** |
| **break** | **float** | **typedef** |
| **case** | **for** | **union** |
| **char** | **goto** | **unsigned** |
| **const** | **if** | **signed** |
| **continue** | **int** | **sizeof** |
| **default** | **long** | **static** |
| **do** | **register** | **void** |
| **double** | **return** | **volatile** |
| **else** | **short** | **while** |
| **enum** | **struct** | |

# 9

# Datatypen en Operatoren

Doelstelling

In dit hoofdstuk leer je wat datatypen zijn, welke datatypen de taal C kent en welke operatoren daarbij horen.

Onderwerpen

De behandelde onderwerpen zijn:

- De gehele getallen: **`int`**, **`char`**, **`short`**, **`long`**, **`unsigned`** en **`signed`**.
- De representatie van gehele en gebroken getallen in het geheugen.
- De gebroken getallen: **`float`**, **`double`** en **`long double`**.
- Typecasting.
- Geformatteerd afdrukken.
- Hexadecimaal, octaal en binair weergeven van gehele getallen.
- Rekenkundige operatoren.
- De relationele bewerkingen.
- Logische operatoren.
- Bitbewerkingen.
- Verkorte notatie bij bewerkingen.
- De bewerkingsvolgorde bij operatoren: de prioriteit en de associativiteit.

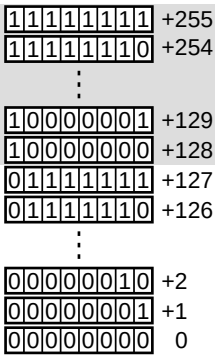
De voorbeelden demonstreren:

- Typecasting.
- Het gebruik van **`float`** bij het bepalen van de Quételet-index.
- Geformatteerd afdrukken.
- Het vergelijken van een **`double`** en een **`float`** met een constante.
- Het hexadecimaal en octaal van gehele getallen.
- Het binair afdrukken van gehele getallen.

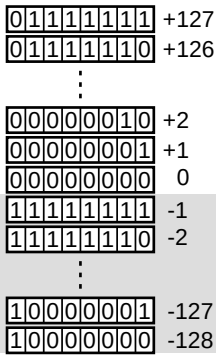
Getallen en andere informatie kunnen op vele manieren worden vastgelegd. Bij de representaties van getallen zijn er veel keuzes te maken. Gaat het om een geheel getal, een positief geheel getal of is het het gebroken getal en is het dan een zogenoemd *fixed point getal* of een *floating point* getal? Hoeveel bits zijn er beschikbaar om het getal vast te leggen? Bij de uitleg van code 3.2 zijn de vier standaard datatypen voor de representatie van getallen al genoemd: **`int`**, **`char`**, **`double`** en **`float`**.

## 9.1 Gehele getallen

De **char** wordt gebruikt voor het vastleggen van karakters. Het is een acht bits getal, dat in het algemeen een ASCII-waarde voorstelt. De waarde 65 is bijvoorbeeld het karakter `'A'` en 97 is een `'a'`.
Het type **int** definieert de gehele getallen. De grootte is compilerafhankelijk. Meestal is de **int** vier bytes groot en heeft een bereik van −2147483648 tot en met +2147483647. Bij een ATmega32 is **int** twee bytes groot en is het bereik −32768 tot 32767. Er zijn vier toevoegingen die de grootte of de representatie van **char** en **int** wijzigen of expliciet maken, namelijk: **short**, **long**, **unsigned** en **signed**. Met **short** en **long** wordt aangegeven dat er minder of meer geheugenruimte gebruikt kan worden. De toevoeging **unsigned** geeft aan dat in plaats van de two's complement representatie een unsigned binaire getalrepresentatie wordt gebruikt. Two's complement is de standaard representatie. Soms is het handig om dat expliciet aan te geven met **signed**.




**Figuur 9.1 : De binaire representatie van een unsigned char.**




**Figuur 9.2 : De two's complement representatie van een signed char.**

In figuur 9.1 en in figuur 9.2 staan de representaties van een **unsigned** en een **signed char**. Bij beide representaties gaat het om acht bits. De binaire representatie loopt van 0 tot 255 en de two's complement van −128 tot +127. Als het meest significante bit 1 is, is het two's complement getal negatief.
Tabel 9.1 geeft voor de gcc-compiler versie 3.4.4 alle mogelijke representaties van de gehele getallen met de bijbehorende grootte en bereik. De **int** is bij deze compiler net zo groot als een **long**, zodat er dus effectief maar vier verschillende groottes zijn: **char**, **short**, **int** en **long long**. In tabel 9.1 zijn de volledige namen gegeven. De niet vet gedrukte namen zijn optioneel. Zo wordt het type **signed short int** ook geschreven als: **signed short**, **short int** of **short**.
Het bestand `limits.h` bevat een aantal constanten (**#define**'s) voor de uiterste waarden van deze typen, bijvoorbeeld `UINT_MIN`, `UINT_MAX` voor het minimum en het maximum van een **unsigned int**.

## 9.2 Typecasting bij gehele getallen

De taal C is niet streng getypeerd. Een variabele van een bepaald type kan vaak direct aan een variabele van een ander type worden toegekend. In het volgende voorbeeld krijgt de integer `i1` de waarde van de **char**-variabele `c1`. Omdat een integer groter is dan een **char** past dit altijd en zijn de numerieke waarden van `c1` en `i1` hetzelfde.

**Tabel 9.1 :** **De representaties van gehele getallen bij de Cygwin gcc-compiler.** In de linker kolom staan de typen. De niet vet gedrukte namen zijn optioneel. De tweede kolom geeft de grootte van het getal in bytes en de derde kolom geeft de grootte in bits. Kolom vier en vijf geven het minimum en het maximum van het bereik.

| type | bytes | bits | minimum | maximum |
|---|---|---|---|---|
| *signed* **char** | 1 | 8 | -128 | +127 |
| **unsigned char** | 1 | 8 | 0 | +255 |
| *signed* **short** *int* | 2 | 16 | -32768 | +32767 |
| **unsigned short** *int* | 2 | 16 | 0 | +65535 |
| *signed* **int** | 4 | 32 | -2147483648 | +2147483647 |
| **unsigned** *int* | 4 | 32 | 0 | +4294967295 |
| *signed* **long** *int* | 4 | 32 | -2147483648 | +2147483647 |
| **unsigned long** *int* | 4 | 32 | 0 | +4294967295 |
| *signed* **long long** *int* | 8 | 64 | -9223372036854775808 | +9223372036854775807 |
| **unsigned long long** *int* | 8 | 64 | 0 | +18446744073709551615 |

```
char c1 = 'a';   // c1 is 97   (01100001)
int  i1;

i1 = c1;         // i1 is 97   (00000000000000000000000001100001)
```

Een integer toekennen aan een variabele van het type **char** kan ook, alleen heeft een integer meer bits dan de acht bits van een **char**. Alleen voor waarden tussen de 0 en 127 is de waarde van de **char**-variabele dan hetzelfde als de integer:

```
int  i2 = 97;   // 00000000000000000000000001100001
int  i3 = 2008; // 00000000000000000000011111011000
char c2, c3;

c2 = i2;        // c2 is 97, is character 'a' (01100001)
c3 = i3;        // c2 is -40                  (11011000)
```

Bij de aanroep van een functie vindt vaak een typeconversie plaats. De functie `getchar` retourneert een variabele van het type **int**. Bij de toekenning aan `c` wordt deze integer een **char**:

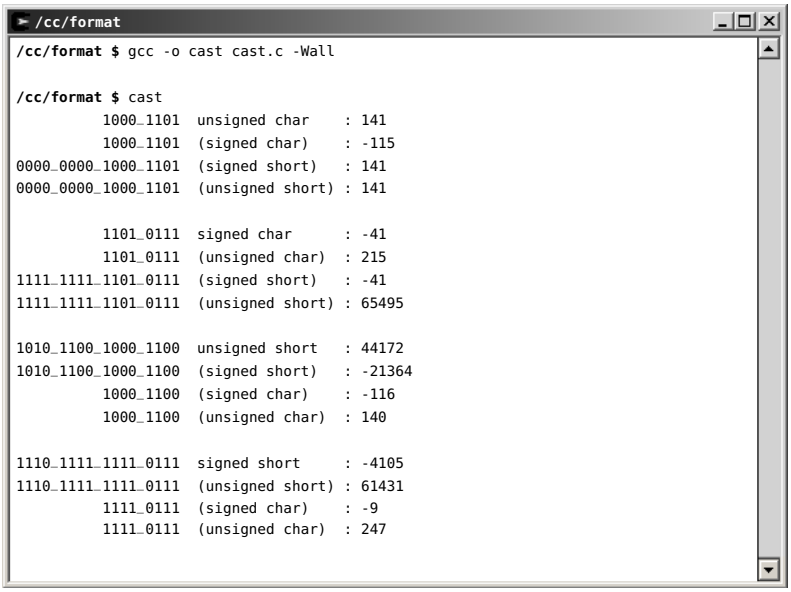
```
int getchar(void);

char c = getchar();
```

Een andere methode is om expliciete typeconversie toe te passen door het type voor de variabele tussen ronde haakjes te zetten.

```
int i;

x = (unsigned char) i;
```

Deze expliciete conversie wordt *typecasting* genoemd en (**unsigned char**) wordt de *cast* of *cast operator* genoemd.

De uitvoer van figuur 9.3 laat het effect van typecasting zien bij een **unsigned char**, een **signed char**, een **unsigned short** en een **signed short**. De programma code van deze uitvoer staat in code 9.1. Eerst wordt een **unsigned char** afgedrukt. De waarde van 141 valt buiten het bereik van de **signed char**. De acht bits `1000_1101` worden als −115 geïnterpreteerd. Bij de typecasting van een **unsigned char** naar de **unsigned short** en de **signed short** wordt het getal met nullen uitgebreid.
Daarna volgt een **signed char**. De waarde van −41 valt buiten het bereik van de **unsigned char**. De bits `1101_0111` worden als 215 geïnterpreteerd. Bij de typecasting




**Figuur 9.3** : **Typecasting bij gehele getallen.** Een `unsigned char`, `signed char`, `unsigned short` en `signed short` getal zijn steeds afgedrukt met drie andere typecasts.

van een **`signed char`** naar de **`unsigned short`** en de **`signed short`** wordt het getal met enen uitgebreid.
De derde groep drukt eerst een **`unsigned short`** af. De waarde van 44172 valt buiten het bereik van de **`unsigned char`**. De bits `1010_1100_1000_1100` worden als −21364 geïnterpreteerd. Bij de typecasting van een **`unsigned short`** naar de **`unsigned char`** of de **`signed char`** worden alleen de minst significante acht bits gebruikt.
De laatste groep drukt eerst een **`signed short`** af. De waarde van −4105 valt buiten het bereik van de **`unsigned char`**. De bits `1010_1100_1000_1100` worden als 61431 geïnterpreteerd. Bij de typecasting van een **`signed short`** naar de **`unsigned char`** of de **`signed char`** worden alleen de minst significante acht bits gebruikt.

Dezelfde effecten treden op bij typecasting tussen **`char`**, **`short`**, **`int`** en **`long`**. De uitvoer van figuur 9.3 laat zien dat bij de typecasting van gehele getallen feitelijk niets aan de bits verandert. Er worden hooguit bits weggegooid of toegevoegd. Wel is de interpretatie van de bits anders.
Uitbreiden van een **`unsigned`** of **`signed`** verandert de interpretatie niet. Inkorten van een getal verandert meestal het getal. Overstappen van een **`unsigned`** naar een **`signed`** of omgekeerd, verandert ook meestal het getal.

De programmacode waarmee de uitvoer van figuur 9.3 is gemaakt, staat in code 9.1. Op regel 5 tot en met 10 staat een functie `print` die de binaire waarde en de decimale waarde van een integer `v` afdrukt. De variabele `nbytes` bevat het aantal bytes dat `v` bevat. De variabele `type` wijst naar een string met het type dat wordt afgedrukt en `s` wijst naar een string met extra opmaak om de uitvoer te verfraaien.
De functie `print` gebruikt op regel 8 de functie `printb` uit code 9.6 om de binaire waarde van `v` af te drukken. Het prototype van `printb` staat op regel 8.
Vanaf regel 19 wordt de functie `print` zestien keer aangeroepen met steeds een ander getal of een andere typecasting.

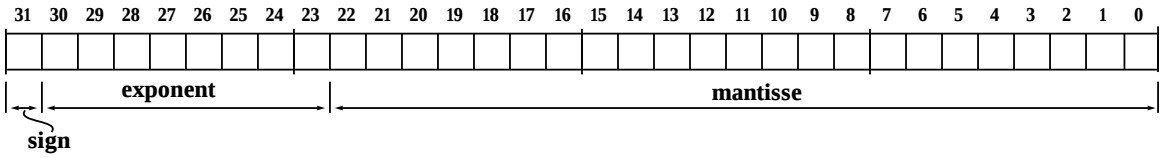
**Code 9.1 : Het effect van de cast-operatoren bij gehele getallen.**

```
 1  #include <stdio.h>
 2
 3  void printb(long long int x, int nbytes, int space);
 4
 5  void print(int v, int nbytes, char *type, char *s)
 6  {
 7    printf("%s", s);
 8    printb(v, nbytes, 1);
 9    printf("  %-17s: %d\n", type, v);
10  }
11
12  int main(void)
13  {
14    unsigned char   uc = 141;
15    signed char     sc = -41;
16    unsigned short  us = 4073;
17    signed short    ss = -4105;
18
19    print(                 uc, 1, "unsigned char",    "          ");
20    print((signed char)    uc, 1, "(signed char)",    "          ");
21    print((signed short)   uc, 2, "(signed short)",   "");
22    print((unsigned short) uc, 2, "(unsigned short)", "");
23    print(                 sc, 1, "signed char",      "\n          ");
24    print((unsigned char)  sc, 1, "(unsigned char)",  "          ");
25    print((signed short)   sc, 2, "(signed short)",   "");
26    print((unsigned short) sc, 2, "(unsigned short)", "");
27    print(                 us, 2, "unsigned short",   "\n");
28    print((signed short)   us, 2, "(signed short)",   "");
29    print((signed char)    us, 1, "(signed char)",    "          ");
30    print((unsigned char)  us, 1, "(unsigned char)",  "          ");
31    print(                 ss, 2, "signed short",     "\n");
32    print((unsigned short) ss, 2, "(unsigned short)", "");
33    print((signed char)    ss, 1, "(signed char)",    "          ");
34    print((unsigned char)  ss, 1, "(unsigned char)",  "          ");
35
36    return 0;
37  }
```

## 9.3 Gebroken getallen

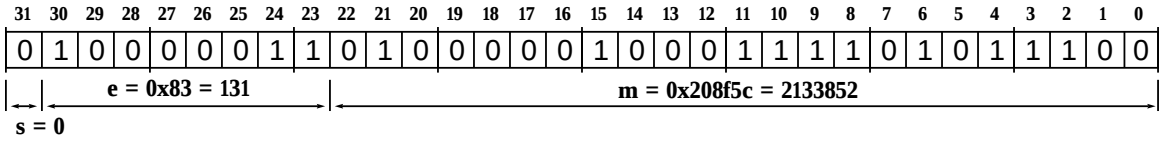
Gebroken getallen zijn getallen die niet geheel zijn, zoals 1,23; 50,6; 0,00338; 6,0 en 4,83 $10^{13}$. Voor gebroken getallen of drijvende komma getallen bestaan er drie typen: **`float`**, **`double`** en **`long double`**. Bij `gcc` neemt een **`float`** vier bytes ruimte in beslag, een **`double`** acht bytes en een **`long double`** tien bytes. Het bereik en de nauwkeurigheid van een **`long double`** is groter dan van een **`double`** en bij een **`float`** kleiner. Bij een kleine microcontroller, zoals de ATmega32, is een **`double`** vier bytes groot, net als een **`float`**.

Gebroken getallen worden opgeslagen volgens het IEEE754 Single Precision Format: het meest significante bit is het tekenbit (*sign*), daarnaast zijn een aantal bits gereserveerd voor de exponent en de rest is de mantisse. Figuur 9.4 toont de verdeling van de 32 bits bij de **`float`**. De exponent bepaalt het bereik en de mantissa




**Figuur 9.4 :** **De 32 bits bij een float.** De **`float`** bestaat uit vier bytes: het eerste bit is het teken (*sign*), de volgende acht bits geven de exponent en de laatste 23 bits vormen de mantisse.

de nauwkeurigheid. Een exponent van 8 bits geeft dat de kleinste waarde ongeveer 1,17 $10^{-38}$ en de grootste waarde ongeveer 3,40 $10^{+38}$ is. Een mantisse van 23 bits betekent dat de zevende decimaal kan afwijken. De nauwkeurigheid van een **`float`** is zes decimalen. Figuur 9.5 laat de bits zien voor het getal 20,07.




**Figuur 9.5 :** **De representatie als `float` van de constante 20,07.** Dit is slechts een benadering. De exacte waarde van deze reeks enen en nullen is 20,0699997.

Tabel 9.2 geeft overzicht van de mogelijke typen voor gebroken getallen bij de `gcc`-compiler. Kleinere microcontrollers kennen vaak alleen de **`float`** en bovendien is er meestal een speciale bibliotheek voor dit type. De exponent bepaalt het bereik van de gebroken getallen. Het bestand `floats.h` bevat een aantal constanten (**`#define`**'s) voor de uiterste waarden van deze typen, bijvoorbeeld `FLT_MIN`, `FLT_MAX` voor het minimum en het maximum van een **`float`**. De mantisse bepaalt de nauwkeurigheid. Een 1-bit fout geeft in het voorbeeld van figuur 9.5 een fout van plusminus 0,0000019:

```
0x208f5b   20,0700016
0x208f5c   20,0699997
0x208f5d   20,0699978
```

De meeste reële getallen kunnen dus niet exact met **`float`** of **`double`** worden gerepresenteerd. De programmeur zal daar voortdurend rekening mee moeten houden.

Handmatig de bits converteren naar een **`float`** of omgekeerd is lastig. De formule om uit de bits een **`float`** te berekenen luidt:

$$(-1)^s \times 2^{e-127} \times (1 + m * 2^{-23})$$

In figuur 9.5 is $s = 0$, $e = 131$ en $m = 2133852$, zodat:

$$1 \times 2^4 \times (1 + 2133852 * 2^{-23})$$

en hier komt 20.0699997 uit.

**Tabel 9.2 :** **De representatie van gebroken getallen bij de Cygwin `gcc`-compiler.** Alle drie de typen hebben naast een exponent en een mantisse een tekenbit. Het minimum em maximum zijn afgeronde waarden.

| type | bits | exponent | mantisse | nauwkeurigheid | minimum | maximum |
|---|---|---|---|---|---|---|
| **`float`** | 32 | 8 | 23 | 6 decimalen | `1.175e-38` | `3.402e+38` |
| **`double`** | 64 | 11 | 52 | 15 decimalen | `2.225e-308` | `1.797e+308` |
| **`long double`** | 80 | 15 | 64 | 18 decimalen | `3.362e-4932` | `1.189e+4932` |

**Code 9.2 : De bepaling van de Quételet-index**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MSG_USAGE  "usage: %s <weight in kg> <length in cm>\n"
5  #define MSG_WEIGHT "       <weight> must be greater than 0\n"
6  #define MSG_LENGTH "       <length> must be greater than 0\n"
7
8  int main(int argc, char *argv[])
9  {
10   int weight, length;
11   float qindex;
12
13   if ( argc < 3 ) {
14     printf(MSG_USAGE, argv[0]);
15     return 1;
16   }
17
18   if ( (weight = atoi(argv[1])) <= 0) {
19     printf(MSG_USAGE, argv[0]);
20     printf(MSG_WEIGHT);
21     return 1;
22   }
23
24   if ( (length = atoi(argv[2])) <= 0) {
25     printf(MSG_USAGE, argv[0]);
26     printf(MSG_LENGTH);
27     return 1;
28   }
29
30   qindex = (float) weight*10000/(length*length);
31
32   printf("Your quetelet index (body mass index) is %3.1f", qindex);
33
34   return 0;
35 }
```

**Figuur 9.6 : Adolphe Quételet** is een Vlaams wiskundige uit het begin van de negentiende eeuw, die veel statistisch onderzoek deed onder de bevolking. Hij wordt beschouwd als de grondlegger van de moderne statistiek.

## 9.4 Typecasting bij gebroken getallen

Het rekenen met drijvende komma of gebroken getallen moet zorgvuldig gedaan worden. In code 9.2 wordt de Quételet-index van een persoon bepaald. Deze index wordt ook wel de Body Mass Index genoemd. Dit is het gewicht in kilogrammen gedeeld door de lengte in centimeters in het kwadraat. Het programma leest twee gehele getallen `weight` en `length` en berekent daaruit het gebroken getal `qindex`. Voor een gewicht van 73 kilo en een lengte van 185 centimeter komt daar 21,3 uit. Als in plaats van regel 30 dit was genoteerd:

```
      qindex = weight*10000/(length*length);
```

dan was het antwoord 21,0 geweest. De uitkomst van het rechterlid is namelijk een integer. De variabele `weight` is een integer, `length` is een integer en `10000` is een

integer dus zal de uitkomst ook een integer zijn. Deze integer (21) wordt aan het gebroken getal `qindex` toegekend. Dus wordt er 21,0 afgedrukt.
Door voor de bewerking `(float)` te zetten wordt er met gebroken getallen gerekend. Uit de berekening komt dan 21,3294375 en wordt er door het programma 21,3 afgedrukt.

Het omzetten van **`int`** naar **`char`** en omgekeerd is het alleen toevoegen of verwijderen van bits. Het omzetten van **`int`**'s naar **`float`**'s en omgekeerd zijn complexe bewerkingen.

De bewerking `(float)` wordt een *cast* of *cast operator* genoemd. Met een cast wordt bijvoorbeeld een **`int`** als **`char`** geïnterpreteerd, een **`char`** als een **`int`**, een **`float`** als een **`int`**, een **`int`** als een **`float`**, en zo verder. Dit wordt gedaan door tussen ronde haakjes voor de variabele het alternatieve type neer te zetten. De variabele zelf verandert niet. Ook het type van de variabele verandert niet. De variabele wordt alleen anders geïnterpreteerd. Vooral bij berekeningen met gehele getallen en gebroken getallen is deze *typecasting* of typeconversie handig.

**Code 9.3 : Voorbeeld typecasting.**

```
#include <stdio.h>

int main(void)
{
  float a;

  a = 10/3;
  printf("%.12f\n",  a);

  a = (float) 10/3;
  printf("%.12f\n",a);

  a = 10%3;
  printf("%.12f\n", a);

  a = (float) 10/3;
  printf("%d\n", a);

  a = (float) 10/3;
  printf("%d\n", (int) a);

  a = 10%3;
  printf("%d\n", (int) a);

  a = 10%3;
  printf("%d\n", a);

  return 0;
}
```

De deling `10/3` is een deling van twee gehele getallen. De uitkomst is dan ook een geheel getal. Dit wordt automatisch afgerond op `3` en levert `3.000000000000` op.

De deling `(float)10/3` is een deling van een gebroken en een geheel getal. De uitkomst is dan een gebroken getal. Dit levert `3.333333253860` op.

De bewerking `10%3` geeft de rest van de geheeltallige deling. Dat is in dit geval `1` en wordt als `1.000000000000` afgedrukt.

Nu wordt de uitkomst van `(float)10/3` als geheel getal afgedrukt. Dat geeft onzin. Compileren met optie `-Wall` geeft een waarschuwing: *int format, double arg (arg2).*

Door de cast `(int)` wordt `a` geïnterpreteerd als integer en wordt er `3` afgedrukt.

Door de cast `(int)` wordt `a` geïnterpreteerd als integer en wordt er `1` afgedrukt.

De uitkomst van `a` is gelijk aan `1.0`. Dit wordt als geheel getal afgedrukt. Dat geeft onzin. In dit geval is dat `0`.
Compileren met optie `-Wall` geeft een waarschuwing: *int format, double arg (arg2).*

**Uitleg code 9.2 regel 32**
`printf`
*`format specifiers`*

Door extra tekens tussen het procentteken (%) en de conversieletter van een *format specifier* op te nemen, wordt het formaat aangepast. De opties worden gegroepeerd in vier groepen: *flags*, *fieldwidth*, *precision* en *modifier*. Deze vier mogelijkheden mogen worden gecombineerd. De syntax van een *format specifier* is:

```
%[flags][fieldwidth][.precision][modifier]conversioncharacter
```

Tabel 9.3 geeft alle opties voor de vier groepen en figuur 9.7 laat een aantal mogelijkheden zien. In alle boeken over C wordt deze optie meer of minder uitgebreid

besproken. Binnen de context van dit boek — microcontrollers — zijn deze opties minder relevant. Als de C-bibliotheek van een microcontroller al met *format specifiers* overweg kan, dan is dat vaak zonder allerlei opties of met een beperkt aantal opties.

**Tabel 9.3 : Speciale tekens voor format specifiers.** Er zijn vier soorten tekens die tussen de `%` en de conversieletter van de *format specifier* geplaatst kunnen worden.

| | | |
|---|---|---|
| *flags* | `-` | Er wordt links uitgelijnd. |
| | `+` | Het plusteken wordt afgedrukt. |
| | *space* | Er wordt een spatie op de plaats van het plusteken afgedrukt. |
| | `0` | Getal wordt links aangevuld met nullen. |
| | `#` | Geeft een alternatieve uitvoer, zoals `0x` en `0X` bij `x` en `X`, en een decimale punt bij `e` en `E`. |
| *fieldwidth* | *number* | Geeft de minimale breedte van de af te drukken tekst. Als de af te drukken tekst breder is, bepaalt dit getal het aantal af te drukken karakters. |
| | `*` | Dit is een plaatsvervanger. Het volgende getal geeft de breedte. |
| *precision* | *number* | Is de nauwkeurigheid bij `e`, `f` en `E` en het aantal significante cijfers bij `g` en `G`. |
| | `*` | Dit is een plaatsvervanger. Het volgende getal geeft de precisie. |
| *modifier* | `l` | Drukt **`int`** als **`long`** af. |
| | `ll` | Drukt **`int`** als **`long long`** af. |
| | `L` | Drukt **`double`** als **`long double`** af. |
| | `h` | Drukt **`int`** of **`char`** als **`short`** af. |

```
/cc/format
/cc/format $ gcc -o fmt fmt.c -Wall

/cc/format $ fmt
| Hello|   Hel|Hello |Hel     |
|   143|   143|143   |  000143|
|  +143|143   |000143|000143  |
|    8f|8F    |  0x8f|0X8F    |
|+2.01e+01||2.007e+01|2.007e+01 |+2.007000e+01|
| 2.01E+01||2.007E+01|20.1      |+20.07|
|    +20.07|     20.1|20.1      |+20.07|
| +2.01e+46|    2E+46|2.01E+46  |+2.007e+46|
| +2.01e+46| 2.01e+46|2.007e+46 |+2.007000E+46|
```

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char s[]="Hello";
6    int  i=143;
7    float f=20.07;
8    double d=20.07e+45;
9
10   printf("|%6s|%6.3s|%-6s|%-8.3s|\n"    ,s,s,s,s);
11   printf("|%6d|%6.3d|%-6d|%8.6d|\n"     ,i,i,i,i);
12   printf("|%+6d|%-6.3d|%-.6d|%-8.6d|\n" ,i,i,i,i);
13   printf("|%6x|%-6X|%#6x|%-#8X|\n"      ,i,i,i,i);
14   printf("|%+3.2e||%6.3e|%-10.3e|%+6e|\n",f,f,f,f);
15   printf("|% 3.2E||%6.3E|%-10.3G|%+6G|\n",f,f,f,f);
16   printf("|%+10.4g|%9.3g|%-10.3g|%+6g|\n",f,f,f,f);
17   printf("|%+10.3g|%9.2G|%-10.3G|%+6g|\n",d,d,d,d);
18   printf("|%+10.2e|%9.2e|%-10.3e|%+2E|\n",d,d,d,d);
19
20   return 0;
21 }
```

**Figuur 9.7 : Geformatteerd afdrukken.** Nevenstaande code geeft een aantal verschillende afdrukmogelijkheden en toont de schermafdruk met het resultaat.

## 9.5 Constanten bij gebroken getallen

Omdat gebroken getallen of drijvende komma getallen, een beperkt aantal bits bevatten, is de nauwkeurigheid beperkt. Bij het vergelijken (==, !=) kan dat problemen geven.
In code 9.4 is op de regels 8 en 9 het getal `3.1` een constante. Dit gebroken getal wordt in C als een **`double`** geïnterpreteerd. De afdrukopdrachten laten zien dat de waarde van `f` een beetje kleiner is dan 3,1 en dat `g` een heel klein beetje groter is dan 3,1. Bij de vergelijking `f==3.1` van regel 8 is `f` een **`float`** en `3.1` een **`double`**. Beide waarden zijn niet exact 3,1 en ongelijk aan elkaar, dus is de vergelijking niet waar en wordt er `0` afgedrukt. Bij de vergelijking `g==3.1` van regel 8 zijn `g` en `3.1` allebei van het **`double`**. Beide waarden zijn niet exact 3.1, maar ze zijn wel hetzelfde, dus is de vergelijking waar en wordt er 1 afgedrukt.

**Code 9.4 : Het vergelijken van een double en een float met een constante.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5     float  f=3.1;
 6     double g=3.1;
 7
 8     printf("%.18f  %d\n", f, f==3.1);         // print 3.099999904632568359  0
 9     printf("%.18f  %d\n", g, g==3.1);         // print 3.100000000000000089  1
10
11     printf("%.18f  %d\n", f, f==(float) 3.1); // print 3.099999904632568359  1
12
13     return 0;
14  }
```

Op regel 11 wordt `3.1` ook met `f` vergeleken. Alleen is bij `3.1` met een expliciete typecasting aangegeven dat deze constante als een **`float`** geïnterpreteerd moet worden. Beide waarden zijn nu ook hetzelfde, dus is de vergelijking ook waar en wordt er 1 afgedrukt.

Gebruik bij gebroken getallen altijd **`double`**. Gebruik alleen **`float`** bij de kleinere microcontrollers.

Om verschillende redenen is het verstandig om altijd **`double`** te gebruiken. Ten eerste gaat het vergelijken met constanten dan altijd goed. Ten tweede zijn veel mathematische functies met **`double`** geschreven en retourneren ook een **`double`**. Ten derde is de typeconversie tussen **`float`** en **`double`** geen eenvoudige bewerking en kost het rekenkracht. Alleen bij C-compilers voor eenvoudige microcontrollers, zoals WinAVR voor de ATmega32, wordt **`float`** gebruikt, omdat een **`double`** daar ook maar 32 bit is.

## 9.6 Hexadecimaal, octaal en binair

Gehele getallen kunnen in C decimaal, octaal of hexadecimaal worden weergegeven. Een prefix `0x` of `0X` geeft aan dat het een hexadecimaal getal is en de prefix `0` geeft aan dat het een octaal getal is. Code 9.5 geeft een voorbeeld.
De ASCII-waarde van de letter `m` wordt in hexadecimale notatie aan de variabele `hex` en in octale notatie aan de variabele `oct` toegekend. Daarna worden beide variabelen decimaal, hexadecimaal en octaal afgedrukt samen met de karakterrepresentatie van deze getallen. De vlag `#` bij de `%#x` en `%#o` bij de afdrukopdrachten zorgt ervoor dat de hexadecimale en octale waarden een prefix krijgen.

**Code 9.5 : Voorbeeld met hexadecimale en octale getallen.**

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int hex=0x6d;
6    int oct=0155;
7
8    printf("%3d\n", hex);   // print 109
9    printf("%#x\n", hex);   // print 0x6d
10   printf("%#o\n", hex);   // print 0155
11   printf("%c\n",  hex);   // print m
12   printf("%3d\n", oct);   // print 109
13   printf("%#x\n", oct);   // print 0x6d
14   printf("%#o\n", oct);   // print 0155
15   printf("%c\n",  oct);   // print m
16
17   return 0;
18 }
```

**Code 9.6 : Functie die gehele getallen binair afdrukt.**

```
1  void printb(long long int x, int nbytes, int space)
2  {
3    int i;
4    int nbits = 8*nbytes;
5    long long int mask = (1ULL << (nbits-1));
6
7    for(i=0; i<nbits; i++) {
8      if ( x & mask ) {
9        putchar('1');
10     } else {
11       putchar('0');
12     }
13     if ( space && ((i+1)%4 == 0) && ((i+1)<nbits) ) {
14       putchar('_');
15     }
16     x <<= 1;
17   }
18 }
```

Standaard is er geen *format specifier* voor het binair afdrukken. Een `%b` ontbreekt. In code 9.6 staat een functie `printb`, die gehele getallen binair afdrukt. Deze functie is in code 9.1 gebruikt.
Met een **`for`**-lus worden alle bits vanaf het meest significante bit een voor een geëvalueerd. Als het bit hoog is, wordt er een 1 en als het bit laag is, wordt er een 0 afgedrukt. Als de variabele `space` hoog is, wordt er tussen vier bits een lage streep (`'_'`) afgedrukt.

**Uitleg code 9.6 regel 1**
**`long long`**

Doordat de parameter `x` van het type **`long long`** is, kan de functie voor alle soorten gehele tallen gebruikt worden. Als het type **`unsigned`** is wordt het getal met nullen uitgebreid en met enen als het **`signed`** is. Deze uitbreiding wordt niet afge-

drukt, mits het aantal bytes `nbytes` correct is opgegeven. Dit kan met de **`sizeof()`** operator, die de grootte van een datatype teruggeeft. Deze operator wordt bij dynamische geheugenallocatie gebruikt, zie paragraaf 15.5. De aanroep van `printb` luidt dan:

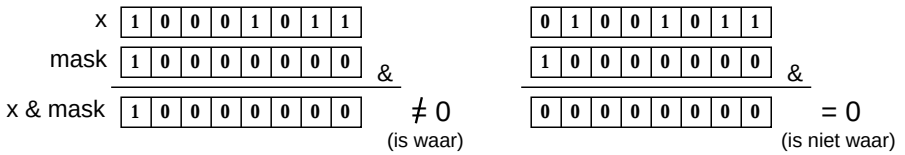
```
int i = 12345;
unsigned char u = 150;

printb(i, sizeof(i), 1); // print: 0000_0000_0000_0000_0011_0000_0011_1001
printb(u, sizeof(u), 0); // print: 10010110
```

**Regel 5**
`1ULL << (nbits-1) ULL`

Om de bits te selecteren wordt er een masker `mask` gebruikt met de lengte van het af te drukken getal (`nbits`) waarvan het meest significante bit 1 is. Voor een **`char`** is dit bijvoorbeeld `10000000`. De suffix `ULL` zorgt ervoor dat de constante `1` een **`unsigned long long`** is. Kleinere typen gebruiken niet alle bits. Bij een **`char`** is het masker feitelijk `00 ... 0010000000`.

**Regel 8**
`x & mask`

Het masker `mask` met de bitbewerking (`&`) maakt de uitdrukking waar is als het meest significante bit hoog is. De bitbewerkingen worden in paragraaf 9.11 besproken. De bitbewerking `&` voert een bitsgewijze EN uit. Dat betekent dat een bit alleen hoog is als de betreffende bits van de operanden allebei hoog zijn.




**Figuur 9.8 : Bitmaskering voor test of meest significante bit hoog is.**

Figuur 9.8 laat zien dat met dit masker de uitdrukking `x & mask` *waar* is als het meest significante bit van `x` de waarde 1 heeft.

**Regel 16**
`x <<= 1`

De uitdrukking `x <<= 1` is een verkorte schrijfwijze voor `x = x << 1`. Meer informatie over de verkorte schrijfwijze staat in paragraaf 9.12 De operator `<<` is een schuifoperator en schuift de bits in dit geval een positie naar links, zie ook paragraaf 9.11. Samen met de gekozen bitmaskering zorgt dit er voor dat het meest significante bit eerst wordt afgedrukt en het minst significante bit het laatst.

## 9.7 Rekenkundige operatoren

C kent vijf rekenkundige bewerkingen: optellen, aftrekken, vermenigvuldigen, delen en de modulus. Tabel 9.4 geeft een overzicht. Bij het delen is de deling geheeltallig als alle operanden gehele getallen zijn. Is een van de operanden een gebroken getal, dan is het resultaat ook een gebroken getal.
Op pagina 58 is de modulusoperator al besproken en is al verteld dat de modulus in C geen wiskundige modulo is en dat het gewoon de rest van een geheeltallige deling geeft.
Net als Java kent C geen symbool voor machtverheffen. De bibliotheek *math*, zie tabel 9.5, bevat een functie `pow()`, die de macht van twee getallen uitrekent. Voor kwadrateren is deze functie niet nodig. Het kwadraat van `a` is ook gelijk aan `a*a`. De parameters en de retourwaarde van de rekenkundige bewerkingen uit tabel 9.5 zijn allemaal van het type **`double`**. Dus ook een afronding `floor(5.3)` levert het gebroken getal 5,0 op.

**Tabel 9.4 : De rekenkundige bewerkingen.**

| bewerking | symbool | voorbeeld | resultaat |
|---|---|---|---|
| optellen | `+` | `x = 32 + 5` | x wordt 37 |
| aftrekken | `-` | `x = 32 - 5` | x wordt 27 |
| vermenigvuldigen | `*` | `x = 32 * 5` | x wordt 160 |
| delen (geheeltallig) | `/` | `x = 32 / 5` | x wordt 6 |
| delen (gebroken getallen) | `/` | `x = 32 / 5.0` | x wordt 6.4 |
| modulus | `%` | `x = 32 % 5` | x wordt 2 |

Naast de bewerkingen uit *math* staan er in de standaard bibliotheek *stdlib* ook een aantal geheeltallige rekenkundige bewerkingen, zie tabel 9.6. Om de bewerkingen uit *math* en *stdlib* te gebruiken zijn deze include-regels nodig:

```
#include <math.h>
#include <stdlib.h>
```

Bij sommige oudere compilers en bij crosscompilers voor kleinere microcontrollers moet de math-bibliotheek expliciet worden meegelinkt met de optie `-lm`.

**Tabel 9.5 : De belangrijkste rekenkundige bewerkingen uit de math-bibliotheek.**

| C-functie | functienaam | omschrijving |
|---|---|---|
| `sin(x)` | sinus | $\sin(x)$ met $x$ in radialen |
| `cos(x)` | cosinus | $\cos(x)$ met $x$ in radialen |
| `tan(x)` | tangens | $\tan(x)$ met $x$ in radialen |
| `asin(x)` | boogsinus | $\arcsin(x)$ met $x \in [-1,1]$ uitkomst in radialen $[-\pi/2, \pi/2]$ |
| `acos(x)` | boogcosinus | $\arccos(x)$ met $x \in [-1,1]$ uitkomst in radialen $[0, \pi]$ |
| `atan(x)` | boogtangens | $\arctan(x)$ uitkomst in radialen $[-\pi/2, \pi/2]$ |
| `atan2(y,x)` | | $\arctan(y/x)$ uitkomst in radialen $[-\pi/2, \pi/2]$ |
| `sinh(x)` | sinus hyperbolicus | $\frac{e^x - e^{-x}}{2}$ |
| `cosh(x)` | cosinus hyperbolicus | $\frac{e^x + e^{-x}}{2}$ |
| `tanh(x)` | tangens hyperbolicus | $\frac{\sinh^2 x}{\cosh x}$ |
| `exp(x)` | exponentiële functie | $e^x$ |
| `log(x)` | natuurlijke logaritme | $\ln(x)$ |
| `log10(x)` | logaritme met grondtal 10 | $^{10}\log(x)$ |
| `sqrt(x)` | wortel | $\sqrt{(x)}$ |
| `ceil(x)` | afronden naar boven | |
| `floor(x)` | afronden naar beneden | |
| `round(x)` | afronden | |
| `fabs(x)` | absolute waarde | $\lvert x \rvert$ |
| `pow(x,y)` | machtverheffen | $x^y$ |

**Tabel 9.6 : De rekenkundige bewerkingen uit de stdlib-bibliotheek.**

| C-functie | functienaam | omschrijving |
|---|---|---|
| `abs(n)` | absolute waarde | $\lvert n \rvert$ |
| `rand()` | random | geeft een willekeurige waarde tussen `0` en `RAND_MAX` terug |
| `srand(n)` | seed random | introduceer een nieuwe startwaarde voor rand() |

## 9.8 Boolean

C kent in tegenstelling tot veel andere talen geen aparte type voor waar en niet waar. Bij relationele operatoren als <, == of >= worden de begrippen waar en niet waar wel gebruikt. Alleen worden deze gerepresenteerd met een `integer`. Een `0` is niet waar (*false*) en alle andere getallen betekenen waar (*true*). Soms worden met **`#define`** twee constanten `TRUE` en `FALSE` gedefinieerd:

Omdat het gebruik van een eigen `TRUE` en `FALSE` of een eigen boolean type niet standaard is, is het beter om dit niet te gebruiken. Het maakt de code niet leesbaarder en is eerder verwarrend. Elke C-programmeur weet dat `0` *false* betekend en dat alle integers *true* zijn.

```
#define TRUE   1
#define FALSE  0
```

Een iets fraaiere methode is om een boolean type te definiëren:

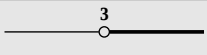
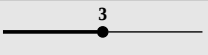
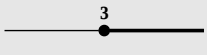
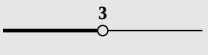
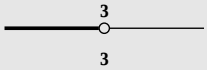
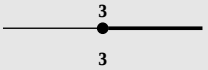
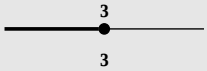
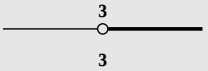
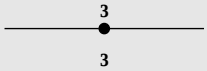
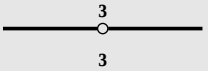
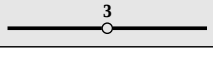
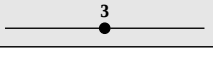
```
typedef enum _boolean {FALSE,TRUE} boolean;
boolean b;
```

Andere C-compilers kennen soms een headerbestand `stdbool.h` met een eigen boolean type.

## 9.9 De relationele bewerkingen

In tabel 9.7 staan de relationele operatoren. De uitkomst van de bewerkingen is waar (1) of niet waar (`0`). In de voorlaatste kolom van de tabel is het domein voor de waarden `x` gegeven waarvoor het gegeven voorbeeld waar (1) is. In de laatste kolom van de tabel is het domein voor de waarden `x` gegeven waarvoor het gegeven voorbeeld niet waar (`0`) is.

**Tabel 9.7 : De zes relationele operatoren voor logische bewerkingen.**

| bewerking | symbool | voorbeeld | domein *waar* | domein *niet waar* |
|---|---|---|---|---|
| groter dan | `>` | `x > 3` |  |  |
| groter dan of gelijk aan | `>=` | `x >= 3` |  |  |
| kleiner dan | `<` | `x < 3` |  |  |
| kleiner dan of gelijk aan | `<=` | `x <= 3` |  |  |
| gelijk aan | `==` | `x == 3` |  |  |
| ongelijk aan | `!=` | `x != 3` |  |  |

## 9.10 Logische operatoren

C kent drie logische bewerkingen: de EN, de OF en de NIET. In C worden deze respectievelijk geschreven als `&&`, `||` en `!`. Figuur 9.9 geeft de waarheidstabellen voor deze drie bewerkingen. C kent standaard geen boolean type. In C betekent `0` niet waar en elk ander geheel getal is waar. Bij ingewikkelde logische uitdrukkingen is het verstandig om ronde haken om elke deelbewerking te zetten, zoals hier is gedaan:

```
if ( (((!a) && b) || (c && b)) || (!(a && d)) ) {
  printf("true\n");
}
```

Er is dan geen misverstand mogelijk over welke bewerking voorrang heeft.

| x | y | x && y |
|---|---|---|
| niet waar | niet waar | niet waar |
| niet waar | waar | niet waar |
| waar | niet waar | niet waar |
| waar | waar | waar |

| x | y | x \|\| y |
|---|---|---|
| niet waar | niet waar | niet waar |
| niet waar | waar | waar |
| waar | niet waar | waar |
| waar | waar | waar |

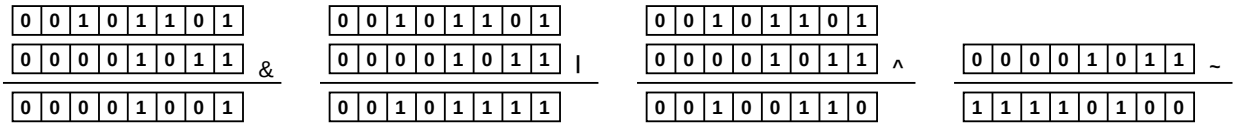
| x | !x |
|---|---|
| niet waar | waar |
| waar | niet waar |

**Figuur 9.9 :** **De waarheidstabellen voor de logische bewerkingen: &&, ||, !.** De waarden van de operanden mogen van alles zijn: nul betekent niet waar en alles ongelijk aan nul is waar. De uitkomst is echter altijd 0 als de bewering niet waar is en is altijd 1 als de bewering waar is.

## 9.11 Bitbewerkingen

Verwar & niet met && en verwar | niet met ||. De symbolen & en && zijn beide EN-functies. Alleen is && de logische EN en & de bitsgewijze EN.
Op dezelfde manier is || de logische OF en | de bitsgewijze OF.

C is een taal, die dicht bij de hardware staat, daarom is het niet vreemd dat deze taal ook een aantal bitbewerkingen kent. Bij het programmeren van microcontrollers moeten bits uit allerlei registers gemanipuleerd kunnen worden. Bitbewerkingen zijn daarbij essentieel. C kent vier logische bitbewerkingen en twee schuifoperatoren. Al deze bewerkingen worden uitgevoerd op gehele getallen. Figuur 9.10 laat de vier logische bitbewerkingen zien met een achtbitsgetal (**char**). De bitbewerkingen manipuleren de bits afzonderlijk. Het symbool & is de bitsgewijze EN (*bitwise AND*). De bitwaarde van de bewerking is 1 als de overeenkomstige bits van de operanden ook allebei 1 zijn, anders is de bitwaarde een 0.
Het symbool | is de bitsgewijze OF (*bitwise OR*). De bitwaarde van de bewerking is 1 als een van de overeenkomstige bits van de operanden 1 is, anders is de bitwaarde 0. Het symbool ^ is de bitsgewijze XOF (*bitwise XOR*). De bitwaarde van de bewerking is 1 als precies een van de overeenkomstige bits van de operanden 1 is, anders is de bitwaarde 0. Het symbool ~ is het *one's complement* van de operand. Alle bits van de operand worden geïnverteerd.




**Figuur 9.10 :** **De logische bitbewerkingen.** Bij alle logische bitbewerkingen worden de bits afzonderlijk gemanipuleerd. Bij & (*bitwise AND*) is de bitwaarde 1 als beide bits 1 zijn. Bij | (*bitwise OR*) is de bitwaarde 1 als een van beide bits 1 is. Bij ^ (*bitwise XOR*) is de bitwaarde 1 als slechts een van de bits 1 is. Bij ~ (*one's complement*) worden alle bits geïnverteerd.

De schuifoperatoren << en >> schuiven de bits respectievelijk een aantal posities naar links of naar rechts. De rechter operand is het aantal bits dat er geschoven moet worden. Bij het naar links schuiven worden er aan de rechterkant nullen toegevoegd. en bij het naar rechts schuiven worden er aan de linkerkant nullen toegevoegd. Bij het naar links schuiven verdwijnen er enen en nullen aan de linkerkant en bij het naar rechts schuiven aan de rechterkant.


**Figuur 9.11 :** **De schuifoperatoren.** Bij << worden de bits naar links geschoven en worden er rechts nullen toegevoegd. Bij >> worden de bits naar rechts geschoven en worden er links nullen toegevoegd.

Figuur 9.11 toont het schuiven voor deze bewerkingen:

```
unsigned char x = 45;   //  x is 0x00101101
unsigned char a = 45;   //  a is 0x00101101
unsigned char y;
unsigned char b;

y = x << 3;             //  y is 0x01101000 (104)
b = a >> 2;             //  a is 0x00001101 (11)
```

Met $n$ bits naar links schuiven is hetzelfde als met $2^n$ vermenigvuldigen en met $n$ bits naar rechts schuiven is hetzelfde als door $2^n$ delen.

Twee bits naar rechts schuiven is hetzelfde als delen door vier ($\frac{45}{4} = 11$). Drie bits naar links schuiven is hetzelfde als vermenigvuldigen met acht ($45 \times 8 = 360$). Omdat `y` een **`unsigned char`** is, valt het meest significante bit weg en krijgt `y` de waarde `360-256=104`.

Bitbewerkingen zijn enorm handig bij het manipuleren van de bits in de registers van microcontrollers. Deze bewerkingen kunnen gebruikt worden om bits te setten, te clearen, te toggelen en te testen. In paragraaf 12.5 wordt dit verder toegelicht.

## 9.12 Verkorte schrijfwijze bij toekenningen

Op pagina 51 zijn de incrementoperator (`++`) en de decrementoperator (`--`) besproken. Hoewel incrementeren en decrementeren andere bewerkingen zijn dan optellen en aftrekken, kunnen de bewerkingen `i++` en `i--` ook geschreven worden als `i=i+1` en als `i=i-1`. De bewerkingen `i++` en `i--` zijn verkorte notaties van deze toekenningen.

In programma's worden regelmatig toekenningen gedaan aan een variabele, terwijl die variabele ook in het rechter deel van de toekenning voorkomt, zoals:

```
x = x * 4;
y = y - 5;
```

Met de verkorte notatie wordt dit:

```
x *= 4;
y -= 5;
```

De `*=` en `-=` zijn net als `=` toekenningsoperatoren. Tabel 9.8 geeft alle toekenningsoperatoren met een voorbeeld en de betekenis van het voorbeeld.

**Tabel 9.8 : Alle toekenningsoperatoren met een voorbeeld en de bijbehorende betekenis**

| symbool | voorbeeld | betekenis voorbeeld |
|---|---|---|
| `=` | `x = 2` | `x = 2` |
| `+=` | `x += 2` | `x = x + 2` |
| `-=` | `x -= 2` | `x = x - 2` |
| `*=` | `x *= 2` | `x = x * 2` |
| `/=` | `x /= 2` | `x = x / 2` |
| `%=` | `x %= 2` | `x = x % 2` |
| `&=` | `x &= 2` | `x = x & 2` |
| `\|=` | `x \|= 2` | `x = x \| 2` |
| `^=` | `x ^= 2` | `x = x ^ 2` |
| `<<=` | `x <<= 2` | `x = x << 2` |
| `>>=` | `x >>= 2` | `x = x >> 2` |

## 9.13 Bewerkingsvolgorde operatoren

C voert — net als bij gewoon rekenen — de bewerkingen in een voorgeschreven volgorde uit. De bewerking:

$$20 - 2 * 8$$

kan in principe uitgerekend worden door eerst 2 van 20 af te trekken en het resultaat daarna met 8 te vermenigvuldigen of door eerst 2 met 8 te vermenigvuldigen en dit resultaat van 20 af te trekken:

$$(20 - 2) * 8 = 144$$
$$20 - (2 * 8) = 4$$

Vroeger werden er in Nederland andere rekenregels gebruikt. Vermenigvuldigen ging voor delen en optellen voor aftrekken. Bij de huidige rekenregels is de prioriteit van vermenigvuldigen en delen hetzelfde. Ook die van optellen en aftrekken zijn gelijk. Deze nieuwe regels sluiten beter aan bij de rekenregels van programmeertalen.
De verouderde rekenregels staan bekend als: *Mijnheer Van Dale Wacht op Antwoord*.

In het eerste geval is het eindresultaat 144 en in het tweede geval 4. De prioriteit of voorrang (*precedence*) is bij deze bewerkingen voor C hetzelfde als voor gewoon rekenen. Vermenigvuldigen gaat voor aftrekken, zodat de uitkomst 4 is.
Bij de gewone regels en bij C hebben vermenigvuldigen en delen dezelfde prioriteit. Dan zijn er nog steeds twee mogelijkheden om een bewerking te evalueren, namelijk van links naar rechts of van rechts naar links:

$20/2 * 5 = 20/(2 * 5) = 2$ associatie van rechts naar links
$20/2 * 5 = (20/2) * 5 = 50$ associatie van links naar rechts

Bij C en bij de gewone rekenregels is de zogenoemde associativiteit (*associativity*) voor vermenigvuldigen en delen van links naar rechts. De uitkomst van de berekening zonder de haakjes is dus 50.

**Tabel 9.9 : Prioriteit en associativiteit van operatoren**

| prioriteit | operatoren | associativiteit |
|---|---|---|
| 1 | `()` `[]` `->` `.` (*expr*)`++` (*expr*)`--` | ⟹ |
| 2 | `!` `~` `+`[1] `-`[2] `*`[3] `&`[4] `++`(*expr*) `--`(*expr*) (*type*) **`sizeof`** | ⟸ |
| 3 | `*`[7] `/` `%` | ⟹ |
| 4 | `+`[5] `-`[6] | ⟹ |
| 5 | `<<` `>>` | ⟹ |
| 6 | `<` `<=` `=>` `>` | ⟹ |
| 7 | `==` `!=` | ⟹ |
| 8 | `&`[8] | ⟹ |
| 9 | `^` | ⟹ |
| 10 | `\|` | ⟹ |
| 11 | `&&` | ⟹ |
| 12 | `\|\|` | ⟹ |
| 13 | `?:` | ⟸ |
| 14 | `=` `*=` `/=` `+=` `-=` `%=` `&=` `^=` `\|=` `<<=` `>>=` | ⟸ |
| 15 | `,` | ⟹ |

opmerkingen:
1. unair plusteken
2. unair minteken
3. referentie-operator bij pointers
4. adresoperator bij pointers
5. binair plusteken
6. binair minteken
7. vermenigvuldiging
8. bitsgewijze EN

De regels voor de prioriteit en de associativiteit staan in tabel 9.9. Hieronder staan links een aantal compact opgeschreven uitdrukkingen.

```
s *= s - 5;
f = g = h = 2;
f = !a && b || c && b || !(a && d);
z = x * ++y;

z = x * y++;
```

```
s = s * (t - 5);
f = (g = (h = 2));
f = (((!a) && b) || (c && b)) || (!(a && d));
++y;
z = x * y;
z = x * y;
y++;
```

Voor de juiste interpretatie zijn de prioriteit- en associativiteitsregels uit de tabel nodig. Soms is dat heel lastig te zien. Rechts staan dezelfde uitdrukkingen in een uitgebreide vorm met haakjes of gesplitst in aparte toewijzingen. Zonder de prioriteit- en associativiteitsregels zijn deze notaties ook te begrijpen.

Er is een groot verschil tussen bijvoorbeeld de + en de ++-operator. Bij de + veranderen de operanden niet en bij de ++ verandert de operand wel:

```
s = a + b;  // s verandert maar a en b veranderen niet
z = i++;    // z verandert en i verandert
```

Deze bijwerking noemt men een neveneffect (*side-effect*). Bij ingewikkelde constructies met neveneffecten ligt niet altijd vast wat de volgorde van de bewerkingen is. Bij de onderstaande toekenning is het niet duidelijk of de arrayindex `j` eerst wordt opgehoogd.

```
w[j] = j++;
```

Er zijn twee interpretaties mogelijk:

```
w[j] = j;
j++;
```

```
j++;
w[j] = j;
```

De GNU-compiler gebruikt — net als de meeste compilers — de linker interpretatie. De oude waarde van `j` wordt eerst als arrayindex gebruikt en daarna wordt `j` opgehoogd.

In paragraaf 8.2 staan ook een paar voorbeelden met neveneffecten. Zelfs als het voor de compilers wel duidelijk is, is het voor de programmeur vaak onduidelijk. Ingewikkelde constructies met neveneffecten maken een programma niet kleiner of sneller, maar verminderen de leesbaarheid en worden daardoor rijker aan fouten. Het is verstandig om haakjes toe te passen of om lastige constructies te splitsen in meerdere kleinere toewijzingen.

# 10

# De ATmega32

**Doelstelling**

Dit hoofdstuk is een inleiding op de ATmega32. Je leert hoe een ATmega32 is opgebouwd, wat de belangrijkste features zijn en hoe je de ATmega32 gebruikt.

**Onderwerpen**

De behandelde onderwerpen zijn:

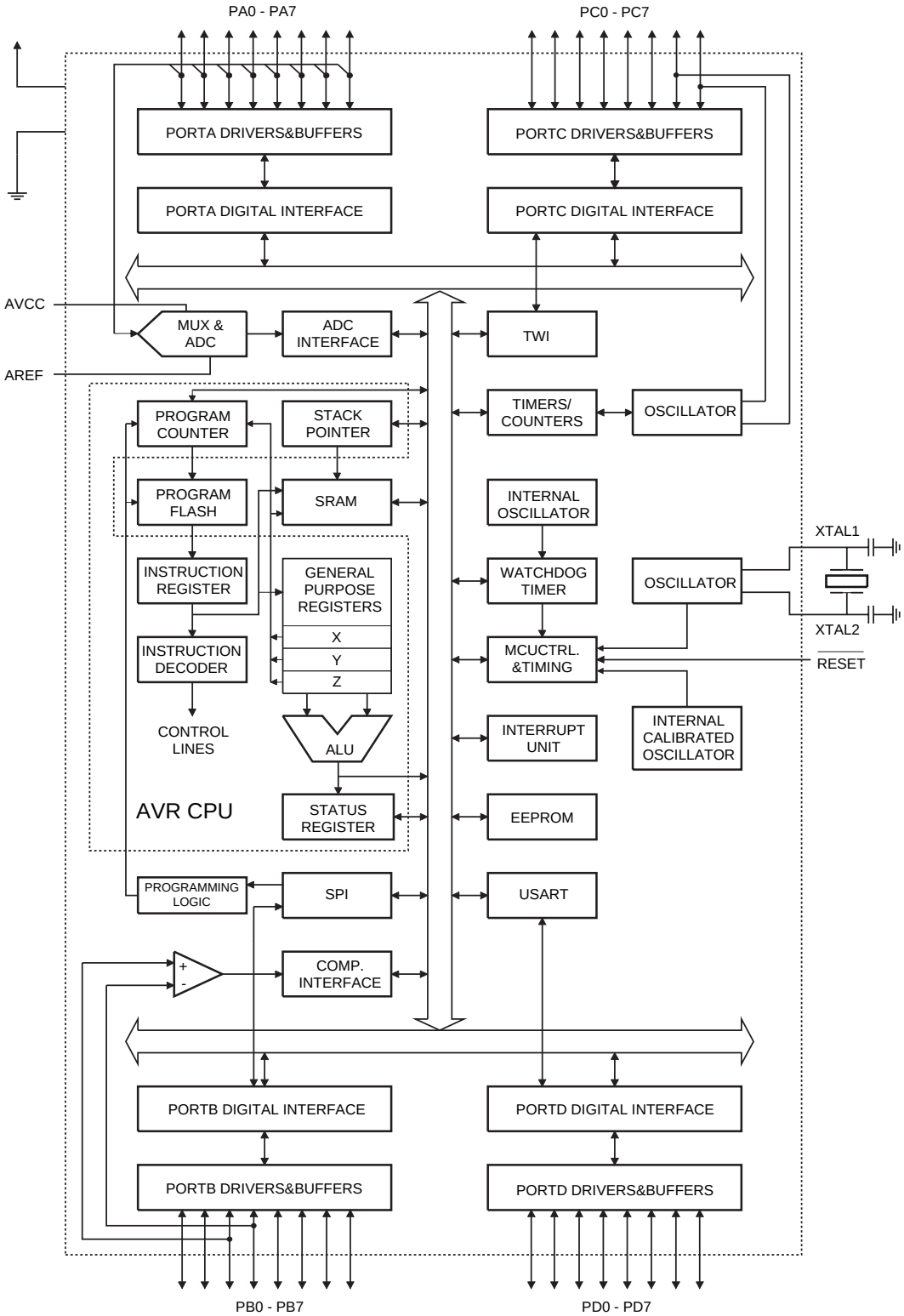
- De geheugenorganisatie van de ATmega32. Besproken worden het RAM-, het EEPROM- en het flashgeheugen.
- De plaats van de resetvector en de interruptvectoren in het geheugen.
- De lock- en fusebits.
- De mogelijke klokconfiguraties.
- De methoden waarmee de ATmega32 kan worden geprogrammeerd, namelijk: parallel en serieel via SPI of via JTAG
- De ontwikkelomgeving voor de ATmega32.

De 32 in de naam ATmega32 staat voor de 32 kB flash die deze component heeft.

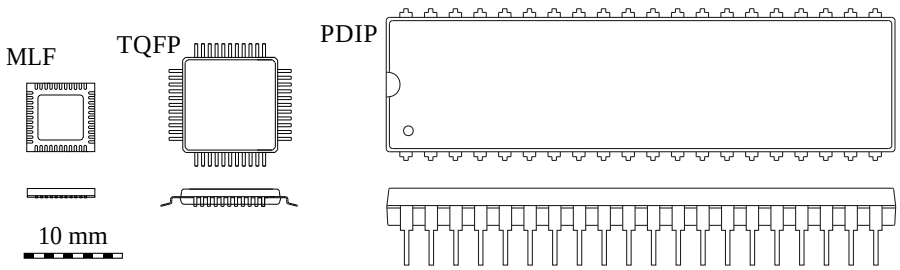
De voorbeelden in dit boek gebruiken de ATmega32 van Atmel. Dat is een microcontroller uit de ATmega-serie. Sommigen zeggen dat AVR voor Advanced Virtual RISC staat. Anderen beweren dat AVR staat voor Alf Vegard RISC en dat de processor is genoemd naar de twee Noren Alf-Egil Bogen en Vegard Wollan, die in hun studententijd de architectuur van de AVR-processor hebben bedacht. De AVR is een RISC-processor met een Harvard-architectuur. De databus is gescheiden van de programmabus. Het blokdiagram staat in figuur 10.2. De instructieset is klein en door het gebruik van *pipelining* duren bijna alle instructies maar één klokslag.

De datasheet van de ATmega32 is — net als de datasheets van andere microcontrollers — zeer uitgebreid. Dit hoofdstuk vat de belangrijkste aspecten uit de datasheet samen. In de hoofdstukken 11 tot en met 13 en de hoofdstukken 18 tot en met 20 wordt aan de hand van praktische voorbeelden meer gedetailleerde informatie over bepaalde eigenschappen gegeven. Zo wordt het interruptmechanisme bij het interruptvoorbeeld besproken.

Deze aanpak past bij de werkwijze als je met een nieuwe microcontroller aan de slag gaat. Het is onmogelijk om eerst alle beschikbare informatie te bestuderen en te begrijpen en daarna pas te gaan beginnen met ontwerpen. Het is beter om eerst een globaal beeld van de microcontroller te krijgen en daarna — *just in time* — de details te bestuderen.




**Figuur 10.2 :** **Het blokdiagram van de ATmega32.** Het hart van de microcontroller is bedacht door Alf-Egil Bogen en Vegard Wollan en is aangegeven met AVR CPU.

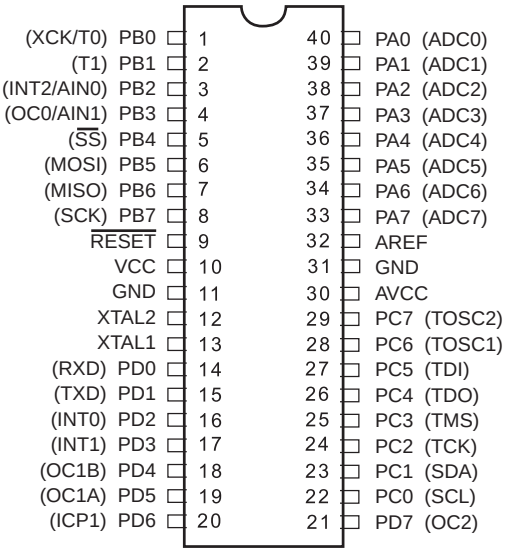



**Figuur 10.3 : De drie behuizingen van de ATmega32.**
Links staat MLF (Micro Lead Frame), in het midden de TQFP (Thin Quad Flat Pack) en rechts de PDIP (Plastic Dual-In-line Package).

## 10.1 De opbouw van de ATmega32

In het blokdiagram worden de individuele aansluitingen van de poorten aangeduid met `PA0-PA7`, `PB0-PB7`, `PC0-PC7` en `PD0-PD7`.

In het blokdiagram van figuur 10.2 staan alle gebruikelijke componenten voor een microcontroller, zoals: program flash, SRAM, stackpointer, EEPROM, ALU, statusregister, programcounter, instructieregister, instructiedecoder en de *general purpose dataregisters*. De ATmega32 heeft vier 8-bits aansluitingen, die in dit boek aangeduid worden met *poort* A, B, C en D.
De ATmega32 is in drie behuizingen verkrijgbaar: PDIP (*Plastic Dual-In-line Package*), TQFP (*Thin Quad Flat Pack*), MLF (*Micro Lead Frame*). Figuur 10.3 laat deze drie packages zien. De MLF en TQFP zijn SMD-componenten (*Surface Mounted Devices*) en de PDIP is een *through hole* component.




**Figuur 10.4 : Een overdruk van de pinout van een 40-pins PDIP-behuizing uit de datasheet van de ATmega32.**

De voorbeelden in dit boek gebruiken de PDIP-behuizing. In figuur 10.4 staat een overdruk van figuur 1 uit de datasheet met de *pinout* van deze behuizing. Bij de in- en uitgangen staan naast de gewone pinnaam tussen ronde haken een of twee andere namen. Elke IO-poort heeft een of meer speciale functies. Pin 16 — dat is pin 2 van poort D (`PD2`) — wordt bijvoorbeeld ook gebruikt voor de externe interrupt 0. Dit is aangegeven met `INT0`.

Aan het blokdiagram van figuur 10.2 en de extra namen bij in- en uitgangen in figuur 10.4 is direct te zien dat de ATmega32 veel speciale functies kent. De belangrijkste features van de ATmega32 zijn:

- een ADC (Analog-to-Digital Converter) met acht kanalen (`ADC0` tot en met `ADC7`);
- twee analoge comparatoren (`AIN0`, `AIN1`);
- een TWI-interface (Two-Wire serial Interface), dat is een tweedraads aansluiting (`SCL`, `SDA`), die gebruikt wordt als $I^2C$-interface;
- een SPI (Serial Peripheral Interface), dit is een vierdraads (`MOSI`, `MISO`, `SCK`, $\overline{\texttt{SS}}$) seriële communicatie, die ook gebruikt kan worden om de component te programmeren;
- een USART (Universal, Synchronous and Asynchronous Receiver and Transmitter), voor RS232-communicatie (`TXD`, `RXD`, `XCK`);
- drie externe interrupts (`INT0`, `INT1`, `INT2`);
- een JTAG-interface voor het debuggen en het programmeren van de ATmega (`TCK`, `TMS`, `TDI`, `TD0`);
- Twee 8-bits tellers/timers met een comparator (`OC0`, `OC2`) waarmee ook PWM-signalen gemaakt kunnen worden;
- Een 16-bits tellers/timer met een comparator(`OC1`) waarmee ook PWM-signalen gemaakt kunnen worden;
- Een *watchdog timer*;
- Een *brown-out detection*;
- Verschillende *sleep modes*;
- Diverse interne oscillatoren en aansluitingen voor externe klokken en oscillatoren (`XTAL1`, `XTAL2`, `TOSC1`, `TOSC2`, `XCK`).

PWM staat voor *Pulse Width Modulation* en betekent pulsbreedtemodulatie. Deze vorm van modulatie wordt veel gebruikt, bijvoorbeeld bij de aansturing van motoren.

De datasheet van de ATmega32 bespreekt uitgebreid alle mogelijkheden. De hoofdstukken 11 tot en met 13 gaan dieper in op een aantal van deze functionaliteiten: het gebruik van de gewone IO, de externe interrupt en het gebruik van tellers. Hoofdstuk 18 behandelt de ADC, en hoofdstuk 20 de USART.

## 10.2 De geheugenorganisatie bij de ATmega32

De ATmega heeft drie soorten geheugens: flashgeheugen voor het programma, RAM voor de opslag van vluchtige data en EEPROM voor de opslag van niet-vluchtige data.

### Het flashgeheugen

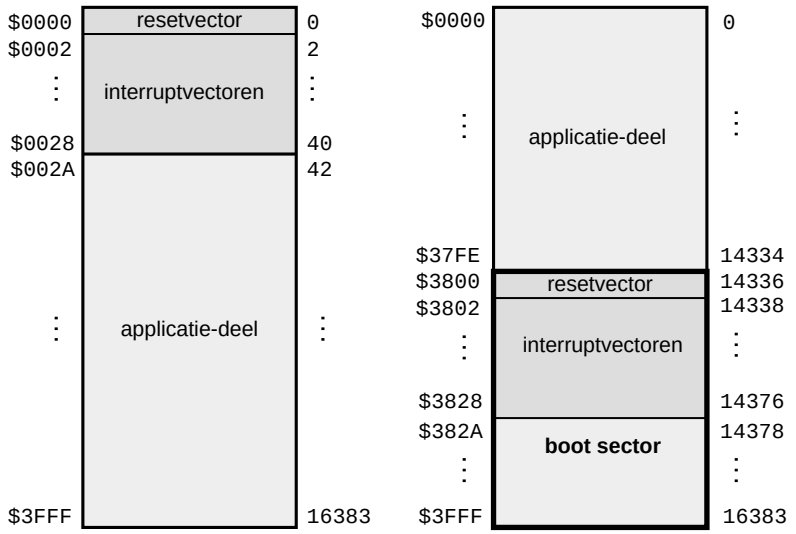
Een *word* is twee bytes oftewel 16-bits breed.

Een *boot loader* is een programma waarmee de microcontroller zichzelf kan (her)programmeren. Dit is handig voor bijvoorbeeld software updates.

Het flashgeheugen bestaat uit ruim 16000 *words*, dat is ongeveer 32 kB. Een deel van dit geheugen kan gereserveerd worden als *boot sector*. Daar is dan ruimte voor een *boot loader*. Om interrupts te kunnen gebruiken, moeten de adressen van de interruptfuncties bekend zijn. Deze adressen staan op een vaste plek in het geheugen direct na de resetvector. Deze verwijzingen worden de interruptvectoren genoemd. De resetvector bevat het adres van het hoofdprogramma. De resetvector is het eerste geheugenadres van het flashgeheugen of het eerste adres van de bootsector. Het gebruik en de mogelijkheden van de bootsector wordt uitgebreid in de datasheets behandeld. De resetvector en interruptvectoren worden in hoofdstuk 12 behandeld.

In figuur 10.5 zijn de hexadecimale en de decimale waarden van de adressen gegeven. Net als in de datasheet zijn de hexadecimale waarden aangegeven met een $-teken voor het getal. In C worden hexadecimale waarden aangegeven met het prefix `0x`. Adres 42 decimaal is dus in de figuur $002A en in C `0x002A`.

De resetvector en de interruptvectoren bevatten sprongopdrachten naar respectievelijk het begin van het hoofdprogramma en de betreffende interruptfuncties (*interrupt service routines*).




**Figuur 10.5 : De indeling van het flashgeheugen.** Links staat de standaard indeling zonder bootsector. Rechts staat een voorbeeld met een bootsector. De geheugenadressen zijn hexadecimaal en decimaal gegeven.

### RAM

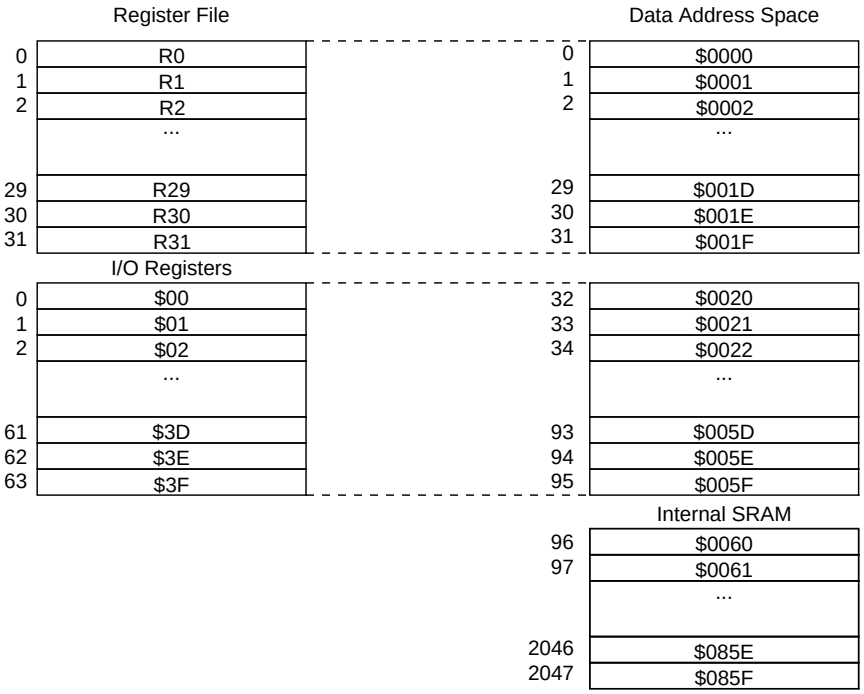
Het RAM-geheugen, de zogenoemde *special purpose registers* en alle zogenoemde IO-registers worden op dezelfde manier geadresseerd. Figuur 10.6 geeft deze adressering en is een overdruk van figuur 9 uit de datasheet. Er zijn 32 speciale purpose registers. Een assemblerprogrammeur moet goed op de hoogte zijn van de functionaliteit van deze registers. De zes registers R26 tot en met R31 worden bij 16-bits berekeningen gebruikt als drie 16-bits registers X, Y en Z. Er zijn 64 IO-register. Hiervan worden twaalf registers gebruikt voor de normale IO, de andere 52 registers worden gebruikt voor de extra mogelijkheden van de microcontroller, zoals instellingen voor interrupts, tellers en de ADC.

De IO-registers worden decimaal genummerd met `0` tot en met 63 en hexadecimaal met $00 tot en met $3F. Figuur 10.6 laat zien dat de IO-registers wat betreft de adressering direct achter de 32 general purpose registers staan. Het adres van een IO-register is zodoende 32 — oftewel $20 hexadecimaal — hoger dan het nummer. Een C-programmeur hoeft deze adressen niet te kennen. Een assemblerprogrammeur heeft hier wel mee te maken en dit kan heel verwarrend zijn.

Het feitelijke RAM-geheugen komt in de adressering na de IO-registers en is 1952 bytes groot.

### EEPROM

Het EEPROM is 1024 bytes groot en Atmel garandeert dat het minimaal 100.000 keer geschreven en gewist kan worden. Het EEPROM is niet direct te adresseren. Het EEPROM-geheugen heeft een eigen adressering, die toegankelijk is via een aantal registers in het IO-registerdeel van het RAM-geheugen. In de datasheet wordt uitgelegd hoe deze registers gebruikt moeten worden om data in het EEPROM te schrijven en uit te lezen.




**Figuur 10.6** : **De indeling van het RAM-geheugen.** Dit is een kopie van figuur 9 uit de datasheet, alleen zijn naast de hexadecimale waarden ook de decimale waarden gegeven.

Wees erg voorzichtig met het aanpassen van de fuse- en lockbits. Het kan zijn dat na de aanpassing de microcontroller niet meer te programmeren is. Bestudeer altijd eerst de documentatie grondig.

### De lockbits en fusebits

De ATmega32 heeft zes lockbits waarmee onder andere ingesteld kan worden dat het flash of de bootsector niet meer geherprogrammeerd kan worden.
Daarnaast zijn er twee fusebytes met zestien fusebits. Met zeven van deze bits wordt de klok ingesteld. Verder is er een `SPIEN`-bit en een `JTAGEN`-bit om het serieel programmeren via de SPI of de JTAG-interface mogelijk te maken. De fuse- en lockbits worden apart geprogrammeerd. De meeste programmeerprogramma's hebben hier een apart menu voor.

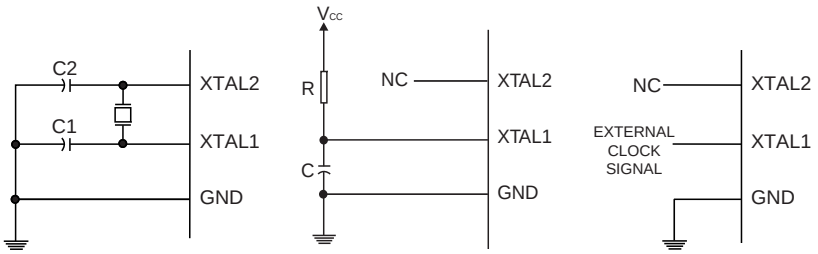
Lees bij het programmeren altijd eerst de huidige waarden van de bits uit de microcontroller uit, wijzig deze instelling en programmeer dan de nieuwe instelling.

## 10.3 De systeemklok en klokopties

De ATmega32 kent verschillende manieren om de systeemklok te configureren. Daar zijn zeven fusebits voor nodig: vier `CKSEL`-bits waarmee het type oscillator worden geselecteerd; twee `SUT`-bits waarmee de *startup time* wordt ingesteld en een `CKOPT`-bit waarmee de oscillatormethode wordt geselecteerd.
Tabel 10.1 toont de verschillende klokconfiguraties. De externe oscillatoren worden aangesloten aan de pinnen `XTAL1` en `XTAL2`. Figuur 10.7 geeft de schema's voor de verschillende configuraties.
Naast `XTAL1` en `XTAL2` heeft de ATmega32 nog vier aansluitingen voor een klok: op `TOSC1` en `TOSC2` kan een resonator worden aangesloten voor timer 2 en op de pinnen `T0` en `T1` kan een externe klok worden aangesloten voor respectievelijk

**Tabel 10.1 : De configuratiebits voor de klok.**

| CKSEL-bits | klokconfiguratie | frequentiegebied |
|---|---|---|
| 1111 | externe keramische of kristaloscillator | 3,0···8,0 MHz |
| 1110 | externe keramische of kristaloscillator | 0,9···3,0 MHz |
| 1101 | externe keramische oscillator | 0,4···0,9 MHz |
| 1100 | externe keramische of kristaloscillator | > 1,0 MHz |
| 1011 | externe keramische of kristaloscillator | > 1,0 MHz |
| 1010 | externe keramische of kristaloscillator | > 1,0 MHz |
| 1001 | externe laagfrequent kristal | geoptimaliseerd voor 32768 Hz |
| 1000 | externe RC-oscillator | 8,0···12,0 MHz |
| 0111 | externe RC-oscillator | 3,0···8,0 MHz |
| 0110 | externe RC-oscillator | 0,9···3,0 MHz |
| 0101 | externe RC-oscillator | 0,4···0,9 MHz |
| 0100 | interne RC-oscillator | 8 MHz |
| 0011 | interne RC-oscillator | 4 MHz |
| 0010 | interne RC-oscillator | 2 MHz |
| 0001 | interne RC-oscillator | 1 MHz (standaardconfiguratie) |
| 0000 | externe klok | |




**Figuur 10.7 : De schema's voor de oscillatoren.** Links staat de standaardaansluiting voor de keramische en de kristaloscillator. De resonator wordt aangesloten op pin XTAL1 en XTAL2. De twee condensatoren moeten gelijk zijn en een waarde van 22 pF hebben. Bij een laagfrequent kristal mogen deze worden weggelaten mits het fusebit CKOPT ingeschakeld is. In het midden staat de aansluiting voor een externe RC-oscillator. De condensator moet minimaal 22 pF zijn en mag worden weggelaten als CKOPT ingeschakeld is. Er wordt dan een interne C van 36 pF gebruikt. Rechts staat de aansluiting voor een externe klok. NC betekent *Not Connected*

timer 0 en timer 1.

## 10.4 Het programmeren van de ATmega32

Er zijn drie methoden waarmee de ATmega32 geprogrammeerd kan worden:

- parallel,
- serieel via de Serial Peripheral Interface,
- serieel via de JTAG-interface.

In de datasheet staat de nodige informatie en er zijn speciale application notes.

### Parallel programmeren

Parallel programmeren wordt niet gebruikt voor ISP (*In System Programming*), omdat er erg veel pinnen gebruikt worden en omdat er een speciale programmeerspanning van 12 V nodig is. Parallel programmeren wordt vooral gebruikt

in speciale programmers om een of meerdere microcontrollers snel te programmeren.

**Serieel programmeren via de SPI**

De drie aansluitingen (MOSI, MISO, SCK) van de SPI en de reset ($\overline{\text{RESET}}$) worden gebruikt om de ATmega32 serieel te programmeren. Dat kan vanuit een andere microcontroller of vanuit een pc via de seriële of parallelle poort. Op het internet staan tientallen schakelingen voor deze seriële manier van programmeren. Ook zijn er diverse programmeerprogramma's waarmee de microcontroller geprogrammeerd kan worden: AVRdude, Ponyprog, AVRisp en natuurlijk ook de ontwikkelomgeving voor AVRstudio zelf. Niet alle schakelingen werken met alle programmeerprogramma's.

Er zijn vele commerciële programmers te koop in allerlei prijsklasses. Vooral voor de seriële en parallelle poort zijn er goedkope oplossingen. Moderne pc's en laptops hebben deze poorten niet meer. Een alternatief is om een USB-RS232 convertor te gebruiken. Programmers, die via de USB-poort werken, gebruiken vaak intern ook een USB-RS232 convertor of hebben een microcontroller voor de communicatie.

Het fusebit SPIEN moet ingeschakeld (*enabled*) zijn om serieel programmeren mogelijk te maken. Bij nieuwe devices is dit bit fabriekswege ingeschakeld. Het SPIEN-bit kan alleen met een parallelle *programmer* of — mits het JTAGEN-bit ingeschakeld is — met een JTAG-programmer worden teruggezet.

**Serieel programmeren via JTAG**

JTAG staat voor *Joint Test Action Group*. Eind jaren zeventig voorzagen een aantal bedrijven, zoals Philips en Intel, dat het testen van PCB's een complex probleem zou worden door het gebruik van multilayers en de steeds kleinere pinafstanden bij de componenten. Deze bedrijven hebben de JTAG opgericht. Deze groep heeft een serieel testprotocol ontwikkeld. De meeste digitale IC's hebben speciale logica en extra aansluitingen voor dit testprotocol. De JTAG-interface bestaat uit vier signalen:

- TDI Test Data In
- TDO Test Data Out
- TMS Test Mode Select
- TCK Test Clock

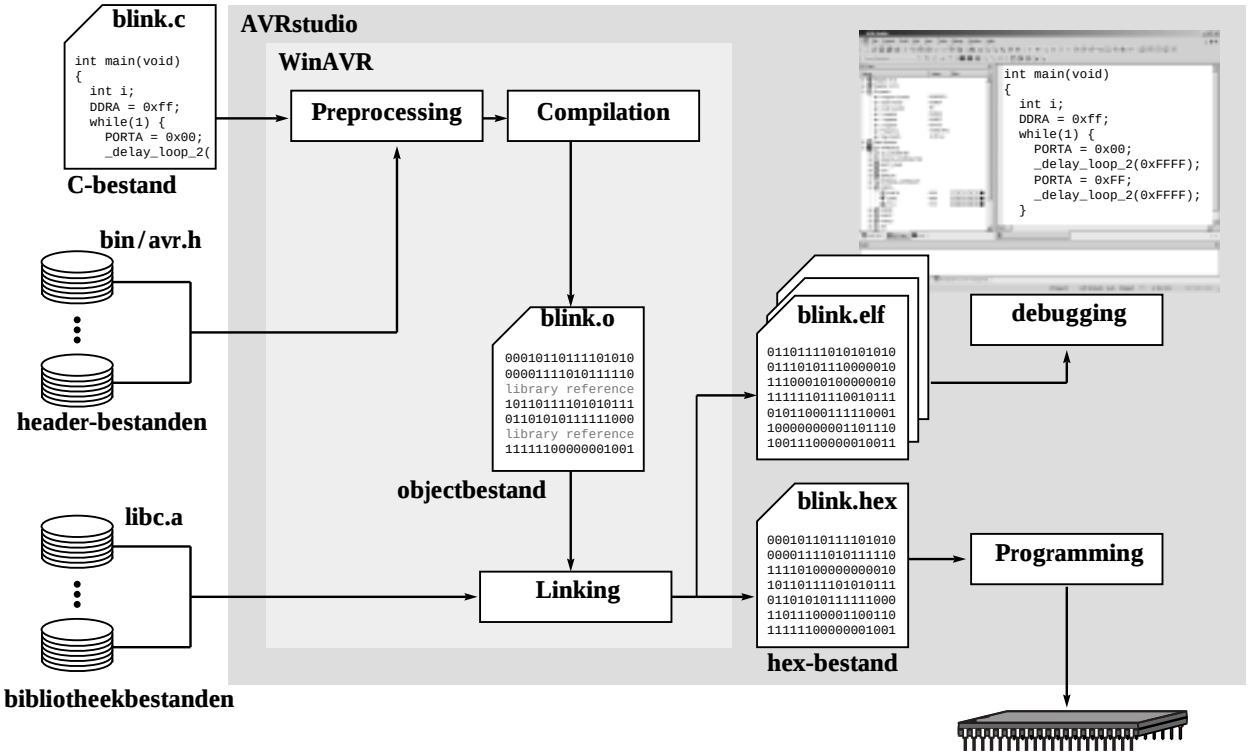
De JTAG-interface is ook geschikt om componenten te programmeren en te debuggen. Bijlage B geeft meer achtergronden van JTAG. JTAG-programmers zijn in allerlei prijsklasses te koop. Sommige kunnen worden gebruikt om te debuggen, andere niet. De JTAG-programmers hebben altijd een microcontroller die communiceert met de te programmeren microcontroller.

Om de JTAG-interface te kunnen gebruiken, moet het fusebit JTAGEN ingeschakeld zijn. Bij nieuwe devices is dit bit fabriekswege ingeschakeld. Dit bit kan alleen met een parallelle *programmer* of — mits het SPIEN-bit ingeschakeld is — met een gewone seriële programmer ingeschakeld worden. Omdat de JTAG-interface gebruikt kan worden voor debuggen, blijven na het programmeren de betreffende IO's in JTAG-mode. Deze functionaliteit kan uitgezet worden met het OCDEN fusebit of met het JTD-bit uit het MCUCSR-register.

De voorbeelden in dit boek gebruiken de JTAG-interface voor het programmeren.

## 10.5 De ontwikkelomgeving voor de ATmega32

Voor het programmeren van de ATmega32 gebruikt dit boek AVRstudio met WinAVR. WinAVR is een gratis Windows ontwikkelomgeving voor de AVR-microcontrollers. Dit pakket bestaat uit avr-gcc — een GNU C-Compiler voor




**Figuur 10.8 : De ontwikkelomgeving met AVRstudio en WinAVR.** Het compilatietraject van WinAVR is vergelijkbaar met dat van de gewone GNU-compiler, zie figuur 2.4. Alleen wordt er nu een hex-bestand en een aantal andere bestanden gegenereerd. Het hex-bestand wordt gebruikt om de microcontroller te programmeren. De andere bestanden worden door AVRstudio gebruikt voor het debuggen of simuleren. Rechtsboven staat de gebruikersinterface van AVRstudio. Het linkerdeel van deze interface toont de registers van de microcontroller, onderaan bevindt zich het commentaarvenster en het rechterdeel is een editor met de code, die gedebugd wordt.

de AVR — en allerlei hulpprogramma's, zoals debugger (`gdb`) en een programmer (`avrdude`). AVRstudio is de ontwikkelomgeving van Atmel en kent standaard geen eigen C-compiler. De GNU C-Compiler van WinAVR wordt in AVRstudio als plug-in gebruikt en is dan volledig geïntegreerd in de AVRstudio ontwikkelomgeving. Figuur 10.8 geeft het compilatietraject met AVRstudio en WinAVR. Voor het debuggen en programmeren wordt AVRstudio gebruikt.

Het compilatietraject van WinAVR komt overeen met dat van de gewone GNU-compiler uit figuur 2.4. Het verschil tussen figuur 2.4 en figuur 10.8 is dat er na het linken nu geen uitvoerbaar programma is. In plaats daarvan is er een zogenoemd `hex`-bestand gemaakt waarmee de microcontroller wordt geprogrammeerd. Naast het `hex`-bestand wordt er nog een aantal bestanden gemaakt, die AVRstudio gebruikt bij het debuggen van de code.

Met een JTAG-programmer of een gewone seriële programmer kan de microcontroller rechtstreeks vanuit AVRstudio geprogrammeerd worden, mits AVRstudio de programmer herkent. De JTAG-programmer kan — mits deze daarvoor geschikt is — ook gebruikt worden om met AVRstudio de microcontroller te debuggen.

# 11

# Led Blink

**Doelstelling**

Dit hoofdstuk start het meest eenvoudige standaardvoorbeeld voor een microcontrollerprogramma: het laten knipperen van een led. Je leert hoe de in- en uitgangen van een microcontroller werken en hoe deze ingesteld worden. Vervolgens leer je hoe je meerdere leds aanstuurt.

**Onderwerpen**

De behandelde onderwerpen zijn:

- De minimale aansluiting voor de microcontroller.
- De aansluiting van een led aan de microcontroller.
- De opbouw van de generieke IO van de ATmega32: het `PORT`-, `DDR`- en het `PIN`-register.
- De adressering van het `PORT`-, `DDR`- en het `PIN`-register.
- Het maken van vertragingstijden met behulp van herhalingslussen: `_delay_loop1`, `_delay_loop2`, `_delay_us` en `_delay_ms`.
- De instelling van de klokfrequentie met de constante `F_CPU`.
- Het aansturen van leds met een transistor.
- Het gebruik van een opzoektabel.
- Het gebruik van timer `0` met een interruptfunctie.
- Het gebruik van het programmageheugen om gegevens op te slaan.
- De werking en aansturing van een 7-segmentdisplay.
- Het gebruik van de functie `rand()`

Drie voorbeelden demonstreren het knipperen van de led. Deze voorbeelden tonen drie verschillende methoden om een tijdvertraging te maken:

- Met een eigen **`for`**-lus.
- Met de macro `_delay_loop2`.
- Het de macro `_delay_ms`.

Vier voorbeelden demonstreren de aansturing van meerdere leds.

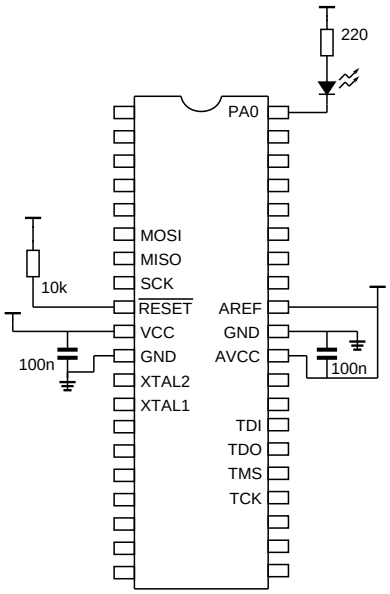
- Cijfers tonen op een dotmatrix.
- Cijfers tonen op een dotmatrix met een interruptfunctie en een timer.
- Cijfers tonen op een dotmatrix met een opzoektabel in flash.
- Een willekeurige getal tonen op een 7-segmentdisplay.

Een groot verschil tussen een programma voor een microcontroller en een programma voor een pc is dat er bij een programma voor een microcontroller geen gewone in- en uitvoer is. Aan de microcontroller zit geen toetsenbord, geen muis en geen beeldscherm. Dit verschil heeft twee belangrijke consequenties:

- Het microcontrollerprogramma moet op een ander machine worden gemaakt. De compiler op deze andere machine moet weten hoe de microcontroller in elkaar zit. Een compiler waarmee een programma voor een andere machine wordt ontwikkeld, heet een *crosscompiler*.
- De ontwerper moet zelf zijn 'toetsenbord' en 'beeldscherm' maken om het programma fysiek te kunnen testen. Er is dus een ontwikkelbord of een specifieke demonstratieopstelling nodig. Een alternatief is om een simulator of *debugger* te gebruiken.

De *debugger* wordt ook wel simulator genoemd. Sommigen gebruiken het woord *debuggen* alleen bij het debuggen van een microcontroller via de JTAG-interface en spreken van *simuleren* als de microcontroller wordt gesimuleerd. Anderen gebruiken in beide gevallen hiervoor het woord debuggen. In beide gevallen wordt de ontwikkelomgeving immers gebruikt om fouten uit de code te halen. Formeel is een debugger een computerprogramma om fouten uit andere programma's te halen en is een simulator een instrument of een programma dat een apparaat of applicatie nabootst.

Elk hoofdstuk behandelt een aantal problemen bij het programmeren van een microcontroller. Voor deze problemen is een specifieke schakeling nodig om het ontwerp te kunnen testen. Daarom wordt er steeds een testschakeling besproken voordat de software wordt uitgelegd.




**Figuur 11.1 : Het schema voor het knipperen van een led.**

In het schema van figuur 11.1 is AREF aangesloten aan de voeding. AREF is de analoge referentie voor ADC. Als er geen ADC wordt gebruikt, is het goed om deze pin aan de voeding aan te sluiten. Pas bij gebruik van een ADC een van de methodes uit paragraaf 18.2 toe.

Het 'Led Blink' voorbeeld is de 'Hello World' voor de microcontrollers. In plaats van een standaard output op het beeldscherm is er nu een led die knippert.

## 11.1 De schakeling voor Led Blink

Figuur 11.1 geeft een minimale configuratie om de software te testen. De led is aangesloten op pin `PA0`. De weerstand van 220 Ω beperkt de stroom door de led en de microcontroller. De waarde van de weerstand hangt af van de voedingsspanning en de eigenschappen van de led. Bij moderne high intensity leds zal een weerstand van bijvoorbeeld 1 kΩ nodig zijn. Omdat de ATmega32 een actief lage reset heeft, is deze met 10 kΩ weerstand aangesloten aan VCC.
Er is geen kristal aangesloten; er wordt gebruik gemaakt van de interne oscillator. De voedingsspanning is aangesloten op VCC en AVCC en de twee GND-pinnen van de Atmel zijn beide geaard. Tussen VCC en GND en AVCC en GND zijn twee capaciteiten van 100 nF nodig om de stoorsignalen te onderdrukken.
Via de pinnen `TCK`, `TMS`, `TDI` en `TDO` kan de ATmega32 serieel met een JTAG-programmer geprogrammeerd en gedebugd worden. Met de pinnen `SCK`, `MISO` en

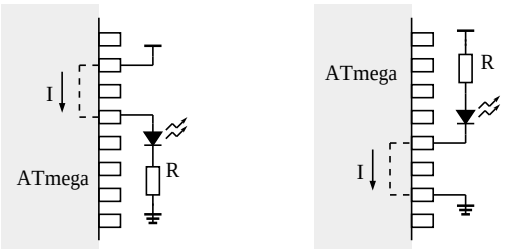
Voor de weerstand $R$ geldt:

$$R = \frac{U_{cc} - U_{led}}{I_{led}}$$

Met $U_{cc}$ 5 V, $U_{led}$ 2 V en $I_{led}$ 15 mA geeft dat 200 Ω. Voor de E12-reeks is dit afgerond naar 220 Ω

In paragraaf 11.5 staat een aantal andere methoden om de led aan te sturen.

MOSI kan de ATmega32 serieel geprogrammeerd worden via een seriële programmer. Debuggen gaat niet met deze verbinding.
De led kan op twee manieren worden aangesloten, namelijk zodanig dat de microcontroller de stroom levert of dat de microcontroller de stroom afvoert. Figuur 11.2 toont deze twee methoden.




**Figuur 11.2 : De stroom door de microcontroller.** De microcontroller moet in de linker situatie (*source*) stroom leveren. In de rechter situatie (*sink*) hoeft de microcontroller de stroom alleen af te voeren.

EMI staat voor elektromagnetische interferentie. (*Electro Magnetic Interference*). Dit is het versturen van en het ontvangen van ongewenste elektromagnetische straling. EMC (*Electro Magnetic Compatibility*) is het vakgebied dat EMI bestrijdt.

Vroeger konden de meeste microcontrollers meer stroom afvoeren (*drain* of *sink*) dan leveren (*source*). Daarom was het gebruikelijk om leds en andere componenten zo aan te sluiten dat de microcontroller de stroom afvoerde. Tegenwoordig kunnen microcontrollers meestal evenveel stroom leveren als afvoeren. De ATmega32 kan per pin 20 mA leveren bij 5 V en 10 mA leveren bij een 3 V voedingsspanning. Het is nog steeds beter om de microcontroller stromen te laten afvoeren vanwege allerlei EMI-aspecten.

## 11.2 De software voor Led Blink

Code 11.1 geeft een algemene beschrijving voor het knipperen van een led. Deze code bevat alleen ANSI C en kan daarom ook gebruikt worden bij andere C-compilers voor de ATmega32.

**Code 11.1 : Code, die te gebruiken is bij alle C-compilers, om een led te laten knipperen.**

```
1  #define DDRA  (*(volatile unsigned char *)(0x1A + 0x20))
2  #define PORTA (*(volatile unsigned char *)(0x1B + 0x20))
3
4  int main(void) {
5    unsigned int i;
6
7    DDRA = 0xff;             // all bits port A output
8
9    while(1) {               // endless loop
10     PORTA = 0x00;                  // bits port A low, led is on
11     for(i = 0; i < 0xFFFF; i++);   // delay
12     PORTA = 0xFF;                  // bits port A high, led is off
13     for(i = 0; i < 0xFFFF; i++);   // delay
14   }
15 }
```

Dit is een eenvoudig voorbeeld dat voor alle compilers werkt. Het is alleen geen goede oplossing.
Gebruik voor tijdvertragingen nooit **for**-lussen, maar gebruik altijd een timer of gebruik `_delay_loop1`, `_delay_loop2`, `_delay_us` of `_delay_ms` uit `delay.h`.

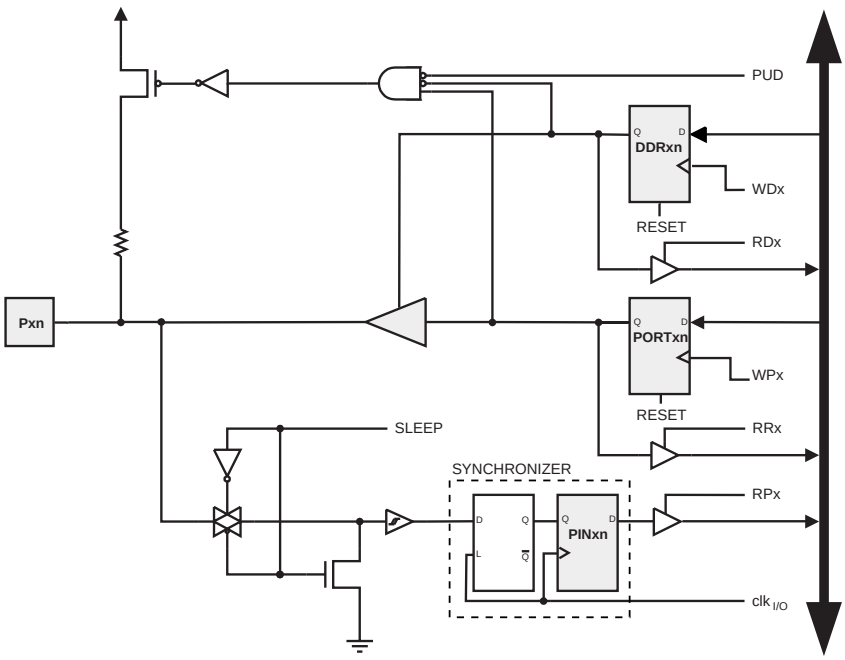
De `main` bestaat na een declaratie en een initialisatie uit een **while**-lus, die oneindig vaak wordt uitgevoerd. De microcontroller zal voortdurend hetzelfde doen, namelijk: de uitgangen van poort A laag maken (regel 10); even wachten (regel 11);

de uitgangen van poort A hoog maken (regel 12) en weer even wachten (regel 13). Omdat het programma voor altijd in de **while**-lus blijft heeft de `main` geen **return**-statement nodig. De oneindige lus is kenmerkend voor een microcontrollerprogramma. De microcontroller voert voortdurend een en hetzelfde programma uit.

Een tijdvertraging maken met een eigen **for**-lus is de meest slechte methode om een tijdvertraging te maken. Deze tijdvertraging is compilerafhankelijk en hangt af van allerlei optimalisatie-opties. Het aantal klokslagen voor een iteratie is bij CodeVision acht, en bij de GNU-compiler is dat twintig zonder optimalisatie (optie `-O0`) en zeven bij een volledig optimalisatie (optie `-Os`). Hoe de compiler de code optimaliseert, kan zelfs afhangen van de eindwaarde. Het resultaat bij `0x7FFF` kan totaal anders zijn bij `0x8000`.

De tijdvertragingen (*delays*) zijn gemaakt met **for**-lussen, die 65536 keer — van `0` tot `0xFFFF` — worden doorlopen. Als de klokfrequentie 1 MHz is en een iteratie van de **for**-lus twintig klokslagen duurt, is de vertragingstijd ongeveer 1,3 s.
De toewijzing op regel 7 zorgt ervoor dat alle acht aansluitingen van poort A uitgang zijn. Als op alle acht aansluitingen een led is aangesloten, zullen alle acht leds tegelijkertijd knipperen. In paragraaf 11.4 en 11.11 wordt uitgelegd hoe de individuele aansluitingen van een poort in- of uitgang gemaakt kunnen worden.

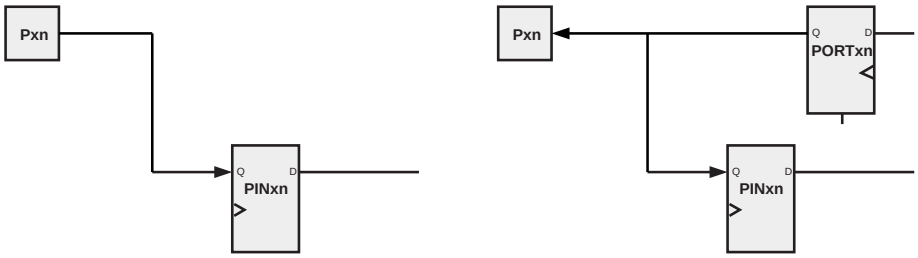
De toewijzing op regel 10 zorgt ervoor dat alle uitgangen van poort A laag worden. Over de led en de weerstand staat nu de voedingsspanning. De led zal gaan branden. Regel 12 maakt alle uitgangen van poort A hoog. Over de led en de weerstand staat nu geen spanning. De led zal uitgaan.




**Figuur 11.3 : De generieke IO van ATmega32.**
Deze figuur is een overdruk van figuur 23 uit de datasheet met een paar aanpassingen. De belangrijkste componenten voor de bespreking van de IO zijn hier grijsgetint, namelijk: de aansluitpin `Pxn`, de tristatebuffer en drie flipfloppen `DDRxn`, `PORTxn` en `PINxn`. Linksboven staat een pulluptransistor met een pullupweerstand om de ingangen hoog te maken. Linksonder staat een transmissiepoort met een pulldowntransistor voor de slaapmodes en een schmitttrigger.

De eerste twee regels in code 11.1 definiëren twee registers `DDRA` en `PORTA`. In figuur 11.3 is de generieke IO getekend. In werkelijkheid heeft elke IO-poort nog een of meer andere specifieke functies. Deze figuur geeft alleen de algemene functionaliteit, die direct met de in- en uitgangen te maken heeft, en een paar specifieke functies die alle aansluitingen hebben.

Afgebeeld is de IO van een willekeurige aansluiting. Bij de ATmega32 kan dat bit 0 tot en met 7 van poort A, B, C of D zijn. De x in de naam bij de flipflop-pen DDRxn, PORTxn en PINxn slaat op de letter van de poort en de n slaat op het bitnummer. DDRxn is een flipflop uit het Data Direction Register van poort x. Als deze flipflop hoog is, is de grijsgetinte tristatebuffer ingeschakeld (*enabled*). De waarde van flipflop PORTxn komt dan op de aansluitpin Pxn te staan. De IO-poort werkt dan als uitgang. PORTxn is een flipflop uit het uitgangsregister van poort x. Als de DDRxn laag is, is de tristatebuffer uitgeschakeld (*disabled*). Er is dan geen directe verbinding tussen PORTxn en de aansluitpin Pxn. De IO-poort werkt dan als ingang. De aansluitpin is via de transmissiepoort en de schmitttrigger verbon-den met flipflop PINxn. PINxn is een flipflop uit het ingangsregister van poort x. Deze flipflop bevat altijd de waarde, die op de aansluitpin staat. Ook als de IO als uitgang werkt. In dat geval krijgt PINxn dezelfde waarde als PORTxn, zie ook figuur 11.4.




**Figuur 11.4 : De IO van de ATmega32 als ingang en als uitgang.**
Links is de configuratie van figuur 11.3 als DDRxn laag is. De aansluitpin is dan een ingang. De sleepmode-, de pullupschakeling, de synchronisatie latch en de databus zijn hier weggelaten. Rechts staat de configuratie van figuur 11.3 als DDRxn hoog is. De aansluitpin is dan een uitgang en PINxn bevat dan het uitgangssignaal.

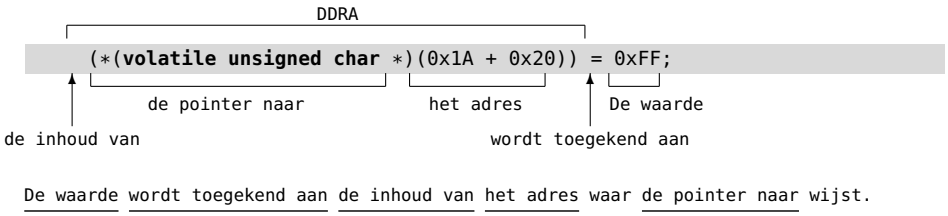
Per poort zijn er acht aansluitpinnen en zijn er dus drie achtbits registers: DDRx, PORTx en PINx. Tabel 11.1 geeft het deel van de *registry summary* uit de datasheet. Er zijn vier poorten dus zijn twaalf IO-registers.

**Tabel 11.1 : De adressen van de in- en uitgangsregisters uit de *register summary* van de datasheet.**

| Nummer | Adres | Naam | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|---|---|---|---|---|---|---|---|---|---|---|
| $1B | $3B | PORTA | PORTA7 | PORTA6 | PORTA5 | PORTA4 | PORTA3 | PORTA2 | PORTA1 | PORTA0 |
| $1A | $3A | DDRA | DDRA7 | DDRA6 | DDRA5 | DDRA4 | DDRA3 | DDRA2 | DDRA1 | DDRA0 |
| $19 | $39 | PINA | PINA7 | PINA6 | PINA5 | PINA4 | PINA3 | PINA2 | PINA1 | PINA0 |
| $18 | $38 | PORTB | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 |
| $17 | $37 | DDRB | DDRB7 | DDRB6 | DDRB5 | DDRB4 | DDRB3 | DDRB2 | DDRB1 | DDRB0 |
| $16 | $36 | PINB | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 |
| $15 | $35 | PORTC | PORTC7 | PORTC6 | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1 | PORTC0 |
| $14 | $34 | DDRC | DDRC7 | DDRC6 | DDRC5 | DDRC4 | DDRC3 | DDRC2 | DDRC1 | DDRC0 |
| $13 | $33 | PINC | PINC7 | PINC6 | PINC5 | PINC4 | PINC3 | PINC2 | PINC1 | PINC0 |
| $12 | $32 | PORTD | PORTD7 | PORTD6 | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1 | PORTD0 |
| $12 | $31 | DDRD | DDRD7 | DDRD6 | DDRD5 | DDRD4 | DDRD3 | DDRD2 | DDRD1 | DDRD0 |
| $10 | $30 | PIND | PIND7 | PIND6 | PIND5 | PIND4 | PIND3 | PIND2 | PIND1 | PIND0 |

Om alle bits van poort A uitgang te maken, moeten alle bits van register DDRA hoog worden. Het adres van register DDRA is 0x3A. De inhoud van dit adres moet dus 0xFF worden. Regel 1 van code 11.1 definieert een macro DDRA. Op

regel 7 wordt deze macro gebruikt. Deze toewijzing is dus een vereenvoudigde schrijfwijze voor de code uit figuur 11.5. De betekenis van deze code is: ken de waarde (`0xFF`) toe aan aan de inhoud (`*`) van het adres (`0x1A+0x1B`) waar de pointer (**`volatile unsigned char *`**) naar wijst.




**Figuur 11.5 : Met de macrodefinitie `DDRA` kunnen waarden aan register `DDRA` worden toegekend.**

Andere compilers gebruiken andere headerbestanden met andere of anders gedefinieerde macro's. Bij CodeVision gebruikt bij de ATmega32 een headerbestand `mega32.h`. Voor elke andere type microcontroller is een ander headerbestand nodig.
Bij de GNU-compiler is bekend welk microcontroller gebruikt wordt. Het headerbestand `avr/io.h` verwijst bij een ATmega32 door naar het bestand `avr/iom32.h`. Bij het overstappen naar een andere microcontroller hoeven de c-bestanden niet te worden aangepast.

De macro `DDRA` definieert de geheugenplek van register `DDRA`. De toewijzing `DDRA = 0xFF` is veel beter leesbaar dan de code uit figuur 11.5. Elk compilersysteem bevat headerbestanden met dit soort macrodefinities. Bij de GCC-compiler van WinAVR is dat een bestand `avr/io.h`. Dit bestand bevat voor alle 64 IO-registers soortgelijke definities. De namen van de registers zijn de namen, die worden gebruikt in het *register summary* van de datasheet. Het statusregister (registernummer `$3F`, adres `$5F`) is hierin bijvoorbeeld gedefinieerd als `SREG`. Naast de registers bevat het headerbestand ook definities voor de bits van de verschillende registers. Zo is bijvoorbeeld `INT0` voor bit 6 van register `GICR`. Tenslotte bevat `avr/io.h` de namen voor de interruptvectoren, bijvoorbeeld `INT0_vect` voor interrupt `0`.

## 11.3 Led Blink met _delay_loop2

Een veel betere methode om een tijdvertraging te maken is om een standaard oplossing te gebruiken. De AVR GNU-compiler heeft een headerbestand `delay.h` met twee vertragingen `_delay_loop_1` en `_delay_loop_2`. Deze vertragingen zijn zo geschreven dat de compiler deze niet optimaliseert. De vertragingstijd is wel afhankelijk van de klokfrequentie. Een snelle klok levert kortere vertragingstijden op als een langzame klok.

Zelf tijdvertragingen met **`for`**-lussen knutselen is niet raadzaam. Een alternatief is om een van de hier beschreven macro's voor de tijdvertraging te gebruiken. Een nadeel is dat macro's slecht overdraagbaar zijn. Alle compilers (CodeVision, IAR) kennen delay macro's, alleen zijn de namen anders. CodeVision kent bijvoorbeeld een macro `delay_ms` en IAR een macro `_delay_cycles`.

De tijdvertraging `_delay_loop1` gebruikt een 8-bits unsigned integer. Per iteratie zijn drie klokslagen nodig. Voor 8-bits unsigned integer zijn er maximaal 256 iteraties. Bij een klokfrequentie van 1 MHz is de vertraging maximaal 768 $\mu$s.
De tijdvertraging `_delay_loop2` gebruikt een 16-bits unsigned integer. Per iteratie zijn vier klokslagen nodig. Voor 16-bits unsigned integer zijn er maximaal 65536 iteraties. Bij een klokfrequentie van 1 MHz is de vertraging zodoende maximaal 262,1 ms.
Code 11.2 komt geheel overeen met die van code 11.1. Alleen zijn de **`for`**-lussen vervangen door `_delay_loop2`. Het aantal iteraties van de delaylussen is vier en de tijdvertraging bij een klokfrequentie van 1 MHz zal ongeveer 262 ms zijn.

**Uitleg code 11.2 regel 1**
`avr/io.h`

Het headerbestand `io.h` bevat de defines en typedefinities, die nodig zijn om de IO- en andere registers te definiëren. In principe is het adres van een register niets anders dan een getal. Het adres van het DDR-register van port A is `0x3A` of `0x1A`.

**Code 11.2 : Led knipperen met delay_loop_2.**

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  int main(void) {
5    DDRA = 0xff;               // all bits port A output
6
7    while(1) {
8      PORTA = 0x00;            // bits port A low, led is on
9      _delay_loop_2(0xFFFF);
10     PORTA = 0xFF;            // bits port B low, led is off
11     _delay_loop_2(0xFFFF);
12   }
13 }
```

Het is natuurlijk veel plezieriger om deze adressen een naam te geven. De code wordt daardoor veel leesbaarder. In de makefile, die AVRstudio maakt, staat — mits er voor een ATmega32 gekozen is — deze regel:

```
MCU = atmega32
```

In de makefile wordt deze optie `-mmcu=$(MCU)` aan de compiler meegegeven. In `io.h` staat een hele **`#if #elif #else #endif`**-constructie met alle AVR-microcontrollers. In dit geval wordt onder water `iom32.h` aangeroepen met alle definities voor de ATmega32.

**Regel 5**
`DDRx`
`PORTx`
`PINx`

De ATmega32 heeft vier 8-bits IO-poorten. Elke poort kent — zoals uitgebreid beschreven is in paragraaf 11.2 — drie registers:

- `DDRx` het Data Direction Register
- `PORTx` het uitgangsregister
- `PINx` het ingangsregister

Hierbij staat `x` voor A, B, C en D. In `avr/io.h` zijn `DDRA` en de andere registers op deze manier gedefineerd:

```
#define DDRA   _SFR_IO8 (0x1A)
```

In `io.h` wordt een headerbestand `sfr_defs.h` aangeroepen met daarin:

```
#define _MMIO_BYTE(mem_addr)  (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr)     _MMIO_BYTE((io_addr) + 0x20)
```

`DDRA` is de inhoud van het adres waar de pointer (**`volatile`** `uint8_t *`) naar wijst. Deze pointer wijst naar adres `0x1A + 0x20 = 0x3A`.

`uint8_t`
`stdint.h`

In de definitie `_MMIO_BYTE` en op andere plaatsen staan soms typen van deze vorm `[u]int[number]_t`, zoals `int8_t`, `uint8_t`, `int16_t` en `uint16_t`. Deze typen zijn vanaf ISO C99 gedefinieerd in `stdint.h` en zijn bedoeld om unsigned en signed integers van een bepaalde bitbreedte te maken. Dat voorkomt misverstanden over het exacte formaat van integers, zoals bij een **`unsigned int`** en een **`unsigned short`** kunnen optreden.

**`volatile`**

In de definitie `_MMIO_BYTE` en in code 11.1 wordt het gereserveerde woord **`volatile`** gebruikt. Dit woord zorgt ervoor dat de compiler een stuk code niet optimaliseert. Het Engelse woord *volatile* kan op vele manieren worden vertaald, bijvoorbeeld met: vluchtig, wispelturig, wisselvallig en onzeker. In dit verband betekent het dat het voor de compiler onzeker is wat er met de variabele of met het stuk code gedaan moet worden. De compiler zal daarom de code alleen vertalen en niet optimaliseren.

In deze code:

```
#define DDRA                 (*(volatile unsigned char *)(0x1A + 0x20))
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
```

blijven de pointers **unsigned char*** en uint8_t* bij het optimaliseren gegarandeerd ook echte pointers naar de geheugenadressen 0x1A + 0x20 en mem_addr.

**Regel 5**
//

Commentaar is al eerder besproken op bladzijde 62 in hoofdstuk 8. Daar is gesteld dat commentaar vaak overbodig is. In dit geval is op deze regel 5 commentaar toegevoegd, omdat in dit voorbeeld de hardware en de software van elkaar verschillen. In de hardware is maar één led aangesloten en in de software worden acht bits aangestuurd. Op regel 8 en op regel 10 is het misschien niet duidelijk dat de led juist aan is als de uitgangen laag zijn en dat de led uit is als de uitgangen hoog zijn.

## 11.4 Led Blink met _delay_ms

In code 11.3 wordt de macro _delay_ms gebruikt om een delay van een halve seconde te maken. Het effect is dat de led een keer per seconde gaat knipperen. Omdat de maximale vertragingstijd bij een klokfrequentie van 1 MHz slechts 262 ms is, is de vertraging van een halve seconde gemaakt met twee vertragingen van 250 ms.

**Uitleg code 11.3 regel 3**
**#define** F_CPU 1000000UL

In delay.h is de constante F_CPU al op precies dezelfde manier gedefinieerd. Als deze regel wordt weggelaten, zal de led ook een keer per seconde knipperen. Voor de duidelijkheid is deze macro toegevoegd. Is daarentegen de klokfrequentie 8 MHz, dan moet F_CPU zo worden aangepast:

```
#define F_CPU 8000000UL
```

De constante F_CPU is de frequentie van de microcontroller.

**Regel 3**
1000000UL

De letters UL achter de constante 1000000 geven aan dat het een **unsigned long** getal betreft.

**Regel 11**
_delay_ms
_delay_us

De macro _delay_ms geeft een tijdvertraging in milliseconden. De constante F_CPU, die de klokfrequentie definieert, moet de juiste waarde hebben. Bij 1 MHz is de maximale vertraging ongeveer 262 ms. De macro _delay_us geeft een tijdvertraging in microseconden. Bij 1 MHz is de maximale vertraging ongeveer 768 $\mu$s.

**Regel 7**
DDRA=0x01;
PORTA=0x01;

De toewijzing DDRA = 0x01 maakt bit 0 in register DDRA hoog. Het gevolg is dat de bits 1 tot en met 7 ingangen zijn en dat de aansluiting 0 een uitgang is. De toewijzing PORTA = 0x01 maakt de uitgang van aansluiting 0 hoog. Een toewijzing PORTA = 0xC1 maakt de flipfloppen 6 en 7 van PORTA ook hoog, maar als deze aansluitingen ingang zijn, heeft dat weinig effect. Alleen kan het zijn dat de pullup van de uitgang actief wordt, zie figuur 11.3.

0b
0x

Soms wordt er 0b11000001 gebuikt in plaats van 0xC1. Dit boek gebruikt altijd de hexadecimale notatie of de bitnotatie uit hoofdstuk 12.

Al deze hier besproken tijdvertragingen zijn gebaseerd op herhalingsopdrachten. Een nadeel hiervan is dat de microcontroller verder niets doet. Een veel belangrijker nadeel is dat interne of externe interrupts de herhalingsopdrachten onderbreken. Op dat moment wordt de tijdvertraging onvoorspelbaar. Het is veel beter om tijdvertragingen met een timer te maken. Dit komt aan de orde in paragraaf 11.8 en in hoofdstuk 13.

**Code 11.3 : Led knippert een keer per seconde.**

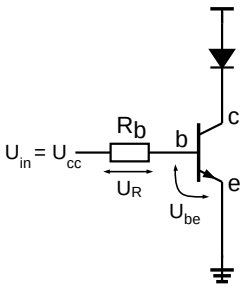
```
1  #include <avr/io.h>
2
3  #define F_CPU 1000000UL
4  #include <util/delay.h>
5
6  int main(void) {
7    DDRA = 0x01;               // bit 0 port A is output
8
9    while(1) {
10     PORTA = 0x00;            // bit 0 port A is low, led is on
11     _delay_ms(250);          // maximum delay with 1 MHz is 262 ms.
12     _delay_ms(250);
13     PORTA = 0x01;            // bit 0 port B is high, led is off
14     _delay_ms(250);          // maximum delay with 1 MHz is 262 ms.
15     _delay_ms(250);
16   }
17 }
```

De definitie `F_CPU` moet voor `#include <util/delay.h>` staan, anders geeft de compiler de waarschuwing dat `F_CPU` niet gedefinieerd is.
Een andere methode is om in AVRstudio bij de configuratie-opties de variabele `Frequency` te definiëren.

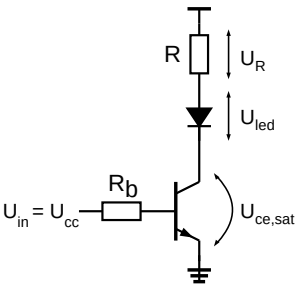
De maximale vertraging is 262 ms/`F_CPU` met de frequentie in MHz. Als de vertragingstijd groter is dan deze maximale waarde, is de nauwkeurigheid slechts 0,1 ms.

## 11.5 Aansturing leds

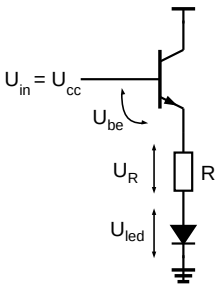
De stroom door de led van figuur 11.1 gaat via de microcontroller naar massa. Per aansluiting kan de ATmega32 niet meer dan 20 mA leveren of afvoeren. Voor alle aansluitingen samen is dat maximaal 200 mA. Het is nuttig om de microcontroller geen of weinig stroom te laten leveren.
De figuren 11.6 tot en met 11.9 geven een aantal configuraties waarbij een transistor de stroom levert.



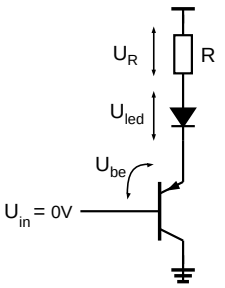

**Figuur 11.6 : Aansturing led met een transistor en een basisweerstand.**




**Figuur 11.7 : Aansturing led met een transistor in verzadiging.**




**Figuur 11.8 : Aansturing led met een NPN-transistor als stroombron.**




**Figuur 11.9 : Aansturing led met een PNP-transistor als stroombron.**

Figuur 11.6 gebruikt een NPN-transistor met een weerstand bij de basis. De uitgang van de ATmega32 wordt aangesloten op de ingang van deze schakeling. Als de uitgang van de microcontroller laag is, spert de transistor en is de led uit. Als de uitgang hoog is, geleidt de transistor en is de led aan.
De basisweerstand kan met de wet van Ohm en $I_c = \beta I_b$ worden berekend:

$$R_b = \frac{U_R}{I_b} = \frac{\beta(U_{cc} - U_{be})}{I_c} \tag{11.1}$$

De voorbeelden in deze paragraaf gebruiken een gewenste stroom en spanning voor de led en een specifieke $\beta$ voor de transistoren.

Deze waarden zijn willekeurig. Er is geen keuze gemaakt voor een type led en transistor. Neem dit niet klakkeloos over. Gebruik de gegevens uit de datasheets van gebruikte componenten.

Bij gemultiplext aansturen van dotmatrices en 7-segmentdisplays, is een hogere stroom door de leds mogelijk en kunnen de weerstandswaarden dus kleiner zijn. Als een ledstroom van 75 mA toelaatbaar is, is $R$ 39 Ωen $R_b$ 2k7.

Als de gewenste stroom door de led 15 mA is, dan is — met $U_{cc}$ 5 V, $U_{be}$ 0,7 V en $\beta$ 200 — de basisweerstand $R_b$ ongeveer 56 kΩ. De basisstroom, die door de microcontroller moet worden geleverd is slechts 75 $\mu$A.
Deze methode wordt vooral toegepast als de leds niet continu branden, bijvoorbeeld bij de multiplexing voor een ledmatrix.

In figuur 11.7 is een extra weerstand $R$ bij de led aangebracht. De led brandt als de ingang hoog is. Bij een juiste keuze van $R_b$ en $R$ is de NPN-transistor in verzadiging. Er geldt dan:

$$R = \frac{U_R}{I_c} = \frac{U_{cc} - U_{ce,sat} - U_{led}}{I_c} \tag{11.2}$$

Met $U_{ce,sat}$ gelijk aan 0,2 V, een ledspanning van 2 V en een ledstroom van 15 mA is $R$ ongeveer 180 Ω.
Voor de basisweerstand geldt:

$$R_b = \frac{U_{cc} - U_{be}}{I_b} = \frac{\beta(U_{cc} - U_{be})}{I_c} \tag{11.3}$$

In verzadiging is $\beta$ vaak een factor 5 lager. De basisweerstand is dan ongeveer 12 kΩ.

De schakeling van figuur 11.8 lijkt op die van figuur 11.7. Er zijn twee verschillen: er is geen basisweerstand en de transistor werkt in het actieve gebied. De led brandt weer als de ingang hoog is. Voor de weerstand $R$ geldt:

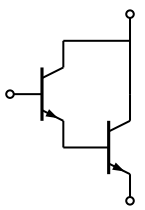
$$R = \frac{U_R}{I_c} = \frac{U_{cc} - U_{be} - U_{led}}{I_c} \tag{11.4}$$

Met $U_{be}$ gelijk aan 0,7 V, een ledspanning van 2 V en een ledstroom van 15 mA is $R$ ongeveer 150 Ω.

Figuur 11.9 is het complement van figuur 11.8. In plaats van een NPN-transistor wordt er nu een PNP-transistor gebruikt. De led brandt nu als de ingang laag is. Voor de weerstand geldt vergelijking 11.4 en dus is $R$ weer 150 Ω.

Duidelijk is dat met een transistor de led op vele manieren kan worden aangestuurd en dat de weerstandswaarden uitgerekend kunnen worden als de parameters van de gebruikte componenten bekend zijn. Als de simulatiemodellen beschikbaar zijn, kunnen de berekende waarden geverifieerd worden met een simulator. Daarnaast is het verstandig een proefopstelling te maken om de ledstroom te meten.

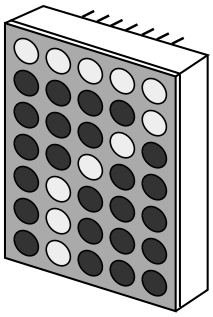
In plaats van een bipolaire transistor kan er voor de aansturing een MOSFET of een darlingtontransistor gebruikt worden. Een darlingtontransistor kan een veel grotere stroom leveren dan een gewone bipolaire transistor. Bij de aansturing van een relay of een motor is een bipolaire transistor niet geschikt. Het voordeel van een MOSFET of een JFET is dat de gatestroom nul is en dat de microcontroller helemaal geen stroom hoeft te leveren. De MOSFET is ook veel stabieler en minder temperatuurafhankelijk dan een bipolaire transistor.
Als er meerdere leds aangestuurd moeten worden, is het een optie om een transistor array toe te passen, zoals bijvoorbeeld de ULN2803A of de TN0604/TP0604. De UN2803A bevat acht darlingtontransistoren. De TN0604 en de TP0604 bevatten respectievelijk vier NMOS- en vier PMOS-transistoren.



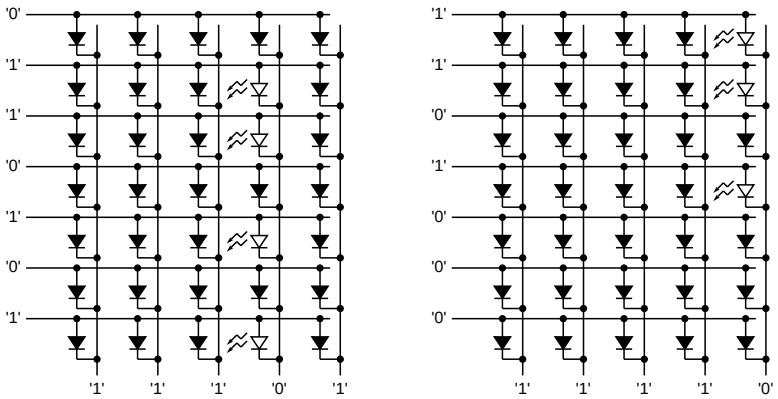
**Figuur 11.10 :**
**De darlingtontransistor.**
Deze transistor bestaat in feite uit twee bipolaire transistoren. De eerste transistor versterkt de stroom een factor $\beta$ en de tweede transistor versterkt deze stroom met nog eens een factor $\beta$. De totale versterking is dan $\beta^2$.

## 11.6 Een led-array of dotmatrix

Een veelgebruikte mogelijkheid om informatie weer te geven is een array van leds. Met een array van vijf bij zeven leds kunnen alle karakters zichtbaar worden gemaakt. Figuur 11.12 toont twee keer een array met vijf kolommen en zeven rijen. In elke rij zijn de anodes en in iedere kolom zijn de kathodes van de leds met elkaar verbonden. In de linker figuur wordt alleen de vierde kolom aangestuurd en in de rechter figuur alleen de vijfde kolom. Door na elkaar steeds een andere kolom aan te sturen branden de leds van elke kolom een deel van de tijd.



**Figuur 11.11 : Een 5x7 dotmatrix met een array van vijf bij zeven leds.**



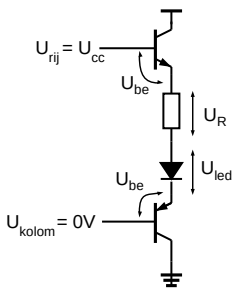

**Figuur 11.12 : Het principe van een ledarray.** Bij de linker array wordt de vierde kolom aangestuurd. In deze kolom lichten de leds op waarvan de anode hoog is. Bij de rechter array wordt de vijfde kolom aangestuurd en lichten de leds op waarvan nu de anode hoog is.

Dit effect wordt gebruikt bij het maken van een televisiebeeld. Tot voor kort was in Europa 50 Hz de standaard voor televisies en monitoren. Tegenwoordig is dat vaak 100 Hz.

Het menselijk oog kan niet meer dan vijfentwintig beelden per seconde onderscheiden. Mits de afbeelding op het display minimaal vijfentwintig keer per seconde ververst wordt, ziet men een stabiel beeld. Er zijn vijf kolommen. Elk kolom brandt dus maar een vijfde van de tijd. Stel dat het hele beeld ververst wordt met een frequentie van 50 Hz, dan wordt het beeld elke 20 ms opnieuw geschreven. Elke kolom is dan 4 ms actief.

De tijd dat een kolom actief is, noemen we $t_{\text{active}}$. In dit voorbeeld is $t_{\text{active}}$ 4 ms.

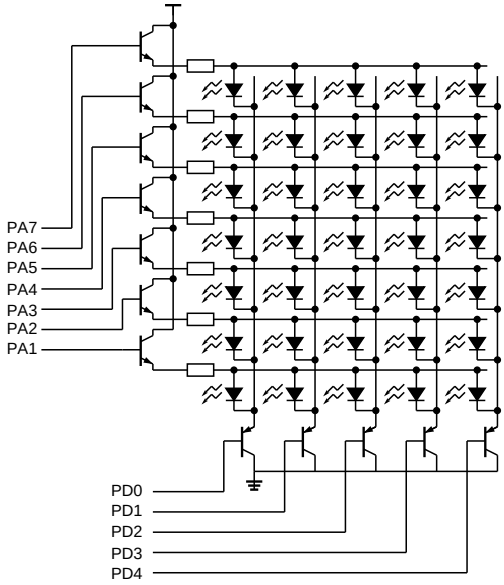
Figuur 11.14 laat zien dat de leds niet rechtstreeks vanuit de microcontroller worden aangestuurd maar via zeven NPN-transistoren. Er kunnen dus steeds zeven leds te gelijk branden, die met hun kathode aan een PNP-transistor zijn verbonden. De stroom door de collector van de PNP-transistor hangt af van het aantal leds dat brandt. De weerstand bij de NPN-transistor zorgt ervoor dat de ledstroom niet te groot wordt. Deze stroom hangt alleen af van de weerstand, de voedingsspanning, de ledspanning $U_{\text{led}}$ en de $U_{\text{be}}$ van de transistoren. In figuur 11.13 staat de aansturing voor een enkele led. Voor de weerstand $R$ geldt:



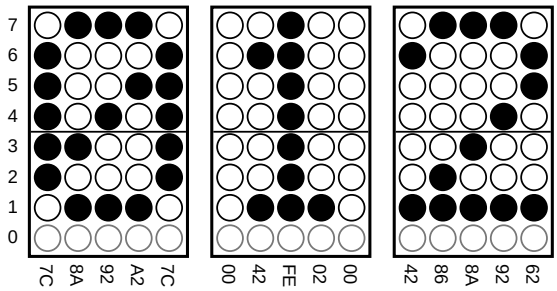

**Figuur 11.13 : De aansturing van een led uit de matrix.**

$$R = \frac{U_{\text{R}}}{I_{\text{led}}} = \frac{U_{\text{cc}} - 2U_{\text{be}} - U_{\text{led}}}{I_{\text{led}}} \tag{11.5}$$

Bij een gewenste stroom van 15 mA is — met $U_{\text{cc}}$ 5 V, $U_{\text{be}}$ 0,7 V en $U_{\text{led}}$ 2 V — de weerstand ongeveer 100 Ω. Omdat de led maar een vijfde van de tijd aan is, kan de ledstroom groter zijn. Mits de DC-stroom maar 15 mA blijft, is een maximale stroom van 100 mA mogelijk. Met een weerstand van 22 Ω werken de leds optimaal.




**Figuur 11.14 :** **De aansturing bij een ledarray of een dotmatrix.** De kathodes van de leds zijn per kolom verbonden via een PNP-transistor verbonden met massa. De basis van deze transistoren worden aangestuurd vanuit poort D van de microcontroller. De anodes van de leds zijn via een PNP-transistor per rij verbonden met de voeding. De basis van deze transistoren worden aangestuurd vanuit poort A van de microcontroller.




**Figuur 11.15 :** **De patronen voor de cijfers `0`, `1` en `2` voor een 5x8-array.** De hexadecimale waarden waarmee de rijen worden aangestuurd staan onder de dots. Het meest significante bit hoort bij de bovenste led.

Als er matrices met veel leds of als meerdere leddisplays moeten worden aangestuurd, is het gebruik van een seriële leddisplay driver zoals de A6275 van Allegro of MAX6952 van Maxim een alternatief. Er zijn vele soorten leddrivers beschikbaar.

## 11.7 Cijfers afbeelden op een dotmatrix

In figuur 11.15 staan de patronen van de cijfers `0`, `1` en `2` bij een dotmatrix. De waarden van de rijen worden in het programma opgeslagen als bytes. In dit geval is er voor gekozen om de bits links uit te lijnen. Dat betekent dat alleen de bits 1 tot en met 7 worden gebruikt en dat bit `0` niet gebruikt wordt.
Het algoritme om een karakter af te beelden, bestaat uit een herhalingslus waar steeds een kolom laag gemaakt wordt, de juiste code voor de rijen op de betreffende uitgangen van de microcontroller staat en waarna er gedurende een korte tijd gewacht:

```
doe voor elke kolom:
  zet juiste code op uitgangen voor de rij
  maak betreffende kolom laag
  wacht een tijd t_active
```

Een opzoektabel is een lijst van een sleutels (*keys*) met waarde (*value*). Met de sleutel kan de waarde in de lijst worden gevonden.
In dit geval is de opzoektabel geïmplementeerd als een tweedimensionaal array. De sleutel bestaat uit twee indices: een voor de kolommen en een voor de karakters.

Een zogenoemde opzoektabel (*look-up table*) is heel geschikt om de waarden voor de rijen te bewaren. Deze waarden zijn met een rij- en een kolomindex direct benaderbaar. Figuur 11.15 laat zien dat voor het karakter '0' de waarden `0x7C`, `0x8A`, `0x92`, `0xA2` en `0x7C` nodig zijn.

Het programma van code 11.4 toont op een ledarray of een dotmatrix iedere seconde achtereenvolgens de cijfers `0` tot en met 9. Het programma is een implementatie van het hiervoor besproken algoritme. De rijen zijn aangesloten op de meest significante zeven bits van `PORTA` en de kolommen aan de vijf minst significante bits van `PORTD`. Zie ook figuur 11.14.

**Code 11.4 : De cijfers `0` tot en met 9 tonen met een dotmatrix of een ledarray.**

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  const unsigned char lookup[][5] = {
5    {0x7C, 0x8A, 0x92, 0xA2, 0x7C}, // 0
6    {0x00, 0x42, 0xFE, 0x02, 0x00}, // 1
7    {0x42, 0x86, 0x8A, 0x92, 0x62}, // 2
8    {0x84, 0x82, 0xA2, 0xD2, 0x8C}, // 3
9    {0x18, 0x28, 0x48, 0xFE, 0x08}, // 4
10   {0xE4, 0xA2, 0xA2, 0xA2, 0x9C}, // 5
11   {0x3C, 0x52, 0x92, 0x92, 0x0C}, // 6
12   {0x80, 0x80, 0x9E, 0xA0, 0xC0}, // 7
13   {0x6C, 0x92, 0x92, 0x92, 0x6C}, // 8
14   {0x60, 0x92, 0x92, 0x94, 0x78}  // 9
15 };
16
```

```
17 int main(void)
18 {
19   int i,t,col;
20
21   DDRA  = 0xFE;
22   DDRD  = 0x1F;
23   PORTD = 0x1F;   // column outputs high
24
25   i=0;
26   while(1) {
27     for (col=0; col<5; col++) {
28       PORTA = lookup[i][col] & 0xFE;
29       PORTD = ~(1<<col) & 0x1F;
30       _delay_ms(4);
31     }
32     t++;
33     if (t == 50) {        // next digit
34       i++;
35       if (i == 10) i = 0; // restart digit
36       t = 0;
37     }
38   }
39 }
```

**Uitleg code 11.4 regel 4-15**

Het tweedimensionale array `lookup` bevat tien keer de vijf waarden die nodig zijn om de karakters `0` tot en met 9 af te beelden. De waarden zijn gedefineerd als **`unsigned char`**. Het sleutelwoord **`const`** vertelt de compiler dat de array niet gewijzigd mag worden na de initialisatie.

Het algoritme, dat het karakter afbeeldt, kent op regel 28 aan `PORTA` het arrayelement `lookup[i][col]` toe. Variabele `i` is de index van het karakter en `col` is de index voor de kolom. Omdat er maar zeven bits nodig zijn, is de waarde eerst nog gemaskeerd met `0xFE`. Op regel 29 wordt de betreffende kolom laag (`~(1<<col)`) gemaakt en gemaskeerd met `0x1F` omdat er maar vijf kolommen zijn. Hier staat een voorbeeld met `col=3`:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| `1` | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| `1<<3` | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| `~(1<<3)` | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| `0x1F` | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| `~(1<<3) & 0x1F` | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

De vertragingstijd $t_{\text{active}}$ is 4 ms. Het karakter wordt dan iedere 20 ms ververst.

**Uitleg code 11.4 regel 32-37**

Elke seconde toont het programma een ander cijfer. Omdat de karakters om de 20 ms verversen, moet na vijftig keer verversen het volgende karakter worden geselecteerd. De variabele `t` telt het aantal keer dat het karakter is ververst. Als `t` vijftig is, wordt deze weer nul gemaakt en wordt de index `i` opgehoogd.

**Uitleg code 11.4 regel 23**

Bij de initialisatie zijn de vijf kolomuitgangen hoog gemaakt. Dit zorgt er voor dat de leds aanvankelijk allemaal uit zijn.

## 11.8 Cijfers afbeelden op een dotmatrix met interrupt en timer

Een probleem bij de oplossing uit de vorige paragraaf is dat het hoofdprogramma voortdurend staat te wachten. Als er tussendoor een andere omvangrijke taak verricht moet worden, werkt de applicatie niet correct. Bovendien is het moeilijk om te bedenken wanneer welke taak verricht moet worden en is het zeer lastig om taken toe te voegen. In ons voorbeeld zijn er twee taken: elke 4 ms moet er een kolom met leds worden aangestuurd en daarna wordt de variabele `t` gewijzigd en eens per seconde ook de variabele `i`.

Een betere methode is om een interrupt te gebruiken. Interrupts komen in hoofdstuk 12 uitgebreid aan de orde. Een interrupt zorgt ervoor dat het hoofdprogramma even onderbroken wordt om een interruptfunctie (*interrupt service routine*) uit te voeren. Als de interruptfunctie klaar is, gaat het hoofdprogramma verder waar het gebleven was.

De ATmega32 heeft een drietal timers (tellers) die een interrupt kunnen geven. In code 11.5 reageert de interruptfunctie op de *overflow* van timer `0`. Timer `0` is een 8-bits teller en wordt via een klokdeler aangestuurd. De deelfactor is ingesteld op 64. Om de 16384 ($=2^8 \times 64$) klokslagen wordt er een interrupt gegenereerd. Bij een klokfrequentie $f_{\text{cpu}}$ van 4 MHz is de klokperiode 250 ns en is er dus elke 4,096 ms een interrupt.

**Uitleg code 11.5 regel 20-31**
**ISR**
**static**

De interruptfunctie (`ISR`) staat op regel 20 en zet steeds de informatie voor de leds klaar, maakt de betreffende kolom actief en bepaalt het volgende kolomindex `col`. De variabele `col` is **`static`**. Dit betekent dat na afloop van de functie de huidige waarde van de variabele bewaard blijft.

**Uitleg code 11.5 regel 18**
**volatile**

Het hoofdprogramma geeft via de pointervariabele `p` aan de interruptfunctie door welk karakter afgedrukt moet worden. Pointer `p` wijst naar een van de sets met waarden in de opzoektabel. Om te voorkomen dat de compiler bij de optimalisatie de variabele `p` kwijt raakt, wordt bij de declaratie van `p` het woord **`volatile`** toegevoegd. De positie van **`volatile`** is heel belangrijk. In dit geval is `p` een volatile pointer:

```
volatile unsigned char *p;   // p is pointer to volatile unsigned char
unsigned char* volatile p;   // p is volatile pointer to unsigned char
```

**Uitleg code 11.5 regel 46-49**

Het hoofdprogramma voert voortdurend een **`for`**-lus uit, die steeds `p` naar een andere set gegevens in de opzoektabel laat wijzen en daarna 1 ms wacht.
Omdat de vertraging groter is dan 262,14 ms/$f_{\text{cpu}}$ (in MHz) kan de vertraging enigszins afwijken van 1 ms. In dit geval is er geen nauwkeurige tijdvertraging nodig, dus voldoet de macro `_delay_ms`.

**Uitleg code 11.5 regel 40-41**
`TCCR0`
`TIMSK`

De instellingen van de registers `TCCR0` en `TIMSK` voor timer `0` worden besproken in hoofdstuk 13. Register `TCCR0` stelt onder andere de prescaling van timer `0` in en in register `TIMSK` wordt het bit geselecteerd dat er een interrupt is als de timer een overflow geeft.

**Uitleg code 11.5 regel 40**
`sei()`

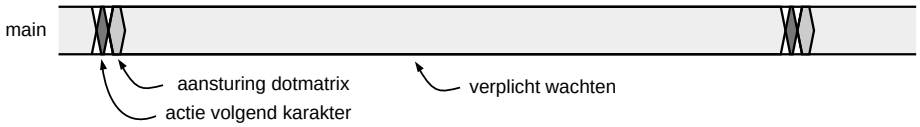
De macro `sei()` zet het interruptmechanisme aan en is gedefineerd in `interrupt.h`, die op regel 2 is ingesloten. Deze functie en het hele interruptmechanisme komt in hoofdstuk 12 aan de orde.

**Code 11.5 : Cijfers 0 tot en met 9 op dotmatrix met behulp van timer 0. ($f_{cpu} = 4$ MHz)**

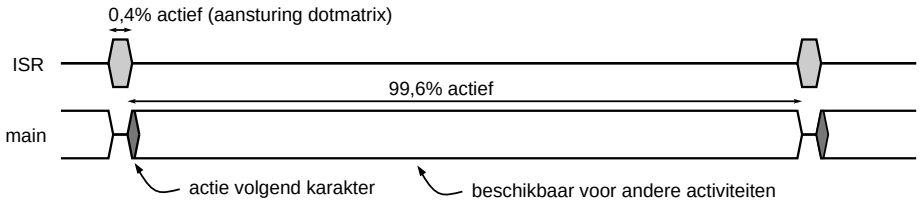
```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <util/delay.h>
4
5  const unsigned char lookup[][5] = {
6    {0x7C, 0x8A, 0x92, 0xA2, 0x7C}, // 0
7    {0x00, 0x42, 0xFE, 0x02, 0x00}, // 1
8    {0x42, 0x86, 0x8A, 0x92, 0x62}, // 2
9    {0x84, 0x82, 0xA2, 0xD2, 0x8C}, // 3
10   {0x18, 0x28, 0x48, 0xFE, 0x08}, // 4
11   {0xE4, 0xA2, 0xA2, 0xA2, 0x9C}, // 5
12   {0x3C, 0x52, 0x92, 0x92, 0x0C}, // 6
13   {0x80, 0x80, 0x9E, 0xA0, 0xC0}, // 7
14   {0x6C, 0x92, 0x92, 0x92, 0x6C}, // 8
15   {0x60, 0x92, 0x92, 0x94, 0x78}  // 9
16 };
17
18 unsigned char* volatile p;
19
20 ISR(TIMER0_OVF_vect)
21 {
22   static int col;
23
24   PORTA = *(p + col) & 0xFE;
25   PORTD = ~_BV(col) & 0x1F;
26   if (col==4) {
27     col = 0;
28   } else {
29     col++;
30   }
31 }
```

```
33 int main(void)
34 {
35   int i;
36
37   DDRA=0xFE;
38   DDRD=0x1F;
39   PORTD=0x1F;                  // column outputs high
40   TCCR0 = _BV(CS01)|_BV(CS00); // prescaling 64
41   TIMSK = _BV(TOIE0);
42
43   sei();
44
45   while(1) {
46     for (i=0; i<=9; i++) {
47       p = (unsigned char *) lookup[i];
48       _delay_ms(1000);
49     }
50   }
51 }
```

Het voordeel van de methode met de timer en de interrupt is dat het hoofdprogramma ontlast wordt. Figuur 11.16 visualiseert de tijdplanning (*scheduling*) voor de oplossing zonder interrupt en figuur 11.17 laat de tijdplanning (*scheduling*) zien voor de oplossing met een interrupt.
Het hoofdprogramma van de oplossing zonder interrupt staat voortdurend te wachten op het volgende moment dat een kolom van de dotmatrix moet worden aangestuurd. Zeker als er meerdere taken uitgevoerd moeten worden, is de planning van deze taken lastig.




**Figuur 11.16 : De tijdplanning bij de oplossing van code 11.4 zonder de interrupt.** Het hoofdprogramma wacht voortdurend op de volgende keer dat er een kolom geschreven moet worden.




**Figuur 11.17 : De tijdplanning bij de oplossing van code 11.5 met de interrupt.** De interruptfunctie regelt de aansturing van de dotmatrix en het hoofdprogramma hoeft nu alleen het volgende karakter te selecteren.

Bij de oplossing met de interrupt en de timer doet het hoofdprogramma bijna niets. Voor de interruptfunctie zijn ruim zestig klokslagen nodig. Na 16384 klokslagen wordt een volgende kolom geschreven. Dit betekent dat er voor de interruptfunctie ongeveer 0,4% van de tijd nodig is. Het hoofdprogramma wacht weliswaar steeds een seconde, maar het bemoeit zich niet met de aansturing van de dotmatrix. Het wisselt alleen de karakters. Dit wisselen kan natuurlijk ook met dezelfde of een andere timer geregeld worden en dan doet het hoofdprogramma zelfs helemaal niets.

## 11.9 Cijfers afbeelden op een dotmatrix met de gegevens in flash

De compiler slaat de gegevens uit de opzoektabel uit code 11.5 op in RAM. In dit geval zijn daar vijftig bytes voor nodig. Het RAM-geheugen van de ATmega32 is ongeveer 2 kB groot. Zeker als er meer karakters of karakters met meer pixels gebruikt worden, kan dit een grote aanslag op de beschikbare dataruimte betekenen. Het is verstandig om opzoektabellen niet in RAM, maar in het programmageheugen te plaatsen. Het flashgeheugen van de ATmega32 is immers 32 kB groot.

De opzoektabel in code 11.6 komt in flash te staan. In dit geval is er voor gekozen om deze op te slaan als een eendimensionaal array. De reden is dat het verwijzen met een pointer naar een eendimensionaal array veel eenvoudiger is dan het verwijzen naar een tweedimensionaal array. Pointer `ptr` wijst naar het begin van de tabel en pointer `ptr + 5*row + col` wijst naar de waarde in kolom `col` van rij `row`.

Het hoofdprogramma van code 11.6 past alleen de index van het karakter aan dat getoond moet worden. Deze index is in feite het rijnummer uit de tabel. Deze index wordt opgeslagen in de globale variabele `show_char`.

De interruptfunctie past het rijnummer aan als alle kolommen geschreven zijn. Dus als kolomnummer 4 is, wordt `show_char` toegekend aan `row`.

**Code 11.6 : Cijfers op dotmatrix met de opzoektabel in flash. ($f_{cpu} = 4$ MHz)**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3  #include <util/delay.h>
 4  #include <avr/pgmspace.h>
 5
 6  const prog_uchar lookup[] = {
 7    0x7C, 0x8A, 0x92, 0xA2, 0x7C, // 0
 8    0x00, 0x42, 0xFE, 0x02, 0x00, // 1
 9    0x42, 0x86, 0x8A, 0x92, 0x62, // 2
10    0x84, 0x82, 0xA2, 0xD2, 0x8C, // 3
11    0x18, 0x28, 0x48, 0xFE, 0x08, // 4
12    0xE4, 0xA2, 0xA2, 0xA2, 0x9C, // 5
13    0x3C, 0x52, 0x92, 0x92, 0x0C, // 6
14    0x80, 0x80, 0x9E, 0xA0, 0xC0, // 7
15    0x6C, 0x92, 0x92, 0x92, 0x6C, // 8
16    0x60, 0x92, 0x92, 0x94, 0x78  // 9
17  };
18
19  const prog_uchar* ptr = lookup;
20  volatile int show_char= 0;
21
22  ISR(TIMER0_OVF_vect)
23  {
24    static int col = 4;
25    static int row = 0;
26
27    if (col==4) {
28      col = 0;
29      row = show_char;
30    } else {
31      col++;
32    }
33    PORTA = pgm_read_byte(ptr+5*row+col) & 0xFE;
34    PORTD = ~_BV(col) & 0x1F;
35  }
```

```
37  int main(void)
38  {
39    int i;
40
41    DDRA=0xFE;
42    DDRD=0x1F;
43    PORTD=0x1F;
44    TCCR0 = _BV(CS01)|_BV(CS00);
45    TIMSK = _BV(TOIE0);
46
47    sei();
48
49    while(1) {
50      for (i=0; i<=9; i++) {
51        show_char = i;
52        _delay_ms(1000);
53      }
54    }
55  }
```

**Uitleg code 11.6 regel 4**
`pgmspace.h`

Het includebestand `pgmspace.h` bevat een aantal typedefinities en functies voor het schrijven van gegevens naar of lezen uit het programmageheugen. De compiler slaat variabelen met het attribute `__progmem__` op in het programmageheugen. Het bestand `pgmspace.h` bevat een macro die de schrijfwijze verkort:

```
#define PROGMEM   __attribute__((__progmem__))
```

Bovendien bevat dit headerbestand een aantal typedefinities als:

```
typedef int8_t PROGMEM prog_int8_t;
```

Deze declaraties zetten `x`, `y` en `z` alledrie in het programmageheugen::

```
int x __attribute__((__progmem__));
int y PROGMEM;
prog_int8_t z;
```

**Uitleg code 11.6 regel 19**
`prog_uchar`

In `pgmspace.h` is het type `prog_uchar` gedefineerd als:

```
typedef unsigned char PROGMEM prog_uchar;
```

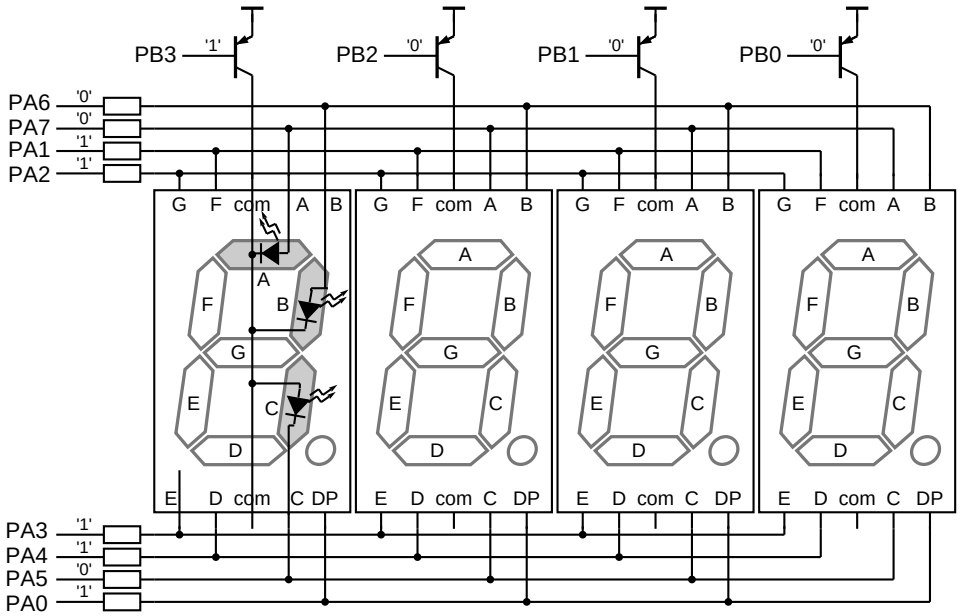
In de code is de variabele `ptr` een pointer naar een **`unsigned char`** in het programmageheugen.

**Uitleg code 11.6 regel 4**
`pgm_read_byte()`

De functie `pgm_read_byte()` leest een byte uit het programmageheugen. De ingangsvariabele van de functie is het adres van de locatie waar de byte gelezen moet worden.
Naast deze leesfunctie kent `pgmspace.h` een aantal lees- en schrijffuncties voor andere dataformaten, zoals voor een word en een dword.

## 11.10 Een 4-digit 7-segmentdisplay aansturen

Een 7-segmentdisplay bestaat uit zeven leds die samen een patroon vormen waarmee cijfers en sommige karakters weergegeven kunnen worden. De segmenten (leds) hebben een rechthoekige vorm en worden aangeduid met de letters `A` tot en met `G`. Meestal heeft het display een achtste led voor een decimale punt (`DP`). De leds van de displays hebben een gemeenschappelijke anode of een gemeenschappelijke kathode.

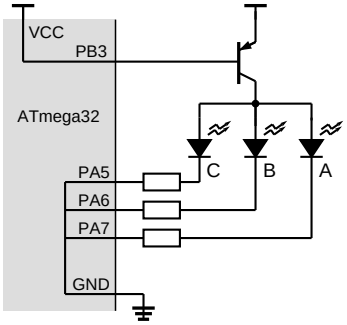



**Figuur 11.18 :** **Een 4-digit 7-segmentdisplay.** De vier displays worden gemultiplext. De kathodes van de leds zijn aangesloten aan `PORTA` van de microcontroller. Vier transistoren sturen de leds aan en de basissen zijn aangesloten op `PORTB` van de microcontroller.

De volgorde van de aansluitingen bij poort A lijkt vreemd. Bij deze oplossing is er voor gekozen om de software overzichtelijk te houden. De leds `A` tot en met `G` zijn aangesloten aan de pinnen `PA7` tot en met `PA0`. In de praktijk kan de PCB-layout aanzienlijk vereenvoudigd worden door een andere volgorde te kiezen.

In figuur 11.18 staan vier 7-segmentdisplays. Deze displays hebben een gemeenschappelijke anode. De kathodes — de aansluitingen `A` tot en met `G` en de decimale punt `DP` — van de vier leds zijn aangesloten aan `PORTA` van de microcontroller.
De anode (`COM`) van ieder display wordt aangestuurd door een transistor. De basis van de transistoren is verbonden met de pinnen `PB0`, `PB1`, `PB2`, en `PB3` van poort `PORTB`.

Als pin `PB3` hoog is en de aansluitingen van de segmenten `A`, `B` en `C` zijn laag dan is op het linker display een zeven zichtbaar. Figuur 11.19 toont het elektrisch gedrag voor deze situatie.
De leds zijn parallel geschakeld. De stroom wordt geleverd door de transistor en hangt af van het aantal leds dat brandt. De stroom door de leds wordt beperkt




**Figuur 11.19 : Het elektrisch gedrag van het 4-digit 7-segmentdisplay voor de situatie dat `PB3` hoog en `PA5`, `PA6` en `PA7` laag zijn.**

Een alternatief bij het aansturen van een meercijferig 7-segmentdisplay is om een seriële 7-segment driver te gebruiken, bijvoorbeeld de M5450 of de MAX7219. Leddrivers zijn beschikbaar in vele soorten en maten.

door de weerstanden. Elke led heeft een eigen afvoer (*sink*) via de microcontroller. De totale stroom die `PORTA` mag afvoeren is 100 mA. Iedere led mag zodoende maximaal 12,5 mA afvoeren.

De displays worden een voor een actief. Als het getal ververst wordt met een frequentie van 50 Hz, wordt het hele getal om de 20 ms ververst. Er zijn vier displays. De tijd $t_{\text{active}}$ dat een display actief is dus 5 ms.

Het algoritme om een getal op het display te zitten is:

```
doe voor ieder 7-segmentdisplay:
  bepaal het cijfer dat getoond moet worden
  zet de waarde voor het betreffende cijfer op poort A
  maak het betreffende display actief
  wacht een tijd t_active
```

Het cijfer dat getoond moet worden kan worden gevonden met de modulus-operator (`%`): `6807 % 10` geeft `7`. Als het getal door tien gedeeld wordt, blijft er $6807/10 = 680$ over. Door dit herhaald toe te passen, worden alle cijfers uit het getal geëxtraheerd:

```
tmp = value;
doe voor ieder 7-segmentdisplay:
  digit = tmp % 10;
  tmp   = tmp / 10;
  zet de waarde voor het betreffende digit op poort A
  maak het betreffende display actief
  wacht een tijd t_active
```

Dit algoritme begint bij het minst significante decimaal. Het is daarom handig om het bijbehorende display aan te sluiten aan pin `0` van poort B. De herhalingslus uit het algoritme is een **`for`**-lus, die loopt van `0` tot en met `3`.
In code 11.7 is dit algoritme geïmplementeerd. Er is — net als bij de codes voor de dotmatrix – een opzoektabel met de af te beelden waarden.

Net als in code 11.4 is er een variabele `t` die het aantal keer telt dat een display ververst is. Eens per seconde (`t == 200`) wordt de af te beelden waarde veranderd. De nieuwe waarde is een willekeurig getal dat de functie `rand()` uit `stdlib.h` bepaalt. Het getal dat `rand()` geeft, ligt tussen `0` en `RAND_MAX`. De ATmega32 gebruikt 16-bits integers en daarom ligt deze waarde tussen `0` en 32767. Modulus `10000` zorgt er voor dat de waarde altijd tussen `0` en `9999` ligt.

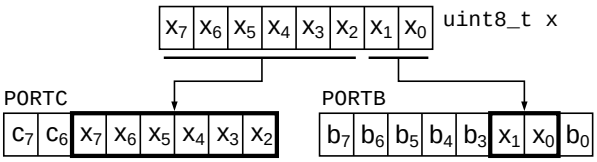
**Code 11.7 : Willekeurige getallen tonen op een 4-digit 7-segmentdisplay.**

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>

const unsigned char lookup[] = {
  0x03, //  0000 0011  = 0
  0x9F, //  1001 1111  = 1
  0x25, //  0010 0101  = 2
  0x0D, //  0000 1101  = 3
  0x99, //  1001 1001  = 4
  0x49, //  0100 1001  = 5
  0x41, //  0100 0001  = 6
  0x1F, //  0001 1111  = 7
  0x01, //  0000 0001  = 8
  0x09  //  0000 1001  = 9
};

int main(void)
{
  int i, t=200;
  int value;
  int tmp, digit;

  DDRA  = 0xFF;
  DDRB  = 0x0F;
  while(1) {
    if (t == 200) {
      value = rand() % 10000;
      t = 0;
    }
    tmp = value;
    for (i=0; i<4; i++) {
      digit = tmp % 10;
      tmp = tmp / 10;
      PORTA = lookup[digit];
      PORTB = ~_BV(i) & 0x0F;
      _delay_ms(5);
    }
    t++;
  }
}
```

## 11.11 In- en uitlezen van informatie vanaf verschillende poorten

In veel situaties is het handig de aansluitingen te verdelen over meerdere poorten of de bits op een niet voor de hand liggende manier aan te sluiten. De layout van de PCB, die hoort bij figuur 11.18, zal veel eenvoudiger zijn als de anodes in een andere volgorde aan poort A worden aangesloten. Ook kan het zijn dat een deel van de poort voor een van de speciale functies nodig is. Bij een ontwerp met een UART en interrupt 0, zijn `PD0`, `PD1` en `PD2` van poort D bijvoorbeeld al in gebruik voor de `RXD`, `TXD` en `INT0`. De overige aansluitingen zijn dan nog beschikbaar. Voor de toekenning aan de `PORT`- en `PIN`-registers zijn dan bitbewerkingen nodig. Deze paragraaf laat met een paar voorbeelden zien hoe dat gaat.

Stel dat van een `uint8_t` integer `x` de twee minst significante bits aan bit 2 en 1 van poort B en de andere zes bits aan de minste significante bits van poort C moeten worden toegekend op de manier zoals figuur 11.20 dit laat zien.



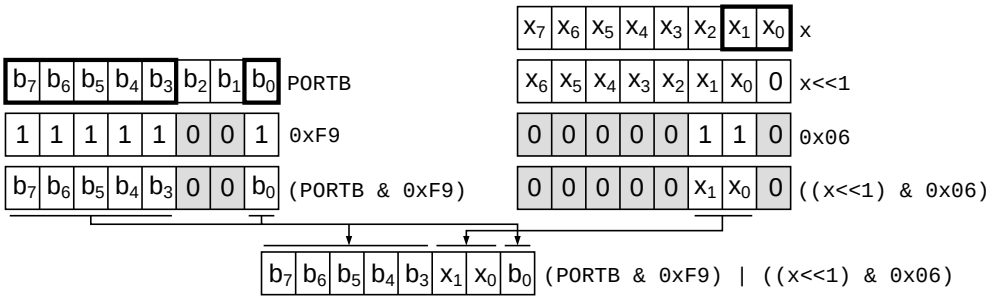

**Figuur 11.20 : Variabele `x` wordt toegekend aan een deel van `PORTB` en een deel van `PORTC`.**

Een deel van de bits van de registers `PORTB` en `PORTC` blijven ongewijzigd en een deel veranderd. De toekenning aan deze registers bestaan uit twee delen: een deel

voor de bits die veranderd moeten worden en een deel voor de bits die hetzelfde blijven. Een masker (&) selecteert de bits en de bitsgewijze OF (|) brengt de delen bij elkaar:

```
PORTB = (PORTB & 0xF9) | ((x << 1) & 0x06);
PORTC = (PORTC & 0xC0) | ((x >> 2) & 0x3F);
```

Figuur 11.21 visualiseert de toekenning aan `PORTB`. Met het masker `0xF9` worden bit 2 en 1 van poort B gemaskerd. De term `(PORTB & 0xF9)` bevat de bits, die niet veranderd worden.

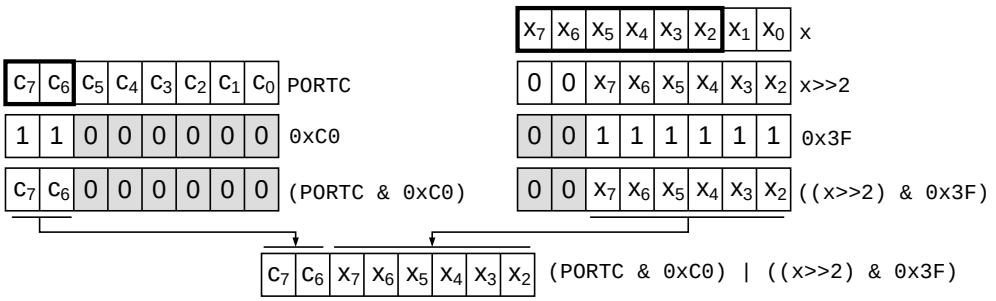



**Figuur 11.21 : De visualisatie van toekenning aan `PORTB`.** De toekenning bestaat uit een deel dat een aantal bits van register `PORTB` bewaart en een deel dat de andere bits wijzigt.

Het masker `0xF9` is het complement van `0x06` en kan met bitsgewijze inversie ook als `~0x06` geschreven worden.

Het masker `0x06` kan ook met `(_BV(2)|_BV(1))` worden beschreven en `0xF9` met `(~(_BV(2)|_BV(1)))`. De bitvalue `_BV()` wordt in paragraaf 12.4 besproken.

De uitdrukking `(x << 1)` schuift de bits van de variabele `x` een positie naar links. Het masker `0x06` maskeert alle bits van `x` behalve bit 2 en 1. De term `((x << 1) & 0x06)` bevat de bits die veranderen. Bit 2 en bit 1 van deze uitdrukking bevatten de waarden van bit 1 en bit 0 van variabele `x`.
De bitsgewijze OF combineert deze twee termen. Het resultaat van de hele toekenning is dat alleen bit 2 en 1 van poort B veranderd zijn.
Figuur 11.22 visualiseert de toekenning aan `PORTC`. Met het masker `0xC0` worden bit 7 en 6 van poort C gemaskerd. De term `(PORTC & 0xC0)` bevat de bits, die niet veranderd worden.

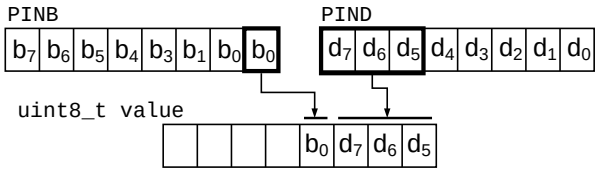



**Figuur 11.22 : De visualisatie van toekenning aan `PORTC`.** De toekenning bestaat uit een deel dat een aantal bits van register `PORTC` bewaart en een deel dat de andere bits wijzigt.

Omdat er bij het schuiven automatische nullen worden ingevuld op de vrijgekomen plaatsen, Is het masker (`0x3F`) in dit geval niet nodig.

De uitdrukking `(x >> 2)` schuift de bits van `x` twee posities naar rechts. Het masker `0x3F` maskeert bit 7 en 6 van poort C. De term `((x >> 2) & 0x3F)` bevat de bits, die veranderen. De zes minst significante bits van deze uitdrukking bevatten de zes meest significante bits van variabele `x`.
De bitsgewijze OF combineert deze twee termen met als resultaat dat alleen de zes minst significante bits van poort C gewijzigd worden.

Ook bij het inlezen van meerdere bits zijn de aansluitingen soms verdeeld zijn over meerdere poorten. In figuur 11.23 wordt een bit van pin B en drie bits van pin D toegekend aan de minst significante bits van een `uint8_t` integer `value`.

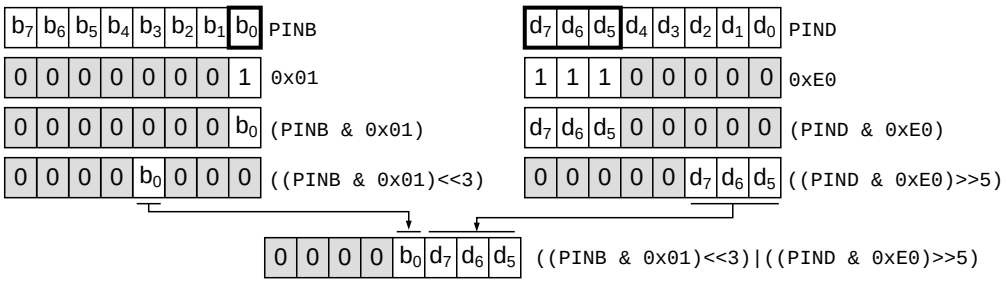



**Figuur 11.23 : Bit `0` van `PORTB` en drie bits van `PORTD` worden aan de variabele `value` toegekend.**

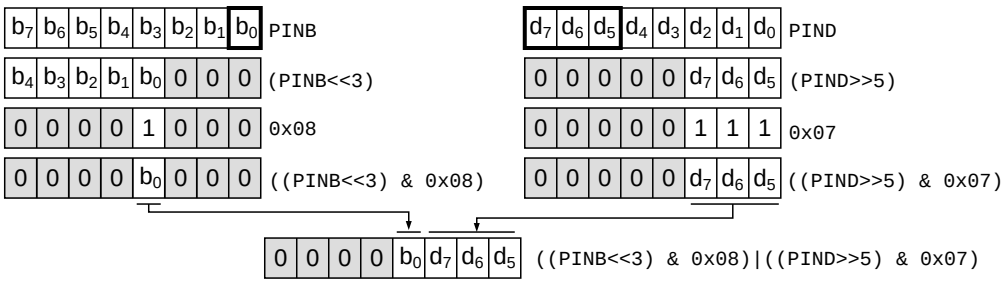
De toewijzing bestaat weer uit twee delen: een deel leest de bit van poort B en een deel leest dat de drie bits van poort D. In de onderstaande toekenning worden de toe te kennen bits eerst gemaskeerd en vervolgens naar de juiste positie geschoven:

```
value = ((PINB & 0x01) << 3) | ((PIND & 0xE0) >> 5);
```

Figuur 11.24 illustreert de stappen in deze toekenning. De overige vier bits van de variabele `value` worden nul gemaakt. Als dat niet moet, kan er een derde term (`value&0xF0`) toegevoegd worden die er voor zorgt dat de vier meest significante bits van `value` bewaard blijven.




**Figuur 11.24 : De toekenning aan `value` door eerst de bits te maskeren en daarna te schuiven.**




**Figuur 11.25 : De toekenning aan `value` door eerst de bits te schuiven en daarna te maskeren.**

Bij de toekenning kunnen de bits ook eerst geschoven worden en daarna gemaskeerd:

```
value = ((PINB<<3) & 0x08) | ((PIND>>5) & 0x07);
```

Figuur 11.25 laat deze bewerking stap voor stap zien. Bij het schuiven en maskeren worden tijdelijke geheugenplaatsen gebruikt de oorspronkelijke waarden in poort B en poort D blijven behouden.

# 12

# Interrupts

| | |
|---|---|
| Doelstelling | In dit hoofdstuk leer je wat een interrupt is, hoe het interruptmechanisme werkt en hoe je de externe interrupt van de ATmega32 gebruikt. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ Het verschil tussen polling en interrupts.<br>▪ Het interruptmechanisme bij de ATmega32.<br>▪ Het gebruik van de externe interne interrupt `0`, de Interrupt Service Routine `ISR` en de macro's `sei()` en `cli()`.<br>▪ Bitmaskering: set bit, clear bit, toggle bit, test bit.<br>▪ De bitvalue macro `_BV()`.<br>▪ De *read-modify-write instruction*.<br>▪ Contactdender.<br>▪ Antidenderschakelingen.<br>▪ Antidenderalgoritmen.<br>De voorbeelden demonstreren het gebruik van:<br>▪ Externe interrupt `0`.<br>▪ Externe interrupt `0` met de bitvalue-notatie `_BV()`.<br>▪ Het uitlezen van een drukknop met behulp van polling.<br>▪ Het uitlezen van acht drukknoppen met behulp van polling.<br>▪ Het uitlezen van acht drukknoppen met externe interrupt 2. |

Polling is vergelijkbaar met deze situatie. Stel dat je op je studeerkamer bezig bent, dat je vrienden verwacht en dat de bel stuk is. Je moet dan op gezette tijden naar de voordeur gaan om te kijken of er al iemand voor de deur staat. Meestal zal je voor niets naar de voordeur lopen. Van het eigenlijke werk komt niet veel terecht.

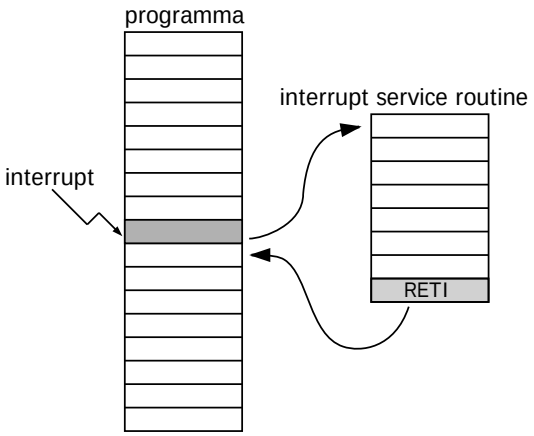
Er zijn twee manieren om te reageren op externe signalen: via *polling* en via interrupts. Bij *polling* ligt het initiatief bij de microcontroller. Op gezette tijden controleert het hoofdprogramma van de microcontroller of de ingangen gewijzigd zijn. Bij een interrupt ligt het initiatief niet bij het hoofdprogramma. Een interne of externe gebeurtenis zorgt voor de interrupt. Op dat moment stopt het hoofdprogramma met waar het mee bezig is. De microcontroller voert eerst de speciale interruptroutine uit en gaat daarna weer verder met het hoofdprogramma.

Het voordeel van het gebruik van een interrupt is evident. Toch wordt *polling* ook veel gebruikt. Bijvoorbeeld omdat de microcontroller te weinig mogelijkheden heeft. De ATmega32 heeft bijvoorbeeld maar drie externe interrupts. Vooral bij het lezen van veel ingangen is *polling* een goed alternatief.

## 12.1 Het interruptmechanisme

Een interrupt is vergelijkbaar met de situatie dat de bel het wel doet. Je doet je gewone werk, bijvoorbeeld: het schrijven van een verslag voor het vak microcontrollers. Pas als er gebeld wordt, ga je naar de voordeur. Daarna ga je terug en ga je verder met wat je aan het doen was.

Een Interrupt Service Routine (ISR) is een routine, die op elk moment door een interne of externe gebeurtenis gestart kan worden. Een externe gebeurtenis is bijvoorbeeld een signaalverandering op een van de interruptpinnen. Een voorbeeld van een interne gebeurtenis is de overflow van een timer.
Een interruptroutine is nuttig omdat het hoofdprogramma door kan gaan met wat het moet doen. Zonder interrupt moet het hoofdprogramma voortdurend checken wat de status van de ingangen en interne registers is. Een interrupt onderbreekt het normale programma op het moment dat er ook echt een gebeurtenis is. De ISR wordt uitgevoerd en het hoofdprogramma gaat daarna weer verder waar het onderbroken was.
Er zijn twee problemen die bij interrupts goed geregeld moeten worden: de ISR moet vanaf elk plaats in het hoofdprogramma gevonden kunnen worden en het programma moet weer verder kunnen gaan op de plaats waar het gebleven was. De ISR wordt gevonden met zogenoemde interruptvectoren. Dat zijn sprongopdrachten, die aan het begin van de programmacode staan, naar de verschillende ISR's. De positie, waar het hoofdprogramma verder moet gaan, wordt onthouden door dit adres op de stack te zetten.




**Figuur 12.1** : **De Interrupt Service Routine.**

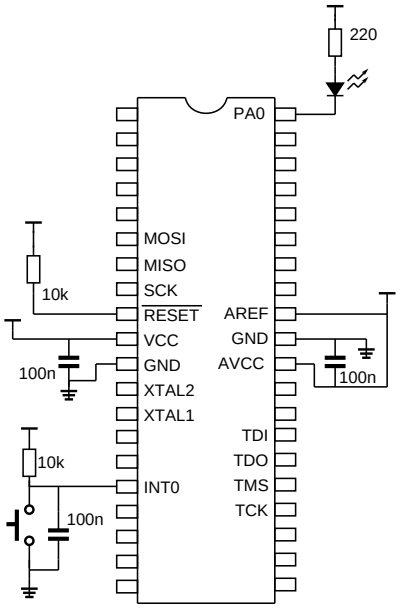
Bij een interrupt zijn dit de handelingen, die achtereenvolgens worden uitgevoerd:

- De huidige instructie wordt afgemaakt.
- Het adres van de volgende instructie wordt op de stack gezet.
- Het adres van de betreffende interruptvector wordt in de programcounter geladen. Op het adres van de interruptvector staat een sprongopdracht naar de ISR.
- Het adres van de ISR wordt in de programcounter geladen.
- De ISR wordt uitgevoerd.
- Bij de RETI instructie (*RETurn from Interrupt*) is de ISR voltooid.
- Het oude adres gaat van de stack naar de programcounter.
- Het hoofdprogramma wordt weer verder uitgevoerd.

Naast het adres worden er vaak nog meer gegevens op de stack gezet. De situatie nadat de interrupt uitgevoerd is, moet in ieder geval zo zijn dat het hoofdprogramma verder kan gaan waar het gebleven was.

## 12.2 De schakeling voor de demonstratie van externe interrupt 0

De ATmega32 kent drie externe interrupts `INT0`, `INT1` en `INT2`. Deze paragraaf behandelt interrupt `INT0`, die verbonden is met pin `PD2`. Een signaalverandering op deze pin, zal de status van de led wijzigen. De led zal uit gaan als deze aan is en gaan branden als deze uit was.
Figuur 12.2 geeft de minimale configuratie om het interrumperen te testen. Pin `PD2` is verbonden met een pullupweerstand van 10 kΩ aan `VCC` en met een scha-




**Figuur 12.2 : Het schema voor het demonstreren van interrupt `INT0`.**

kelaar of drukknop aan `GND`. Ingang `INT0` wordt laag wanneer de drukknop ingedrukt wordt. Als het contact verbroken wordt, wordt de ingang weer hoog. De condensator van 100 nF voorkomt contactdender.
De rest van de schakeling is gelijk aan die van figuur 11.1. De led is met een weerstand van 220 Ω aangesloten op pin `PA0`. De $\overline{\texttt{RESET}}$ zit met een 10 kΩ weerstand aan VCC. Er wordt gebruik gemaakt van de interne oscillator. Er zijn twee capaciteiten van 100 nF tussen VCC en GND en AVCC en GND geplaatst.

## 12.3 De software voor de externe interrupt 0

Het statusregister is een belangrijk register. De ALU zet hierin na elke berekening een aantal bits (*flags*). Een paar voorbeelden zijn de *carry flag* `C`, de *zero flag* `0` en de *negative flag* `N`.

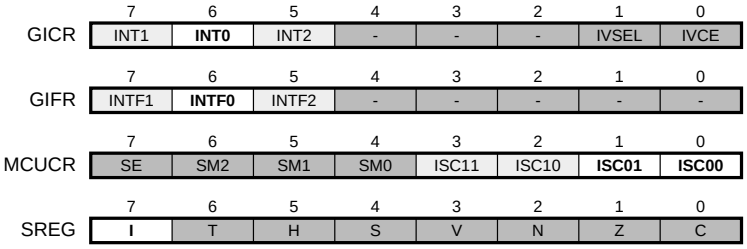
Bij elk soort interrupt is altijd een aantal bits uit verschillende registers nodig. Meestal gaat het om vier soorten bits die in vier verschillende registers staan:

- Met het globale interrupt bit `I` uit het statusregister `SREG` kan het hele interruptmechanisme worden aan- en uitgezet. Als dit bit laag is, is de microcontroller ongevoelig voor alle soorten interrupts. Is het bit hoog dan is het interruptmechanisme actief. De AVR GNU-compiler gebruikt de functie `cli()` om dit bit laag te maken (*CLear Interrupt bit*) en de functie `sei()` om dit bit hoog te maken (*SEt Interrupt bit*).
- De ATmega32 heeft twintig verschillende interrupts. Deze interrupts kunnen allemaal afzonderlijk aan- of uitgezet worden. Voor interrupt `0` is dat bit `INT0` uit het General Interrupt Control Register.
- De verschillende interrupts hebben ook allemaal een bit waarmee de microcontroller aangeeft dat de interrupt heeft plaats gevonden. De microcontroller gebruikt bij interrupt `0` daarvoor bit `INTF0` uit het General Interrupt Flag Register. Als deze bit hoog is, wordt er automatisch naar de betreffende ISR gesprongen. De programmeur hoeft hier niets aan te doen. Bij het debuggen kan hij door dit bit hoog te maken een interrupt forceren.

**Tabel 12.1 : De Interrupt Sense Control bits voor interrupt 0.**

| ISC01 | ISC00 | Gevoeligheid van het ingangssignaal |
|---|---|---|
| 0 | 0 | een laag niveau geeft een interrupt |
| 0 | 1 | elke verandering geeft een interrupt |
| 1 | 0 | de neergaande flank geeft een interrupt |
| 1 | 1 | de opgaande flank geeft een interrupt |

- Daarnaast kent elke interrupt een aantal instelmogelijkheden. Voor Interrupt 0 zijn dat de bits ISC01 en ISC00 uit het MCU Control Register. Deze bits leggen de gevoeligheid voor het ingangssignaal vast, zie tabel 12.1.




**Figuur 12.3 : De bits en de registers, die nodig zijn voor interrupt 0.**
Deze bits zijn vet gedrukt en hebben een witte achtergrond. Bits, die bij andere interrupts gebruikt worden hebben een lichtgrijze en de overige bits hebben een donkergrijze achtergrond.

Figuur 12.3 geeft de bits en de registers die nodig zijn bij het toepassen van interrupt 0. Om een externe interrupt 0 mogelijk te maken, moet INT0 van GICR hoog zijn en moet de interruptvlag I van het statusregister SREG gezet zijn. Eventueel kunnen de ISC01 en ISC00 bits uit MCUCR worden aangepast voor de juiste gevoeligheid van het ingangssignaal.

**Code 12.1 : Led aan en uit zetten met interrupt 0.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  ISR(INT0_vect)
5  {
6    PORTA ^= 0x01;   // toggle output
7  }
8
9  int main(void) {
10   DDRA  = 0x01;    // bit 0 PORT A is output
11
12   GICR  = 0x40;    // enable interrupt 0
13   MCUCR = 0x02;    // falling edge ISC01=1 ISC00=0
14
15   sei();           // enable global interrupt
16
17   while(1);        // do nothing
18 }
```

Code 12.1 reageert op de externe interrupt 0. Elke keer dat de knop ingedrukt wordt, wordt het interruptsignaal laag en wordt de interrupt service routine uit-

gevoerd. Om — in het algemeen — een interrupt toe te passen, moeten er vier zaken bekend of ingesteld zijn:

1. De interrupt flag uit het statusregister moet hoog zijn om het hele interruptmechanisme van de microcontroller aan te zetten. Dit wordt gedaan met de functie `sei()`.
2. De interrupts waarvoor de microcontroller gevoelig zijn.
3. De Interrupts Service Routines moeten gedefinieerd zijn.
4. De bijbehorende interruptvectoren moeten gedefinieerd zijn. Een interruptvector is een vaste plaats in het geheugen en bevat een sprongopdracht naar het adres van de betreffende Interrupt Service Routine.

Figuur 12.4 toont de plaatsen waar deze vier punten in de code 12.1 staan.

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

ISR(INT0_vect)
{
  PORTA ^= 0x01;
}

int main(void) {
  DDRA  = 0x01;
  GICR  = 0x40;
  MCUCR = 0x02;
  sei();

  while(1);
}
```

4 Interruptvector voor externe interrupt 0

3 Interrupt Service Routine (ISR)

2 Zet externe interrupt 0 (INT0) aan

1 Zet hele interruptmechanisme aan

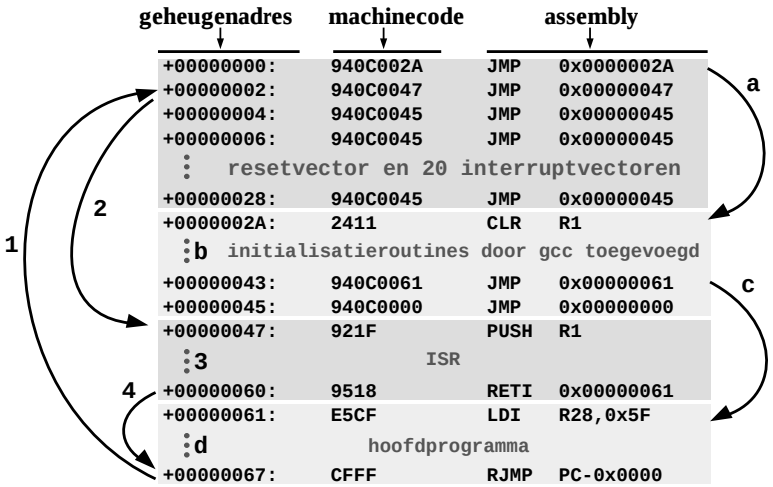
**Figuur 12.4 : De vier aspecten die nodig zijn voor een externe interrupt 0.**

Een interruptvector is een vaste plek in het programmageheugen. Daar staat een sprongopdracht naar het begin van de betreffende ISR. De ISR kan op deze manier altijd gevonden worden.

**Tabel 12.2 : De resetvector en de twintig interruptvectoren van de ATmega32.**

| | adres (in words) | naam bij AVR GCC | omschrijving |
|---|---|---|---|
| | 0x0000 | *resetvector* | *voor power-on-reset, brownout, watchdog, JTAG* |
| 1 | 0x0002 | INT0_vect | External Interrupt Request 0 |
| 2 | 0x0004 | INT1_vect | External Interrupt Request 1 |
| 3 | 0x0006 | INT2_vect | External Interrupt Request 2 |
| 4 | 0x0008 | TIMER2_COMP_vect | Timer/Counter2 Compare Match |
| 5 | 0x000A | TIMER2_OVF_vect | Timer/Counter2 Overflow |
| 6 | 0x000C | TIMER1_CAPT_vect | Timer/Counter1 Capture Event |
| 7 | 0x000E | TIMER1_COMPA_vect | Timer/Counter1 Compare Match A |
| 8 | 0x0010 | TIMER1_COMPB_vect | Timer/Counter1 Compare Match B |
| 9 | 0x0012 | TIMER1_OVF_vect | Timer/Counter1 Overflow |
| 10 | 0x0014 | TIMER0_COMP_vect | Timer/Counter0 Compare Match |
| 11 | 0x0016 | TIMER0_OVF_vect | Timer/Counter0 Overflow |
| 12 | 0x0018 | SPI_STC_vect | Serial Transfer Complete |
| 13 | 0x001A | USART_RXC_vect | USART Rx Complete |
| 14 | 0x001C | USART_UDRE_vect | USART Data Register Empty |
| 15 | 0x001E | USART_TXC_vect | USART Tx Complete |
| 16 | 0x0020 | ADC_vect | ADC Conversion Complete |
| 17 | 0x0022 | EE_RDY_vect | EEPROM Ready |
| 18 | 0x0024 | ANA_COMP_vect | Analog Comparator |
| 19 | 0x0026 | TWI_vect | Two-wire Serial Interface |
| 20 | 0x0028 | SPM_RDY_vect | Store Program Memory Ready |

Elk type microcontroller kent zijn eigen features. Sommige AVR's hebben twee in plaats van drie timers of hebben geen ADC. De ATmega128 kent acht interrupts en ATmega32 kent er slechts drie. Het aantal interruptvectoren is dus voor elke ATmega anders. In tabel 12.2 staan de interruptvectoren van de ATmega32. Deze vectoren zijn in het headerbestand `avr/iom32.h`, dat door `avr/io.h` wordt aangeroepen, gedefinieerd.




**Figuur 12.5 : De assembly, die hoort bij code 12.1 met de acties bij het starten van de microcontroller en bij een aanroep van de ISR.**
Het programma begint bij de resetvector en springt [a] naar een aantal initialisatie routines [b], die `gcc` heeft toegevoegd. Vervolgens springt [c] het naar hoofdprogramma en voert dit uit [d]. Bij een interrupt springt [1] het programma naar de interruptvector en springt [2] daar vandaan naar de ISR en voert [3] deze uit. Als de ISR klaar is, springt [4] het programma terug naar het hoofdprogramma.

In AVRstudio kan de code worden gedisassembleerd door in de debugmode de *disassembler* aan te roepen. Deze laatste vertaalt het `hex`-bestand naar machinecode en assembly. Als het disassembler bestand actief is, wordt bij het debuggen niet door de C-code maar door de machinecode en assembly heen gestapt.

In figuur 12.5 is de C-code 12.1 vertaald naar machinecode en assembly. Op geheugenlocatie `0x0000` staat de resetvector. Dat is een sprongopdracht naar het begin van het programma. De power-on-reset, de brown-out, de watchdogtimer en JTAG maken hiervan gebruik bij het resetten van het programma. Figuur 12.5 laat zien dat dit gaat via een initialisatie, die de compiler heeft toegevoegd.
Na de resetvector komen de twintig interruptvectoren. `INT0_vect` is niet anders dan adres `0x0002`. Op dit adres staat een verwijzing naar de ISR-routine. Als er een externe interrupt `0` is, wordt het adres van de volgende instructie op de stack gezet en springt het programma naar adres `0x0002`. Daar staat een sprongopdracht naar de ISR. Als de ISR klaar is, wordt het adres van de stack gehaald en gaat het hoofdprogramma door waar het gebleven was.

**Uitleg code 12.1 regel 2**
`#include <avr/interrupt.h>`

Het bestand `avr/interrupt.h` bevat de definities van een aantal macro's die bij interrupts worden gebruikt, zoals `sei()` en `ISR()`.

**Regel 4**
```
ISR(vector)
{
  ISR-code
}
```

`ISR()` is de macro, die bij AVR GCC de interrupts afhandelt. Om een *Interrupt Service Routine* goed uit te voeren moeten een aantal gegevens worden bewaard voordat de ISR-code wordt uitgevoerd. Nadat de ISR-code is uitgevoerd, worden deze gegevens weer teruggezet. Het hoofdprogramma gaat dan gewoon verder waar het gebleven was. De macro `ISR()` handelt het interruptmechanisme op de juiste wijze af.

De macro `ISR()` cleart de globale interrupt bit (`I`) in het statusregister `SREG` voordat ISR-code wordt uitgevoerd. De *interrupt service routines* zijn standaard dus niet interrumpeerbaar. In de AVR LIBC User manual staat hoe een interrumpeerbare ISR gemaakt kan worden. Het interrumperen van een interrupt kan heel complex zijn. Dit geeft snel een stack-overflow. Het is daarom sterk af te raden om interrumpeerbare interrupts te gebruiken. Met de gewone `ISR()` zijn de meeste problemen op te lossen.

**Regel 15**
`sei()`
`cli()`

De macro's `sei()` en `cli()` zijn gedefineerd in `avr/interrupt.h`:

```
#define sei()  __asm__ __volatile__ ("sei" ::)
#define cli()  __asm__ __volatile__ ("cli" ::)
```

Deze macro's maken de statusvlag `I` uit het statusregister `SREG` respectievelijk hoog of laag en gebruiken daarvoor de assemblerinstructie `sei` (SEt global Interrupt) en `cli` (CLear global Interrupt).

**Regel 15**
`while(1);`

Na de initialisaties voert het hoofdprogramma een oneindige lus uit met een leeg statement `;`. Het hoofdprogramma doet dus niets.
Een oneindige lus met een leeg statement wordt door de AVR GNU-compiler correct verwerkt. Alleen heeft de simulator van AVRstudio moeite met deze code als er gecompileerd is met de optimalisatie aan. Het lijkt dan dat de simulator vastloopt en dat het programma niet goed is. Dat is niet het geval. In de disassembler kan gewoon door de code gestapt worden.
Dit probleem kan worden voorkomen door een no-operating instructie in de **`while`** te zetten.

```
while (1) {
  asm volatile ("nop");
}
```

De opdracht `asm` geeft aan dat er een assembler instructie (`nop`, no-operating) volgt. Het sleutelwoord **`volatile`**, zorgt ervoor dat de compiler deze opdracht niet wegeoptimaliseerd.

## 12.4 De software voor interrupt 0 met bitnotatie

Code 12.2 is identiek aan code 12.1, alleen wordt nu de bitnotatie gebruikt. Het voordeel van deze schrijfwijze is dat de toewijzingen betekenisvoller zijn. Een nadeel is dat de code meer specifiek voor de AVR GNU-compiler is geschreven. Met de bitbewerkingen uit paragraaf 9 en de macro `_BV()` zijn alle bittoewijzingen en bittests te maken.

**Uitleg code 12.2 regel 10**
`_BV()`
*`bit value`*

Het headerbestand `sfrdefs.h` dat automatisch wordt ingesloten met `io.h` bevat de definitie van `_BV()`:

```
#define _BV(bit) (1 << (bit))
```

De macro `_BV()` (*bit value*) wordt gebruikt om een bit te maskeren. `_BV(3)` betekent dus `1<<3` en dat is gelijk aan `0x08`. De compiler doet deze vertaling. In de C-code staat `_BV(3)`. Dit is goed leesbaar. Het betekent *bit value* 3; dus bitnummer 3. In de machinecode staat gewoon `0x08`. Hieronder staan vijf manieren om bit 3 van register `DDRA` hoog te maken. De andere bits worden laag gemaakt.

```
DDRA = 0x08;
DDRA = 0b00001000;
DDRA = 1 << 3;
DDRA = _BV(3);
DDRA = _BV(PA3);
```

**Code 12.2 : Led aan en uit zetten met interrupt 0 met bitnotatie.**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3
 4  ISR(INT0_vect)
 5  {
 6    PORTA ^= _BV(0);
 7  }
 8
 9  int main(void) {
10    DDRA  = _BV(0);
11
12    GICR  = _BV(INT0);
13    MCUCR = _BV(ISC01);  // falling edge ISC01=1 ISC00=0
14
15    sei();
16
17    while (1) {
18      asm volatile ("nop");
19    }
20  }
```

Met `_BV()` kunnen ook meerdere bits worden gemaskeerd. De volgende toewijzing maakt de oneven bits van register `PORTA` hoog.

```
PORTA = _BV(7)|_BV(5)|_BV(3)|_BV(1);
```

**Regel 6**
`_BV()`
*bit manipulations*

Met de bitoperatoren `&`, `|` en `^` kunnen de afzonderlijke bits uit een register gemanipuleerd worden. In tabel 12.3 staan de verschillende bitbewerkingen om bits te zetten, te clearen, te toggelen en te testen. Paragraaf 12.5 bespreekt deze bewerkingen en toont een aantal voorbeelden.

**Tabel 12.3 : De bitbewerkingen met _BV().**

| Naam | Bitwerking | Toelichting |
|---|---|---|
| *bit set* | `r \|= _BV(bit);` | Deze toewijzing maakt alleen bitnummer `bit` van register `r` hoog. |
| *bit clear* | `r &= ~(_BV(bit));` | Deze toewijzing maakt alleen bitnummer `bit` van register `r` laag. |
| *bit toggle* | `r ^= _BV(bit);` | Deze toewijzing toggelt alleen bitnummer `bit` uit register `r`. |
| *bit test* | `(r & _BV(bit))` | Deze uitdrukking is waar als bitnummer `bit` uit register `r` hoog is. |
| | `!(r & _BV(bit))` | Deze uitdrukking is waar als bitnummer `bit` uit register `r` laag is. |

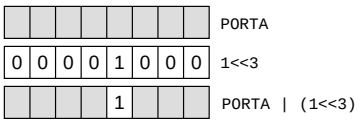
## 12.5 Bitbewerkingen voor set, clear, toggle en test

De huidige AVR GNU-compiler kent geen functies of macro's meer voor het setten en clearen van bits. Oudere versies gebruikten hiervoor de macro's `sbi` en `cbi`. Aangeraden wordt deze macro's niet meer te gebruiken. In bijlage H.2 staat hierover meer informatie.
De reden dat deze macro's niet meer gebruikt worden, is dat dit ook heel efficiënt met bitbewerkingen gedaan kan worden. Verwarrend is dat dit op meerdere manieren gedaan kan worden. Hierna volgen een aantal voorbeelden met bitbewerkingen. Er zijn steeds drie notaties mogelijk: met een hexadecimale waarde, met

een schuifoperator (<<) en met `_BV()`. Deze notaties zijn equivalent. De notaties met `_BV()` zijn het best leesbaar.

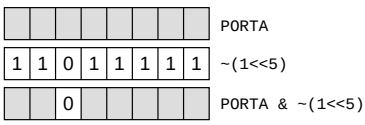
Met de bitsgewijze OF (|) wordt een specifiek bit hoog gemaakt. In dit voorbeeld is dat bit 3. De andere zeven bits veranderen niet:

```
//  Set bit 3 and leave other bits unchanged
  PORTA |= 0x08;
  PORTA |= (1 << 3);
  PORTA |= _BV(3);
```



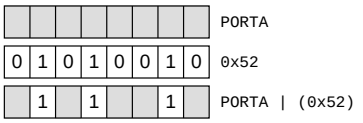

Met de bitsgewijze EN (&) en de bitinversie (~) wordt een specifiek bit laag gemaakt. In dit geval is dat bit 5:

```
//  Clear bit 5 and leave other bits unchanged
  PORTA &= ~0x20;
  PORTA &= ~(1 << 5);
  PORTA &= ~_BV(5);
```



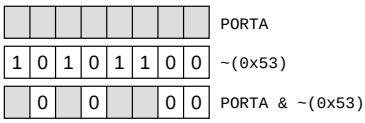

Met de bitsgewijze OF (|) kunnen ook meerdere bits hoog gemaakt worden. In dit voorbeeld is dat de bit 6, 4 en 1:

```
//  Set bit 6, 4 and 1 and leave other bits unchanged
  PORTA |= 0x52;
  PORTA |= (1 << 6) | (1 << 4) |(1 << 1);
  PORTA |= _BV(6)|_BV(4)|_BV(1);
```



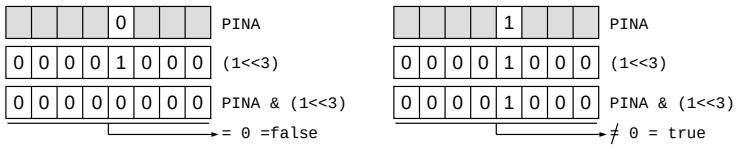

Met de bitsgewijze EN (&) en de bitinversie (~) kunnen ook meerdere bits laag gemaakt worden. In dit geval zijn dat de bits 6, 4, 1 en 0:

```
//  Clear bit 6, 4, 1 and 0 and leave other bits unchanged
  PORTA &= ~0x53;
  PORTA &= ~((1 << 6) | (1 << 4) | (1 << 1) | (1 << 0));
  PORTA &= ~(_BV(6)|_BV(4)|_BV(1)|_BV(0));
```



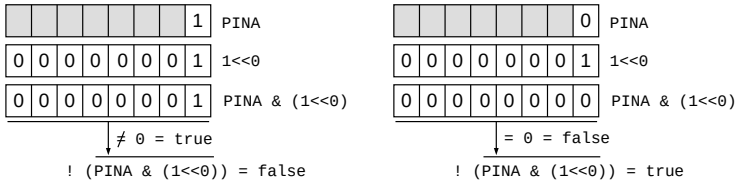

Met de bitsgewijze EN (&) kan getest worden of er een bepaald bit hoog is. Als bit 3 van register `PINA` hoog is, zijn deze uitdrukkingen *waar* en anders zijn deze *niet waar*:

```
// test if bit 3 is set
  if ( PINA & 0x08 ) { }
  if ( PINA & (1 << 3) ) { }
  if ( PINA & (_BV(3)) ) { }
```




Met de bitsgewijze EN (&) en de logische negatie (!) wordt getest of er een bepaald bit laag is. Als bit 0 van register `PINA` laag is, zijn deze uitdrukkingen *waar* en anders zijn deze *niet waar*:

```
// test if bit 0 is clear
  if ( ! (PINA & 0x01) ) { }
  if ( ! (PINA & (1 << 3)) ) { }
  if ( ! (PINA & (_BV(3))) ) { }
```




Voor het testen van bits zijn in `sfr_des.h` twee macro's `bit_is_set` en `bit_is_clear` gedefinieerd, zie tabel 12.4. Het is verstandig deze te gebruiken omdat de bitoperatoren bij de AVR GNU-compiler automatisch een **`int`** impliceren en de registers maar uit 8-bits bestaan.

**Tabel 12.4 : Macro's voor het testen van bits[1].**

| Naam | Bitwerking | Toelichting |
|---|---|---|
| *bit test* | `bit_is_set(r,bit)` | Deze uitdrukking is waar als bitnummer `bit` uit register `r` hoog is. |
| | `bit_is_clear(r,bit)` | Deze uitdrukking is waar als bitnummer `bit` uit register `r` laag is. |
| *loop until* | `loop_until_bit_is_set(r, bit)` | Deze macro wacht totdat `bit` uit register `r` hoog is. |
| | `loop_until_bit_is_clear(r, bit)` | Deze macro wacht totdat `bit` uit register `r` laag is. |

[1] Deze macro's zijn gedefinieerd in `sfr_defs.h`. Dit headerbestand wordt automatisch ingesloten als `io.h` gebruikt wordt.

Om meerdere bits uit een register tegelijkertijd te testen, zijn er verschillende mogelijkheden. Hieronder staan vijf methoden die testen of de bits 7 en 0 van `PINA` hoog zijn:

```
// test if bit 7 and 0 are set
  if ( (PINA & 0x81) == 0x81 ) { }
  if ( (PINA & ((1 << 7)|(1 << 0))) == ((1 << 7)|(1 << 0)) ) { }
  if ( (PINA & (_BV(7)|_BV(0))) == (_BV(7)|_BV(0)) ) { }
  if ( (PINA & (_BV(7))) && (PINA & (_BV(7))) ) { }
  if ( (bit_is_set(PINA,7)) && (bit_is_set(PINA,0)) ) { }
```

De opdrachten met `&=`, `|=` en `^=` zijn verkorte schrijfwijzes. De toewijzing `PORTA |= 0x0F;` is een heel andere toewijzing dan `PORTA = 0x0F`. De laatste kent alleen een constante toe aan het register `PORTA`. Het is een enkele schrijfopdracht. Mits er geoptimaliseerd is met `-Os` is er slechts een klokslag nodig. De toewijzing `PORTA |= 0x0F;` betekent `PORTA = PORTA | 0x0F;` en bestaat uit drie handelingen: het lezen van de waarde van `PORTA`, het veranderen van deze waarde (`PORTA | 0x0F`) en het terugschrijven van het resultaat naar register `PORTA`. Mits er geoptimaliseerd wordt, zijn daar drie klokslagen voor nodig. Deze toewijzingen worden *read-modify-write instructions* genoemd.

Zonder optimalisatie zijn er respectievelijk vijf en negen klokslagen.

Hieronder staan twee initialisatiemethoden voor het `MCUCR`-register met twee interrupts met een opgaande klokflank.

```
MCUCR |= _BV(ISC11);
MCUCR |= _BV(ISC10);
MCUCR |= _BV(ISC01);
MCUCR |= _BV(ISC00);
```

```
MCUCR = _BV(ISC11)|
        _BV(ISC10)|
        _BV(ISC01)|
        _BV(ISC00);
```

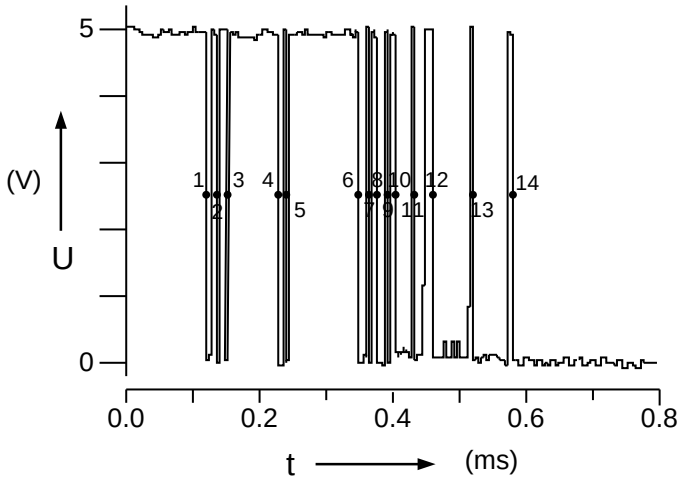
Met optimalisatie is bij de toewijzing één klokslag nodig en bij de *read-modify-write* zijn er acht klokslagen nodig. Zonder optimalisatie zijn er respectievelijk vijf en 36 klokslagen nodig.

De linker initialisatie gebruikt vier keer een *read-modify-write instruction* en de rechter een enkele toewijzing van een constante. De laatste methode heeft daarom de voorkeur.

Paragraaf 18.3 bespreekt diverse manieren om registers te initialiseren.

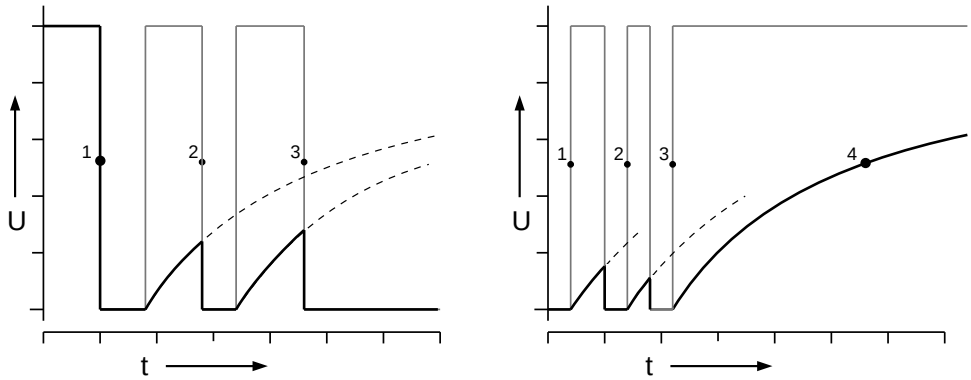
## 12.6 Contactdender

De meeste schakelaars en drukknoppen produceren contactdender. Het contact wordt niet in één keer gemaakt of in één keer verbroken. Figuur 12.6 toont de dender (*bouncing*) bij een maakcontact. In dit voorbeeld gaat het signaal niet een keer maar veertien keer van hoog naar laag. Dender heeft een mechanische oorzaak. Het is niet te voorspellen of het optreedt en in welke mate. Elke soort schakelaar heeft een ander gedrag. Dender hangt af van allerlei externe factoren, zoals vuil en slijtage. Soms is de duur van de dender een paar tiende milliseconde en soms duurt het enige milliseconden.




**Figuur 12.6 :** **De dender bij het indrukken van de drukknop.** Het signaal gaat veertien van hoog naar laag voordat het definitief laag is geworden.

Als het signaal een even aantal keer van hoog naar laag gaat, wordt er een even aantal interrupts gegenereerd. Het signaal van figuur 12.6 gaat veertien keer van hoog naar laag. Dit betekent dat de led in een paar milliseconde tijd zeven keer aan en uit gaat. Het effect is dat er niets zichtbaars gebeurt. Bij dender lijkt het systeem soms wel en soms niet goed te reageren.




**Figuur 12.7 :** **Het signaal van een drukknop met en zonder antidendercondensator.**
Het signaal zonder condensator is dun getekend en licht grijs. Het signaal met condensator is dik en zwart getekend. In de linker figuur wordt de drukknop ingedrukt. Zonder de condensator gaat het signaal denderen. Met de condensator wordt het signaal ook direct nul. Het stijgen van het signaal gaat — vanwege de pullupweerstand — veel langzamer. Zodat alleen bij overgang 1 het signaal van hoog naar laag gaat. Bij de andere overgangen is het signaal nog niet hoog genoeg geworden. In de rechter figuur wordt de drukknop losgelaten. Het signaal met condensator zal langzaam stijgen, maar wordt door de dender telkens nul. Na de laatste dender stijgt het signaal wel door en is bij 4 de enige overgang van laag naar hoog.

## 12.7 Hardwarematige antidendermaatregelen

Dender (*bouncing*) kan met software en met hardware worden voorkomen. In de literatuur en op het internet zijn verschillende antidenderschakelingen te vinden. Deze zijn te groeperen in een paar fundamentele oplossingen. Soms wordt van het

ingangssignaal een one-shot gemaakt, maar meestal wordt het signaal gefilterd. In de schakeling van figuur 12.2 is het ingangssignaal gefilterd door een condensator parallel bij de drukknop te plaatsen.
Het effect van deze condensator is in figuur 12.7 te zien. Bij het indrukken van de knop wordt het signaal snel laag en bij dender gaat het langzaam omhoog. Er is nu maar een enkele overgang van hoog naar laag. Bij het loslaten van de knop wordt het signaal langzaam hoog en bij dender weer snel laag. Pas na de laatste opgaande flank wordt het signaal na enige tijd hoog.
Omdat deze laatste overgang zo traag verloopt, is een schmitttrigger gewenst. In figuur 12.2 is geen schmitttrigger getekend, omdat er in de ATmega32 al een aanwezig is, zoals in figuur 11.3 te zien is.

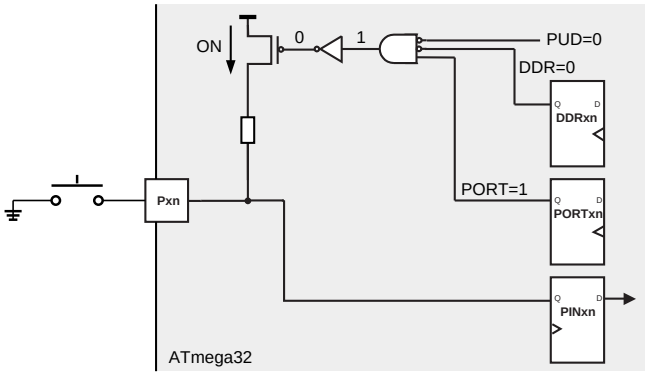
De schmitttrigger wordt besproken in bijlage D.9.

De waarde van de condensator hangt af van de waarde van de pullupweerstand en vooral van het soort schakelaar of drukknop dat gebruikt wordt. Meestal is een RC-tijd van 1 ms tot 10 ms een redelijke keuze. Voor een RC-tijd van 1 ms en een pullupweerstand van 10 kΩ is, moet de $C$ minimaal 100 nF zijn.

## 12.8 Softwarematige antidendermaatregelen

Dender (*bouncing*) kan ook softwarematig worden voorkomen. In de literatuur en op het internet zijn vele antidenderalgoritmen (*debouncing algorithms*) te vinden. Deze zijn te verdelen in twee soorten oplossingen: het ene type maakt gebruik van timers en interrupts en het andere gebruikt *polling* en vertragingsfuncties.
In paragraaf 13.6 staat een voorbeeld van een antidenderalgoritme met een timer en een interrupt. Het voordeel van deze methode is dat het hoofdprogramma ontlast wordt en de tijdplanning (*scheduling*) veel eenvoudiger is.
Bij de *polling* methode kijkt het programma op gezette tijden naar de pin met de schakelaar of drukknop. Het nadeel is dat in de meeste gevallen de knop niet ingedrukt wordt en dat de microcontroller daardoor tijd verspild.

Het aantrekkelijke van een softwarematige oplossing is dat er geen extra componenten nodig zijn, de PCB eenvoudiger is en het product goedkoper zal zijn. Als de interne pullupweerstand wordt gebruikt, kan de weerstand uit figuur 12.2 weggelaten worden.




**Figuur 12.8 : Een drukknop die is aangesloten op IO-cel met een interne pullup.** De pullup is actief als `PUD` en `DDRx` laag zijn en `PORTx` hoog is.

**Tabel 12.5 : De instelling van de pullup.**

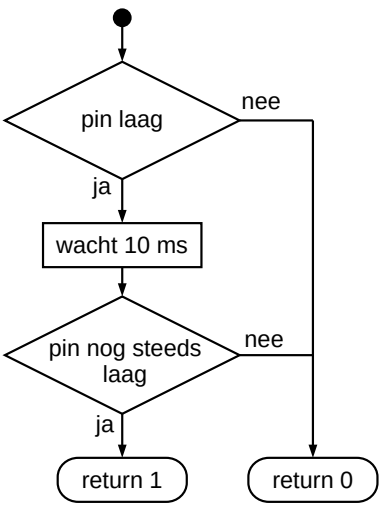
| PUD | DDRx | PORTx | pullup |
|---|---|---|---|
| 1 | - | - | uit (alle pullups zijn uit) |
| 0 | 1 | - | uit (de IO-cel is uitgang) |
| 0 | 0 | 0 | uit |
| 0 | 0 | 1 | aan |

In figuur 11.3 is te zien dat de ATmega32 in elke IO-cel een pullupweerstand en een pulluptransistor heeft. De transistor wordt door drie signalen aangestuurd: de uitgang van de DDR-flipflop van de IO-cel, de uitgang van de PORT-flipflop van de IO-cel en het PUD-bit uit het SFIOR-register. Als PUD (*Pull Up Disbled*) actief is, worden alle pullupweerstanden uitgeschakeld. Een pullup is alleen zinvol als de aansluiting een ingang is, daarom is de pullupweerstand uit als DDRx hoog is. Als de IO-cel als ingang werkt, wordt de PORTx-flipflop als registercel gebruikt om de pullup aan te kunnen zetten. Als PUD en DDRx laag zijn en PORTx hoog is, is de pullup ingeschakeld. In figuur 12.8 is een drukknop aangesloten op een IO-cel waarvan de pullup aan is. Tabel 12.5 geeft de signalen waarmee de pulluptransistor aan en uit wordt gezet.

Het programma van code 12.3 hoort bij figuur 12.2 maar met de aanpassing van figuur 12.8. Het programma gebruikt geen interrupt, maar het gebruikt *polling* om de drukknop uit te lezen.
De pullup van de ingang wordt op regel 17 bij de initialisatie aan gezet. Het hoofdprogramma roept iedere 0,2 seconde een functie `button_pressed` aan. Het algoritme staat in figuur 12.9 en zorgt ervoor dat de functie 1 teruggeeft als de knop ingedrukt is.

**Code 12.3 : Uitlezen van drukknop door middel van polling.**

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  int button_pressed(void)
5  {
6    if ( ! (PIND & _BV(2)) ) {
7      _delay_ms(10);
8      if ( ! (PIND & _BV(2)) ) return 1;
9    }
10
11   return 0;
12 }
13
14 int main(void)
15 {
16   DDRA  = _BV(0);
17   PORTD = _BV(2);  // pullup on
18
19   while (1) {
20     if ( button_pressed() ) {
21       PORTA ^= _BV(0);
22     }
23     _delay_ms(200);
24   }
25 }
```




**Figuur 12.9 : Het antidenderalgoritme om de drukknop uit te lezen.**
Als de ingang na 10 ms nog steeds laag is, is de knop ingedrukt.

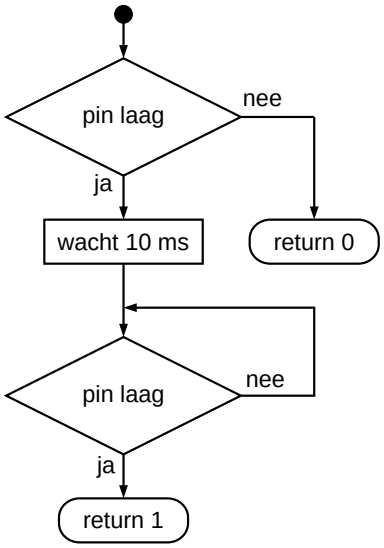
In dit voorbeeld is de ontdendertijd 10 ms. Het hoofdprogramma wacht 0,2 s. Als de knop langer dan 0,2 s wordt ingedrukt, wordt er opnieuw een druk op de knop geregistreerd. Dat kan ongewenst zijn. Natuurlijk kan de wachttijd langer gemaakt worden, maar dan bestaat juist de kans dat een korte indruk van de knop

gemist wordt. De tijdvertragingen voor het ontdenderen en het wachten hangen sterk af van het type knop en van wat de applicatie moet doen. Bij een toetsenbord is het juist plezierig dat de aanslag voortdurend herhaald wordt zolang de toets ingedrukt blijft.

Een alternatief is om de functie `button_pressed` te laten wachten totdat de drukknop losgelaten en de ingang weer hoog is. Het programma reageert pas als de knop losgelaten is. Het algoritme voor deze methode staat in figuur 12.10 en het programma staat in code 12.4. Het testen of bit 2 van poort D laag is, is in deze `button_pressed` gedaan met de macro `bit_is_clear`.
Het nadeel van deze methode is dat het programma blijft hangen zolang de gebruiker de knop ingedrukt houdt. Zeker bij programma's met een ingewikkelde tijdplanning (*scheduling*) kan dat er toe leiden dat het programma niet goed gaat functioneren.

**Code 12.4 : Alternatief voor uitlezen drukknop met behulp van polling.**

```
 1  #include <avr/io.h>
 2  #include <util/delay.h>
 3
 4  int button_pressed(void)
 5  {
 6    if ( bit_is_clear(PIND, 2) ) {
 7      _delay_ms(10);
 8      while ( bit_is_clear(PIND, 2) );
 9      return 1;
10    }
11
12    return 0;
13  }
14
15  int main(void)
16  {
17    DDRA  = _BV(0);
18    PORTD = _BV(2);   // pullup on
19
20    while (1) {
21      if ( button_pressed() ) {
22       PORTA ^= _BV(0);
23      }
24      _delay_ms(200);
25    }
26  }
```

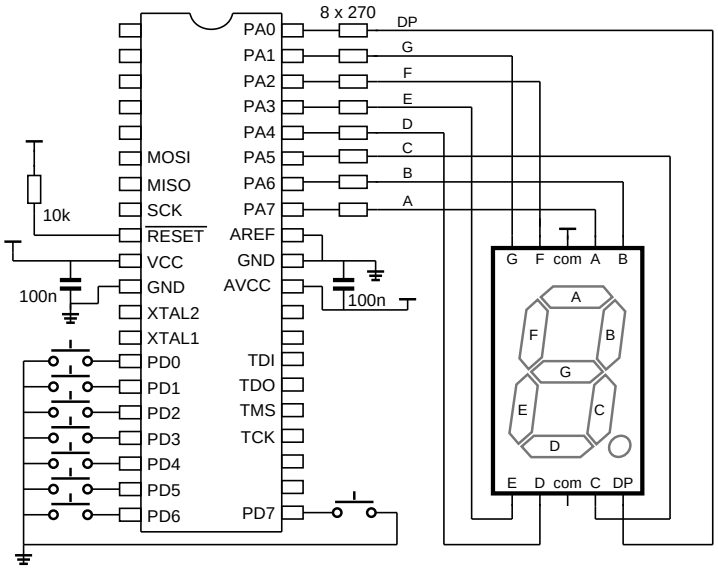



**Figuur 12.10 : Het antidenderalgoritme dat wacht op loslaten drukknop.**

## 12.9 Het uitlezen van acht drukknoppen met polling

In de schakeling van figuur 12.11 zijn acht drukknoppen verbonden met poort D van de microcontroller. Poort A is verbonden met de kathodes van het 7-segmentdisplay. De gemeenschappelijke anode van het display is direct met de voedingsspanning verbonden. De weerstanden zijn 270 Ω. De totale stroom die de poort

moet afvoeren, blijft dan ruim onder de 100 mA. De drukknoppen hebben geen pullupweerstand en geen condensator. In plaats daarvan worden de interne pullupweerstanden van de microcontroller gebruikt en worden de drukknoppen softwarematig ontdenderd.




**Figuur 12.11 : Schakeling met acht drukknoppen en een 7-segmentdisplay.**

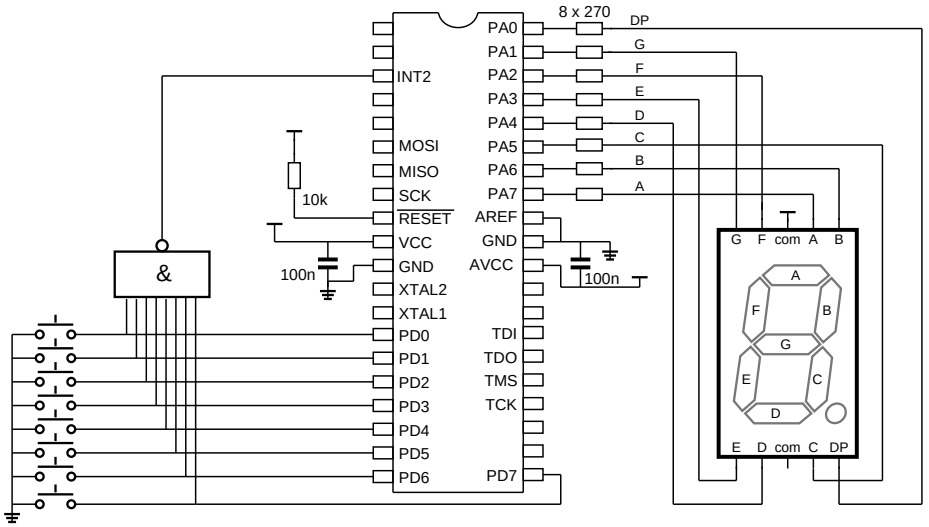
De applicatie voor de schakeling van figuur 12.11 zet op het display een 0, 1 of 7 als er respectievelijk op drukknop `PD0`, `PD1` of `PD7` wordt gedrukt.

Het programma staat in code 12.5. De waarden voor het 7-segmentdisplay staan in de opzoektabel `lookup`. Op regel 41 worden de pulluptransistoren aangezet. Het hoofdprogramma bevat een functie `button_pressed`, die het bitnummer van de drukknop teruggeeft. Als er geen knop ingedrukt is of als er meerdere knoppen ingedrukt zijn, geeft de functie $-1$ terug.

De functie leest eerst `PIND` en bewaart de geïnverteerde waarde in het achtbits integer `x`. Als er op een knop gedrukt is, is een bit hoog. De **`for`**-lus gaat na op welke van de acht knoppen gedrukt is. De uitdrukking `x == _BV(i)` vergelijkt de waarde van `x` met respectievelijk $2^i$. Als deze uitdrukking waar is, wordt de variabele `bitnumber` gelijk aan $i$ en wordt de **`for`**-lus afgebroken. Als er meerdere knoppen ingedrukt zijn, is `x` altijd ongelijk aan $2^i$ en blijft het bitnummer $-1$. Als er op geen enkele knop gedrukt is, is het bitnummer ook $-1$.

Als `bitnumber` negatief is, wordt er $-1$ teruggegeven, anders wordt er eerst 10 ms gewacht. Daarna wordt er gewacht totdat er geen enkele knop ingedrukt is. Tenslotte wordt het gevonden bitnummer teruggegeven.

De oneindige lus van het hoofdprogramma test met de functie `button_pressed` voortdurend of er op een knop gedrukt is. Het bitnummer dat de functie teruggeeft is de index van het bijbehorende cijfer uit de opzoektabel.




**Figuur 12.12 : Schakeling met acht drukknoppen en NAND voor interrupt.**

## 12.10 Het uitlezen van acht knoppen met externe interrupt 2

Een reden om gebruik te maken van *polling* is het feit dat de ATmega32 slechts drie interrupt aansluitingen heeft. Bij meer drukknoppen moet daarom *polling* worden gebruikt of er is extra hardware nodig. Figuur 12.12 gebruikt een NAND om een interruptsignaal te maken. De ingangen van poort D zijn hoog door de interne pullupweerstanden van de microcontroller. Als er geen knop ingedrukt is, zijn de ingangen van de NAND hoog en is het interruptsignaal laag. Als er een knop ingedrukt wordt, is dit signaal hoog.

De ATmega32 heeft in tegenstelling tot oudere ATmega's drie interrupts. De derde interrupt wijkt af van de andere twee. De instelling voor de gevoeligheid staat niet in het `MCUCR`-register, maar in het MCU Control and Status Register. Als het `ISC2`-bit laag is, is deze interrupt gevoelig voor een neergaande flank en als dit bit hoog is voor een opgaande flank. In figuur 12.13 staat `GICR`-register en in figuur 12.14 het `MCUCSR`-register.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| INT1 | INT0 | **INT2** | - | - | - | IVSEL | IVCE |

**Figuur 12.13 : Het `GICR`-register met het `INT2`-bit.**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| JTD | **ISC2** | - | JRTF | WDRF | BORF | EXTRF | PORF |

**Figuur 12.14 : Het `MCUCSR`-register met het `ISC2`-bit.**

De code voor de schakeling uit figuur 12.12 staat in code 12.6. Deze lijkt op code 12.5 die gebruikt is om de drukknoppen met *polling* uit te lezen.
In plaats van een functie `button_pressed` heeft het programma van code 12.6 een interruptfunctie `ISR(INT2_vect)`. Deze functie lijkt op `button_pressed`, maar heeft net als alle andere interruptfuncties geen retourwaarde. Daarom is de toekenning

**Code 12.5 : Het uitlezen van acht drukknoppen met behulp van polling.**

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


const unsigned char lookup[] = {
  0x03, //  0000 0011  = 0
  0x9F, //  1001 1111  = 1
  0x25, //  0010 0101  = 2
  0x0D, //  0000 1101  = 3
  0x99, //  1001 1001  = 4
  0x49, //  0100 1001  = 5
  0x41, //  0100 0001  = 6
  0x1F  //  0001 1111  = 7
};

int button_pressed(void)
{
  uint8_t x;
  int i, bitnumber;

  x = ~(PIND);
  bitnumber = -1;
  for(i=0; i<8; i++) {
    if ( x==_BV(i) ) {
      bitnumber = i;
      break;
    }
  }
  if (bitnumber < 0) return bitnumber;
  _delay_ms(10);
  while ( PIND != 0xFF );

  return bitnumber;
}

int main(void)
{
  int b;

  DDRA   = 0xFF;
  PORTD  = 0xFF;  // pullups on

  while (1) {
    if ( (b = button_pressed()) >= 0 ) {
      PORTA = lookup[b];
    }
  }
}

```

**Code 12.6 : Het uitlezen van acht drukknoppen met behulp van een interrupt.**

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

const unsigned char lookup[] = {
  0x03, //  0000 0011  = 0
  0x9F, //  1001 1111  = 1
  0x25, //  0010 0101  = 2
  0x0D, //  0000 1101  = 3
  0x99, //  1001 1001  = 4
  0x49, //  0100 1001  = 5
  0x41, //  0100 0001  = 6
  0x1F  //  0001 1111  = 7
};

ISR(INT2_vect)
{
  uint8_t x;
  int i, bitnumber;

  x = ~(PIND);
  bitnumber = -1;
  for(i=0; i<8; i++) {
    if ( x == _BV(i) ) {
       bitnumber = i;
       break;
    }
  }
  if (bitnumber < 0) return;
  _delay_ms(10);
  while ( PIND != 0xFF );

  PORTA = lookup[bitnumber];
}

int main(void)
{
  DDRA   = 0xFF;
  PORTA  = 0xFF;        // leds on
  PORTD  = 0xFF;        // pullups on
  GICR   = _BV(INT2);   // interrupt 2 on
  MCUCSR = _BV(ISC2);   // rising edge

  sei();

  while (1) {
    asm volatile ("nop");
  }
}
```

aan PORTA in de interruptfunctie opgenomen. Het hoofdprogramma doet na de initialisatie van de in- en uitgangen en de initialisatie van de interrupts niets. De oneindige lus bevat alleen een no-operating instructie.
De interruptfunctie is niet ideaal, omdat het een tijdvertraging bevat. Als er op een knop gedrukt wordt, blijft het programma minimaal 10 ms in de interruptfunctie en kan gedurende deze tijd niet op andere interrupts reageren. Dit probleem kan worden opgelost door voor de tijdvertraging een timer te gebruiken. In code 13.5 uit paragraaf 13.6 staat hiervan een voorbeeld.

# 13

# Timers

| | |
|---|---|
| Doelstelling | In dit hoofdstuk leer je wat een timer is, hoe deze is opgebouwd en welke registers hiervoor gebruikt worden. Je leert om met timer 0 en met timer 2 van de ATmega32 een tijdvertraging te maken. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De opbouw van timer 0 bij de ATmega32.<br>▪ De keuze van de kristaloscillator.<br>▪ Het berekenen van de parameters voor een exacte tijdvertraging.<br>▪ Het maken van een vaste tijdvertraging.<br>▪ Het maken van een real time clock.<br>▪ Een antidenderalgoritme met een timer.<br><br>De voorbeelden demonstreren het gebruik van:<br>▪ Het knipperen van een led met behulp van timer 0.<br>▪ Een real time klok met timer 2.<br>▪ Een antidenderalgoritme met behulp van externe interrupt 0 en timer 0. |

Tijd is bij microcontrollers belangrijk. Embedded systemen hebben vaak een beperkte taak, kennen een beperkt aantal instellingen, maar moeten op gezette tijden hun functies uitvoeren. De elektrische tandenborstel gaat na twee minuten een paar keer uit en aan om de gebruiker te laten weten dat hij lang genoeg gepoetst heeft. De kamerthermostaat hoeft niet elk microseconde de temperatuur te controleren.

De manier om met een digitaal systeem tijd te meten is klokpulsen tellen. Een 16-bits teller met een klokfrequentie van 32768 Hz geeft elke twee seconde een overflow.

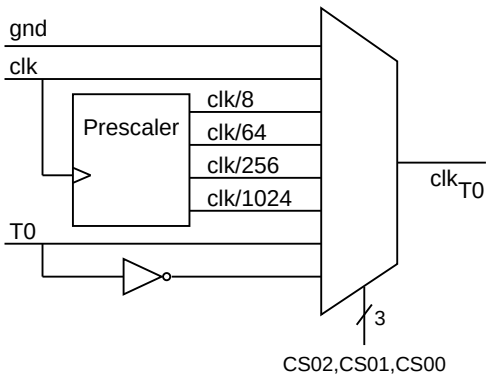
Digitaal is tijd $t$ gelijk aan het een aantal getelde klokpulsen $m$ vermenigvuldigd met de klokperiode van het systeem $T_{\text{cpu}}$.

$$t = mT_{\text{cpu}} \tag{13.1}$$

Daarom hebben microcontrollers meestal een of meerdere tellers. Omdat een teller gebruikt kan worden om te tellen en om tijd te meten, wordt deze soms *counter* en soms *timer* genoemd.

## 13.1 Timer 0

De ATmega32 heeft twee 8-bits tellers — timer 0 en timer 2 — en een 16-bits teller – timer 1 — met elk een aantal specifieke mogelijkheden. Deze paragraaf laat zien hoe timer 0 gebruikt kan worden om een nauwkeurige tijd te definiëren.
Een 8-bits teller kan maximaal 256 klokslagen tellen. Zonder speciale voorzieningen betekent dit dat met een systeemklok van 4 MHz ($T_{\text{cpu}} = 0{,}25\ \mu$s) de maximaal te meten tijd slechts 64 $\mu$s is. Daarom heeft timer 0, net als de andere tellers, een klokdeler (*prescaler*). De ATmega gebruikt hiervoor bij timer 0 een schakeling die de systeemklok deelt, maar die ook een externe klok kan selecteren, die aangesloten is op pin T0 (PB0). Als GND wordt geselecteerd is er geen klok en telt de teller niet.




**Figuur 13.1 : De klokdelerschakeling van de timer.** De klok voor timer 0 ($\text{clk}_{\text{T0}}$) kan een gedeelde klok zijn van de prescaler of een externe klok via pin T0.

In figuur 13.1 is te zien dat deze schakeling uit een klokdeler en een 8-to-1 multiplexer bestaat. Met de drie *Counter Select* bits CS02, CS01 en CS00 uit het register TCCR0 wordt de juiste klok geselecteerd, zie ook tabel 13.1.

**Tabel 13.1 : De selectiebits voor het instellen van de klok van timer 0.**

| CS02 | CS01 | CS00 | klokconfiguratie |
|---|---|---|---|
| 0 | 0 | 0 | Timer 0 is uit |
| 0 | 0 | 1 | $\text{clk}_{\text{T0}} = \text{clk}$ |
| 0 | 1 | 0 | $\text{clk}_{\text{T0}} = \text{clk}/8$ |
| 0 | 1 | 1 | $\text{clk}_{\text{T0}} = \text{clk}/64$ |
| 1 | 0 | 0 | $\text{clk}_{\text{T0}} = \text{clk}/256$ |
| 1 | 0 | 1 | $\text{clk}_{\text{T0}} = \text{clk}/1024$ |
| 1 | 1 | 0 | $\text{clk}_{\text{T0}}$ is externe pin T0 telt bij stijgende klokflank |
| 1 | 1 | 1 | $\text{clk}_{\text{T0}}$ is externe pin T0 telt bij dalende klokflank |

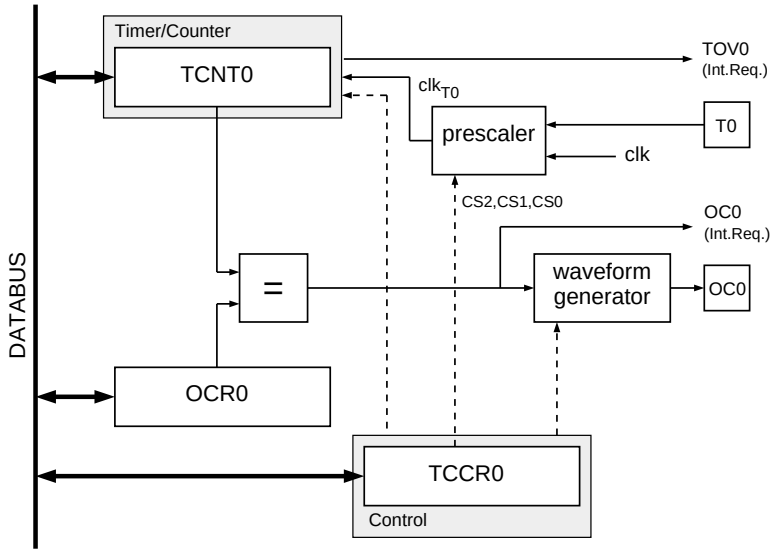
Met een klokdeling is de tijd $t$ afhankelijk van de prescaling $P$, het aantal getelde pulsen $m$ en de klokperiode van het systeem $T_{\text{cpu}}$.

$$t = mPT_{\text{cpu}} \tag{13.2}$$

Een klok met een frequentie van 4 MHz heeft een periodetijd van 0,25 $\mu$s.

Bij 4 MHz geldt voor de maximaal te meten tijd van timer 0 dat: $m = 256$, $P = 1024$ en $T_{\text{cpu}} = 0{,}25\ \mu$s is, zodat deze gelijk is aan 65,5 ms.

In figuur 13.2 staat het blokschema van timer 0. De timer bevat een teller TCNT0, een register OCR0, de prescaler van figuur 13.1, een *waveform generator* en een register met de configuratiebits voor de teller, de *waveform generator* en de *prescaling*.




**Figuur 13.2 : Het blokschema van timer 0.**

Dit hoofdstuk gebruikt de timer 0 en 2 is de normale modus. Bovendien wordt de *output compare*-uitgang van de timer niet gebruikt. Hoofdstuk 22 bespreekt de andere modi van de timers en laat zien hoe PWM-signalen worden gegenereerd kunnen worden.

Register TCNT0 (*Timer/CouNTer 0*) bevat de waarde van de teller, die via de databus kan worden uitgelezen of gewijzigd. Het register OCR0 (*Output Compare Register 0*) bevat een 8-bits getal waarmee de waarde van de teller kan worden vergeleken. Register (*Timer/Counter Control Register 0*) bevat de instelling van de timer, waaronder de drie configuratiebits CS2, CS1 en CS0 voor de prescaling. Met de waveform generator kunnen verschillende soorten PWM-signalen worden gemaakt. De timer heeft twee interruptsignalen: TOV0 (*Time OVerflow 0*) wordt hoog als de teller vol is en OC0 (*Output Compare 0*) wordt hoog als de waarde van de teller gelijk is aan de waarde in OCR0.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| TCCR0 | FOC0 | WGM00 | COM01 | COM00 | WGM01 | **CS02** | **CS01** | **CS00** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TIMSK | OCIE2 | TOIE2 | TICIE1 | OCIE1A | OCIE1B | TOIE1 | OCIE0 | **TOIE0** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TIFR | OCF2 | TOV2 | ICF1 | OCF1A | OCF1B | TOV1 | OCF0 | **TOV0** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TCNT0 | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OCR0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**Figuur 13.3 : De bits en de registers die nodig zijn bij timer 0.**
Deze bits voor timer 0 hebben een witte achtergrond. De bits met een lichtgrijze achtergrond worden door timer 1 en 2 gebruikt. De bits die bij een timer met de overflow interrupt gebruikt worden, zijn vet gezet.

Naast de drie registers uit figuur 13.2 zijn er ook een aantal bits nodig uit de algemene timerregisters TIMSK (*Timer/counter Interrupt MaSK register*) en TIFR (*Timer/counter Interrupt Flag Register*). TIMSK bevat twee bits TOIE0 (*Timer/counter0 Overflow Interrupt Enable*) en OCIE0 (*timer/counter0 Output Compare match Interrupt Enable*). Hiermee worden respectievelijk de overflow interrupt en de output compare interrupt aangezet. TIFR bevat twee vlaggen TOV0 (*Timer/counter0 OVerflow Flag*) en OCF0 (*Output Compare Flag 0*). Deze worden door de microcon-

troller hoog gemaakt, als er respectievelijk een overflow interrupt en de output compare interrupt optreedt. Alle registers voor timer 0 staan in het overzicht van figuur 13.3.

Timer/counter 0 is — net als de andere timer/counters — te gebruiken als:

- een timer die een overflow interrupt geeft als de teller vol is;
- een timer die een overflow geeft als de teller gelijk is aan de waarde in het output compare register;
- een PWM signaalgenerator;

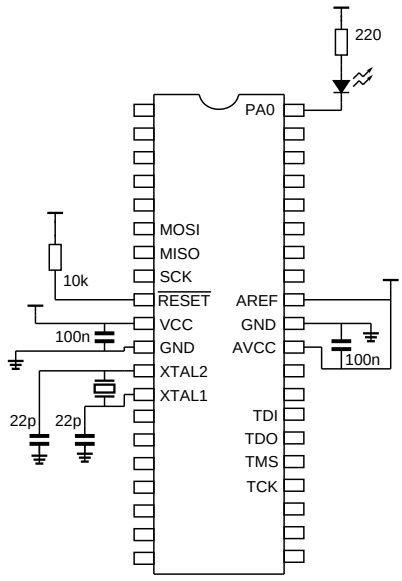
Dit hoofdstuk demonstreert het gebruik van timer 0 als timer met een overflow interrupt. De bits uit de verschillende registers, die hierbij gebruikt worden, zijn in figuur 13.3 vet gezet. Hoofdstuk 22 bespreekt uitgebreid de andere mogelijkheden van de timers en het genereren van PWM-signalen.

## 13.2 De schakeling voor het testen van de timer/counter

Timer 0 kan getest worden met de schakeling uit figuur 11.1. In figuur 13.4 is deze schakeling opnieuw getekend, alleen is er een extern kristal 4 MHz toegevoegd. Voor het maken van nauwkeurige tijden moet de systeemklok stabiel en nauwkeurig zijn. Tabel 13.2 geeft een beeld van de nauwkeurigheden van de oscillatoren. Een externe kristaloscillator is ongeveer honderd keer nauwkeuriger dan een keramische oscillator. De nauwkeurigheid van de interne RC-oscillator is veel slechter; ook als deze softwarematig gekalibreerd is.

**Tabel 13.2 : De nauwkeurigheid van de diverse oscillatoren.**

| frequentie | kristal | keramisch | intern RC | intern RC gekalibreerd |
|---|---|---|---|---|
| | ±20 ppm | ±2000 ppm | ±3% | ±1% |
| 1 MHz | 20 Hz | 2 kHz | 30 kHz | 10 kHz |
| 4 MHz | 80 Hz | 8 kHz | 120 kHz | 30 kHz |




**Figuur 13.4 : De schakeling voor het testen van de timer.**

Het kristal is aangesloten op de ingangen XTAL1 en XTAL2 van de microcontroller. Het kristal moet dicht bij de microcontroller worden geplaatst en er moeten twee condensatoren van 22 pF worden aangebracht. Deze condensatoren zorgen ervoor dat de oscillator op de juiste wijze gaat oscilleren. Bij het programmeren moeten de vier klokselectiebits CKSEL[3:0] voor de klok 1111 worden gemaakt voor de juiste klokconfiguratie, zie ook paragraaf 10.3.

## 13.3 Berekening parameters voor exacte tijdvertraging

De *duty cycle* is de verhouding tussen de tijd dat een systeem actief is ten opzchte van de totale tijd. Bij periodieke signalen is dat de verhouding tussen de tijd dat een signaal actief (hoog) is en de periodetijd.

Stel dat de led moet knipperen met een frequentie van exact 1,000 Hz en dat de *duty cycle* 50% is. De led moet dan 500 ms aan en 500 ms uit zijn. Daar is een tijdvertraging van exact 500 ms nodig.

De maximale tijd tussen twee interrupts kan met formule 13.2 worden berekend. Voor timer 0 is $m$ maximaal 256 en $P$ maximaal 1024. Voor een frequentie van 4 MHz is de maximale vertragingstijd ongeveer 65,5 ms. Dat is een factor 7,6 korter dan 500 ms nodig is.

Langere tijden worden gemaakt door een aantal interrupts te tellen. In de interruptroutine, die de uitgang laat toggelen, komt dan iets dergelijks te staan:

```
if (++interruptcount == aantal te tellen interrupts)  {
   toggle bit0 van port A
   interruptcount = 0;
}
```

De variabele interruptcount wordt bij elke interrupt een opgehoogd. Als het aantal te tellen interrupts bereikt wordt, toggelt het bit en wordt interruptcount nul gemaakt. Als $n$ het aantal te tellen interrupts is, dan geldt voor de vertragingstijd $t_{\text{delay}}$:

$$t_{\text{delay}} = nmPT_{\text{cpu}} \tag{13.3}$$

Het aantal te tellen interrupts $n$, het aantal klokslagen van de teller $m$ en de prescaling $P$ zijn alle drie gehele getallen. In dit geval is de klokfrequentie ($f_{\text{cpu}}$) gegeven en is er een bepaalde vertraging nodig. Het product van $n$ en $m$ is het totaal aantal gedeelde (*prescaled*) klokslagen en daarvoor geldt:

$$nm = \frac{t_{\text{delay}} f_{\text{cpu}}}{P} \tag{13.4}$$

Het product $nm$ is in tabel 13.3 gegeven voor de verschillende prescalewaarden.

**Tabel 13.3 : Het totaal aantal gedeelde (prescaled) klokslagen nodig voor een tijdvertraging van 500 ms bij een klokfrequentie van 4 MHz.**

| $P$ | 1 | 8 | 64 | 256 | 1024 |
|---|---|---|---|---|---|
| $nm$ | 2000000 | 250000 | 31250 | 7812,5 | 1953,125 |

Het totaal aantal klokslagen $nm$ dat de timer maakt, moet altijd een geheel getal zijn. De prescaling van 256 en 1024 levert een gebroken getal op en voldoen daarom niet. Het is zaak $m$ en $P$ zo groot mogelijk te kiezen. Dat levert het kleinste aantal interrupts op en verstoort het hoofdprogramma het minst. Voor $m$ groter of gelijk aan 100 staan in tabel 13.4 alle combinaties van $m$ en $P$ die een geheeltallige waarde van $n$ opleveren.

**Tabel 13.4 : Het aantal interrupts voor verschillende waarden van m.**

| $P$ | 1 | 8 | 64 | 256 | 1024 |
|---|---|---|---|---|---|
| $nm$ | 2000000 | 250000 | 31250 | 7812,5 | 1953,125 |
| $m = 250$ | 8000 | 1000 | 125 | | |
| $m = 200$ | 10000 | 1250 | | | |
| $m = 160$ | 12500 | | | | |
| $m = 128$ | 15625 | | | | |
| $m = 125$ | 16000 | 2000 | 250 | | |
| $m = 100$ | 20000 | 2500 | | | |

Door $m$ en $P$ zo groot mogelijk te kiezen, is $n$ klein. De microcontroller besteedt dan de minste tijd aan het meten van de vertraging. Voor de interruptfunctie van code 13.1 zijn ongeveer 60 klokslagen nodig. De interrupt duurt bij 4 MHz dan ongeveer 15 $\mu$s. Als $n$ gelijk aan 125 is, wordt er 1,875 ms van de 500 ms besteed aan het meten van de tijdvertraging. Dat is ongeveer 0,4% van de processortijd. Bij een prescaling van 8 is dat 3% en zonder prescaling is dat 24%!

Een verstandig keus is om de prescaling 64 te nemen en de timer te laten tellen tot 250 ($= m$). Dan is het aantal interrupts $n$ dat geteld moet worden het kleinst, namelijk 125.

## 13.4 Software voor testen timer 0

In code 13.1 staat een Interrupt Service Routine voor timer 0. Deze laat eens in de 125 keer de uitgang bit 0 van poort A toggelen. De teller start steeds bij zes. Dit betekent dat de teller 250 gedeelde klokslagen telt. Bij de initialisatie aan het begin van het hoofdprogramma wordt de prescaling 64 gemaakt. De vertragingstijd bij een klokfrequentie van 4 MHz is dan 500 ms ($125 \times 250 \times 64 \times 0{,}25$ $\mu$s).
Het hoofdprogramma bestaat naast de initialisatie voor de uitgang en de initialisatie voor de timer en uit een oneindige lus die niets doet. In de praktijk staat in deze lus het hoofdprogramma.

**Code 13.1 : Knipperen van een led met behulp van timer 0. ($f_{cpu} = 4$ MHz)**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  unsigned int interruptcount = 0;
5
6  ISR(TIMER0_OVF_vect)
7  {
8    TCNT0 = 6;                      // it counts 256-6= 250 prescaled clockcycles
9
10   if (++interruptcount == 125) {  // toggles only once of 125 interrupts
11      PORTA = PORTA ^ _BV(0);
12      interruptcount = 0;
13   }
14 }
15
16 int main(void) {
17   DDRA  = _BV(0);
18
19   TCCR0 = _BV(CS01) | _BV(CS00);  // normal mode, OC not used, prescaling is 64
20   TCNT0 = 6;                      // timer 0 counts 256-6= 250 prescaled clockcycles
21   TIMSK = _BV(TOIE0);             // enable timer 0 interrupt overflow
22
23   sei();
24
25   while(1);                       // do nothing
26 }
```

Paragraaf 22.2 geeft meer informatie over de normale modus van timer 0

Al eerder is gezegd dat het verstandig is de prescaling $P$ en het te tellen aantal gedeelde (*prescaled*) klokslagen $m$ zo groot mogelijk te kiezen. Het aantal te tellen interrupts is dan klein en het meten van de tijd kost dan weinig processortijd.
Er is nog een reden om $P$ groot te kiezen. Voor het opstarten van de interrupt is een flink aantal klokslagen nodig. De interrupt flag in het statusregister moet worden gecleard. De huidige waarde van de programcounter moet op de stack worden gezet. Om de interrupt routine te vinden zijn een aantal sprongopdrachten nodig. Bij de GNU-compiler zijn ongeveer 25 klokslagen nodig om bij regel 8 te komen. Bij $P = 8$ betekent dat drie gedeelde klokslagen. Als er daarna 250 gedeelde klokslagen geteld worden, komt de verstreken tijd tussen twee interrupts overeen met 253 gedeelde klokslagen. De tijdvertraging is dan niet 500 ms maar 506 ms. Een oplossing is om regel 8 te vervangen door

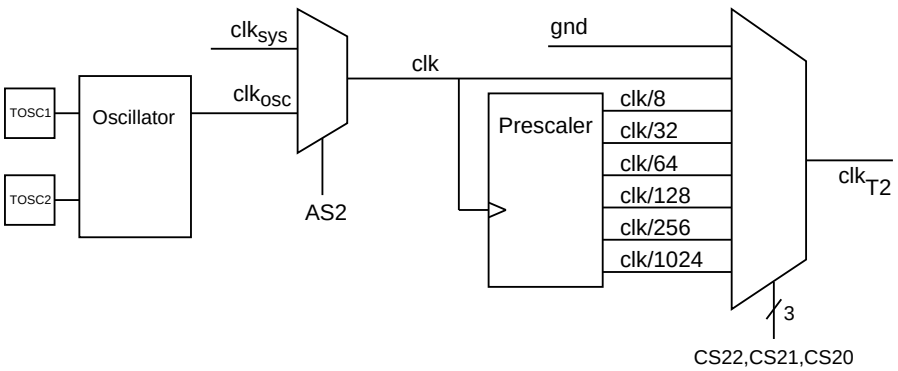
Er kan nog meer mis gaan. Voor het verhogen van TCNT0 zijn meerdere klokslagen nodig. Eerst wordt de waarde opgehaald, daarna wordt er 6 bijgeteld en vervolgens wordt dit weer aan TCNT0 toegekend. Als de timer ondertussen verhoogd is, wordt er een gedeelde klokslag te veel geteld.

```
        TCNT0 += 6;
```

Door TCNT0 met zes op te hogen worden er weer 250 gedeelde klokslagen geteld.

## 13.5 Real time clock met timer 2

Elke timer heeft zijn specifieke eigenschappen en mogelijkheden. Voor een *real time clock* is het beter om timer 2 te gebruiken. Deze timer heeft de mogelijkheid een eigen laag frequent kristaloscillator toe te passen. Timer 2 werkt dan op een nauwkeurige frequentie van 32768 Hz, terwijl het hele systeem op snelle frequentie werkt. Het kristal wordt aangesloten op de pinnen TOSC1 en TOSC2; extra condensatoren zijn niet nodig. Intern is de oscillator geoptimaliseerd voor 32768 Hz.

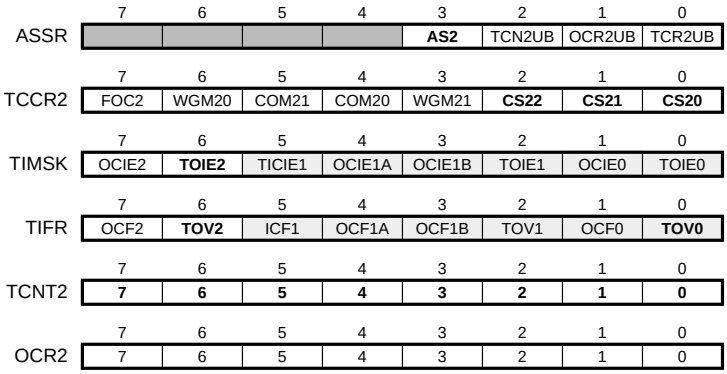



**Figuur 13.5** : **De klokdelerschakeling van de timer 2.** De klok voor timer 2 ($clk_{T2}$) is een gedeelde klok van de systeemklok ($clk_{cpu}$) of van de oscillator ($clk_{osc}$).

In figuur 13.5 staat de prescaler van timer 2. Deze prescaler heeft geen aansluiting voor een externe klok en kan de klok ook door 32 en 128 delen. Als bit AS2 uit het *ASynchronous Status Register* (ASSR) hoog is, wordt het signaal uit de oscillator als klok voor deze teller gebruikt. Figuur 13.6 toont de bits en registers van timer 2. Deze zijn overeenkomstig met die van timer 0. Alleen gebruikt timer 2 vier bits uit het ASSR-register, zie tabel 13.5
Code 13.4 bevat een opzet voor een real time clock. De functie updateRTC() wordt in de Interrupt Service Routine van code 13.2 aangeroepen. Als op TOSC1 en TOSC2 een kristal is aangesloten is er elke seconde een interrupt en worden de seconden en eventueel de overige datum- en tijdvariabelen aangepast.

**Tabel 13.5 : De selectiebits voor het instellen van de klok van timer 2.**

| AS2 | CS22 | CS21 | CS20 | $clk_{T2}$ |
|---|---|---|---|---|
| | 0 | 0 | 0 | Timer 2 is uit |
| 0 | 0 | 0 | 1 | $clk_{cpu}$ |
| 0 | 0 | 1 | 0 | $clk_{cpu/8}$ |
| 0 | 0 | 1 | 1 | $clk_{cpu/32}$ |
| 0 | 1 | 0 | 0 | $clk_{cpu/64}$ |
| 0 | 1 | 0 | 1 | $clk_{cpu/128}$ |
| 0 | 1 | 1 | 0 | $clk_{cpu/256}$ |
| 0 | 1 | 1 | 1 | $clk_{cpu/1024}$ |
| 1 | 0 | 0 | 1 | $clk_{osc}$ |
| 1 | 0 | 1 | 0 | $clk_{osc/8}$ |
| 1 | 0 | 1 | 1 | $clk_{osc/32}$ |
| 1 | 1 | 0 | 0 | $clk_{osc/64}$ |
| 1 | 1 | 0 | 1 | $clk_{osc/128}$ |
| 1 | 1 | 1 | 0 | $clk_{osc/256}$ |
| 1 | 1 | 1 | 1 | $clk_{osc/1024}$ |

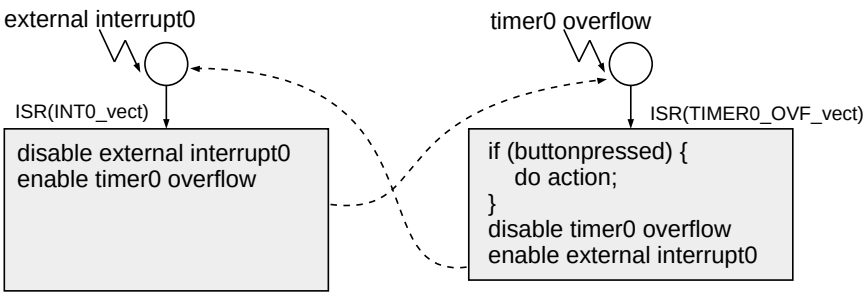



**Figuur 13.6 : De bits en de registers van timer 2.**
Deze bits voor timer 2 hebben een witte achtergrond. De bits met een lichtgrijze achtergrond worden door timer 0 en 1 gebruikt. De bits met een donkergrijze achtergrond worden niet gebruikt. De bits, die in deze paragraaf worden gebruikt, zijn vet gezet.

Op regel 13 in code 13.2 is de externe oscillator geselecteerd en op regel 13 is de klokdeling ingesteld op 128 door de configuratiebits `CS22` en `CS20` hoog te maken.

## 13.6 Een antidenderalgoritme met timer 0

In hoofdstuk 12 is het denderprobleem van een schakelaar of drukknop hardware en softwarematig opgelost. Er zijn vele manieren om dat te doen. Hier wordt nog een oplossing gegeven met een timer en een interrupt.
In essentie komen de methoden met een interrupt altijd op hetzelfde neer. Na de interrupt door de drukknop wordt er een tijdje gewacht en wordt er gekeken of de drukknop dan nog steeds ingedrukt is. Alle tussenliggende veranderingen worden genegeerd. Het tijdje wachten kan prima met een timer gedaan worden. Er zijn dan twee Interrupt Service Routines: een voor de externe interrupt en een voor de timer.




**Figuur 13.7 : Het antidenderalgoritme.** Aanvankelijk is alleen de `ISR(INT0_vect)` actief. Als er op de knop wordt gedrukt, wordt `ISR(TIMER0_OVF_vect)` ingeschakeld en `ISR(INT0_vect)` uitgeschakeld. Na de timer 0 interrupt wordt `ISR(TIMER0_OVF_vect)` uitgeschakeld en `ISR(INT0_vect)` weer ingeschakeld.

**Code 13.2 : Bestand rtc_test.c.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include "rtc.h"
4
5  ISR(TIMER2_OVF_vect)
6  {
7    updateRTC();
8  }
9
10 int main(void) {
11   DDRA  = _BV(0);
12
13   ASSR  = _BV(AS2);
14   TCCR2 = _BV(CS22)|_BV(CS20);
15   TCNT2 = 0;
16   TIMSK = _BV(TOIE2);
17
18   sei();
19
20   while(1);
21 }
```

**Code 13.3 : Bestand rtc.h.**

```
1
2  extern unsigned int seconds;
3  extern unsigned int minutes;
4  extern unsigned int hours;
5  extern unsigned int day;
6  extern unsigned int month;
7  extern unsigned int year;
8
9  void updateRTC(void);
```

**Code 13.4 : Bestand rtc.c voor realtime clock.**

```
1  unsigned int second = 0;
2  unsigned int minute = 0;
3  unsigned int hour   = 0;
4  unsigned int day    = 1;
5  unsigned int month  = 1;
6  unsigned int year   = 2007;
7
8  unsigned char mTable[] = {31,28,31,30,31,30,
9                            31,31,30,31,30,31};
10
11 void updateRTC(void)
12 {
13   if ( ++second < 60 ) return;
14   second = 0;
15   if ( ++minute < 60 ) return;
16   minute = 0;
17   if ( ++hour < 24 ) return;
18   hour = 0;
19   if ( ++ day < (mTable[month-1] + 1) ) return;
20   if ( day == 29 ) {
21     if ( (((year % 4)==0) && ((year % 100) != 0)) ||
22          ((year % 400) == 0) ) {
23       return;
24     }
25   }
26   day = 1;
27   if ( ++month <= 12) return;
28   month = 1;
29   year++;
30 }
```

De truc is dat aanvankelijk alleen de externe interrupt actief is en dat — als er een externe interrupt is — deze zichzelf uitschakelt en tegelijkertijd een timer start. Na de timer interrupt kijkt de timer interrupt routine of de knop nog steeds ingedrukt is. Als dat zo is wordt de gewenste activiteit uitgevoerd. Vervolgens wordt de timer uitgezet en de externe interrupt weer aangezet. Figuur 13.7 geeft dit algoritme met de twee interrupt service routines weer.

De ISR voor de externe interrupt schakelt zichzelf uit. Dit is gedaan om te voorkomen dat deze ISR weer opnieuw aangeroepen wordt. Alleen de eerste neergaande flank moet immers gedetecteerd worden.
De timer interrupt schakelt zichzelf ook uit. Dit zorgt er voor dat de timer interrupt alleen aangeroepen wordt na het indrukken van de knop. Anders wordt de actie steeds opnieuw uitgevoerd zolang de knop ingedrukt blijft.

Code 13.5 geeft het programma met het antidenderalgoritme. In de ISR voor externe interrupt 0 wordt op regel 7 register TCNT0 gelijk aan 56. De timer telt

**Code 13.5 : Antidender algoritme met timer 0. ($f_{cpu} = 1$ MHz)**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3
 4  ISR(INT0_vect)
 5  {
 6    GICR  &= ~(_BV(INT0));  // disable INT0
 7    TCNT0  = 56;            // count 200 prescaled cycles
 8    TIMSK |= _BV(TOIE0);    // enable TIMER0 overflow
 9    TIFR  &= ~(_BV(TOV0));  // clear TOV0 flag
10  }
11
12  ISR(TIMER0_OVF_vect)
13  {
14    if ( bit_is_clear(PIND,2) ) {
15      PORTA ^= _BV(0);
16    }
17    TIMSK &= ~(_BV(TOIE0)); // disable TIMER0 overflow
18    GICR  |= _BV(INT0);     // enable INT0
19  }
20
21  int main(void)
22  {
23    DDRA  = _BV(0);
24
25    GICR  = _BV(INT0);      // enable INT0
26    MCUCR = _BV(ISC01);     // falling edge
27    TCCR0 = _BV(CS01);      // prescaling is 8
28
29    sei();
30
31    while(1);
32  }
```

daardoor tweehonderd gedeelde klokslagen ($m = 200$). Op regel 7 is de klokdeling ingesteld op acht ($P = 8$). Voor een frequentie van 1 MHz ($T_{cpu} = 1\mu s$) is met vergelijking 13.2 de vertraging 1,6 ms. In de ISR voor externe interrupt 0 is het verder nodig om op regel 9 het TOV0-bit (timer0 interrupt flag) laag te maken. Als de timer overflow interrupt wordt uitgezet, gaat de teller gewoon door met tellen. De teller zal om 1,6 ms een overflow geven, zodat het TOV0-bit voortdurend hoog is, met als gevolg dat ogenblikkelijk na een externe interrupt de timer interrupt routine uitgevoerd wordt.
Op regel 14 wordt getest of het signaal op de interrupt ingang laag is. Externe interrupt 0 is aangesloten op pin PD2 van de ATmega32. Het signaal is laag als bit 2 van het ingangsregister van poort D laag is.

# 14

# Arrays

| | |
|---|---|
| Doelstelling | Je leert in dit hoofdstuk wat een array is, waarvoor je een array gebruikt en hoe je in C een array toepast. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De declaratie van arrays.<br>▪ Toewijzen aan en aanroepen van arrays.<br>▪ Lezen en schrijven buiten het bereik van een array.<br>▪ Meerdimensionale arrays.<br>▪ De declaratie bij meerdimensionale arrays.<br>▪ De toewijzingen bij meerdimensionale arrays.<br>De voorbeelden gebruiken arrays voor het berekenen van de getallen van Fibonacci en het creëren van de driehoek van Pascal:<br>▪ Het berekenen van de getallen van Fibonacci en de Gulden Snede.<br>▪ Het afdrukken van een tweedimensionaal array.<br>▪ Het vullen en afdrukken van een tweedimensionaal array.<br>▪ Het afdrukken van de driehoek van Pascal en de diagonaal met de getallen van Fibonacci. |

In hoofdstuk 3 tot en met 13 zijn op verschillende plaatsen arrays gebruikt. Het begrip array en de begrippen pointer en string hebben veel gemeen en worden vaak door elkaar gebruikt. Toch zijn er ook grote verschillen. Dat is vaak verwarrend. Dit hoofdstuk bespreekt de array. In de volgende twee hoofdstukken komen de pointers en strings aan bod.
Een array is een plek in het geheugen waar een hoeveelheid gelijksoortige gegevens staan. Anders gezegd, een array is een verzameling gegevens van hetzelfde type. Dit kan bijvoorbeeld een verzameling `float`'s of een verzameling `char`'s zijn. Een array is te vergelijken met een ladenkastje met laatjes van dezelfde afmeting waarin dezelfde soort dingen bewaard worden. De laatjes zijn dan genummerd 0, 1, 2 enzovoorts. Met het nummer kan elk laatje worden gevonden.

**Figuur 14.1 : Leonardo di Pisa** heeft deze reeks getallen voor het eerst onderzocht. Leonardo leefde van ongeveer 1175 tot 1250 in Italië en wordt ook wel Fibonacci (zoon van Bonacci) genoemd.

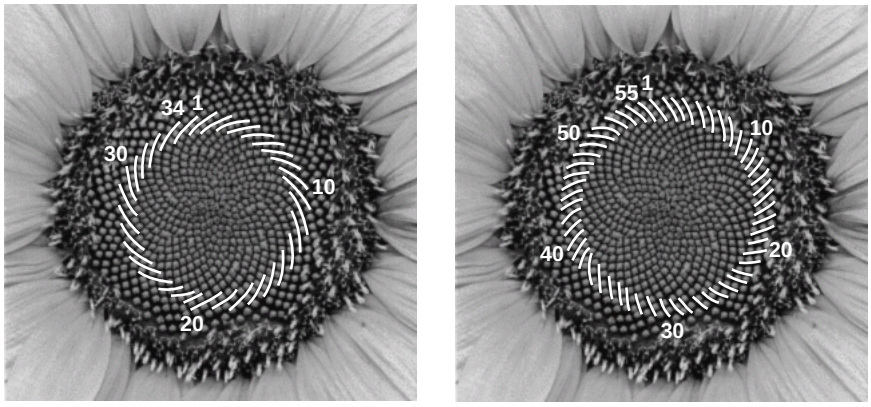
## 14.1 De getallen van Fibonacci en de Gulden Snede

Het programma uit code 14.1 gebruikt een array en onderzoekt de relatie tussen de reeks van Fibonacci en de Gulden Snede. De reeks van Fibonacci is: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, enzovoorts. De volgende waarde uit deze reeks is steeds de som van de twee voorafgaande waarden.

De reeks van Fibonacci heeft een direct verband met de Gulden Snede. Dat is een verhouding tussen twee getallen, die veel in de natuur voor komt en in de kunst gebruikt wordt. De Gulden Snede (*Golden Number*) wordt aangeduid met $\Phi$ (*Phi*) en is gelijk aan:

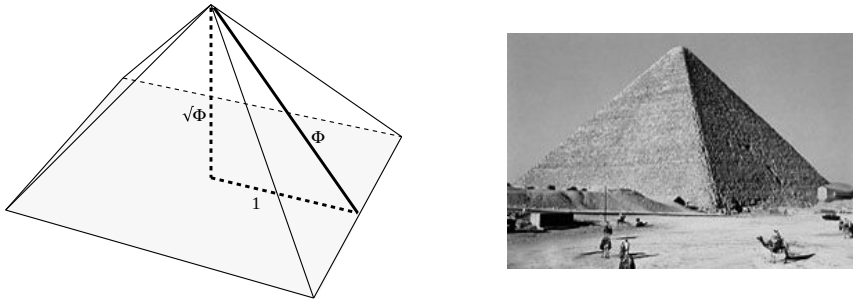
$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1{,}618 \tag{14.1}$$

In figuur 14.2 staat twee keer het hart van dezelfde zonnebloem. In de zonnebloem is een groot aantal spiralen zichtbaar. Een deel van deze spiralen draaien met de klok mee een ander deel draait er tegen in.




**Figuur 14.2** : **Het aantal spiralen bij een zonnebloem is een getal van fibonacci.** In de linker figuur is de serie met 34 spiralen witgestreept en in de rechter figuur de serie met 55 spiralen.

In de linker foto van de zonnebloem is de serie spiralen, die met de klok mee draait, wit gemaakt en in de rechter foto is de serie spiralen, die tegen de klok in draait, wit gemaakt. In de linker foto van de zonnebloem zijn 34 spiralen en in de rechter foto zijn 55 spiralen gemerkt. Dit zijn twee opeenvolgende getallen uit de reeks van Fibonacci.




**Figuur 14.3** : **De Piramide van Cheops.** De piramide is ongeveer 230 m breed en tegenwoordig ongeveer 138,5 m hoog. Volgens sommige onderzoekers is de hoogte 148,5 m geweest. Anderen zeggen dat dit 146 m was. In het eerste geval is de verhouding 1,667 en in het tweede geval 1,612.

Vaak wordt beweerd dat de Gulden Snede is gebruikt door kunstenaars als da Vinci, Seurat en Dali. Deze *ideale* verhouding zou ook voor komen in gebouwen als de Piramide van Cheops, het Parthenon in Athene en de Notre Dame in Parijs. Hiervoor is echter geen enkel bewijs. De afmetingen, waarmee dit bewezen

wordt, lijken willekeurig gekozen of zijn te onnauwkeurig bekend. De getallen van Fibonacci en Φ hebben wel een wetenschappelijke waarde. Zo heeft de Britse wiskundige Penrose deze gebruikt bij een verklaring voor het bestaan van quasi-kristallen. Op zijn minst kan worden opgemerkt dat de getallen van Fibonacci en de Gulden Snede interessante wiskundige fenomenen zijn.

## 14.2 Berekenen getallen van Fibonacci en de Gulden Snede

Het programma uit code 14.1 berekent de eerste eenentwintig getallen uit de reeks van Fibonacci en slaat deze op in een array. Vervolgens wordt de reeks afgedrukt en wordt aangetoond dat de verhouding tussen twee opeenvolgende Fibonacci-getallen de Gulden Snede benadert.

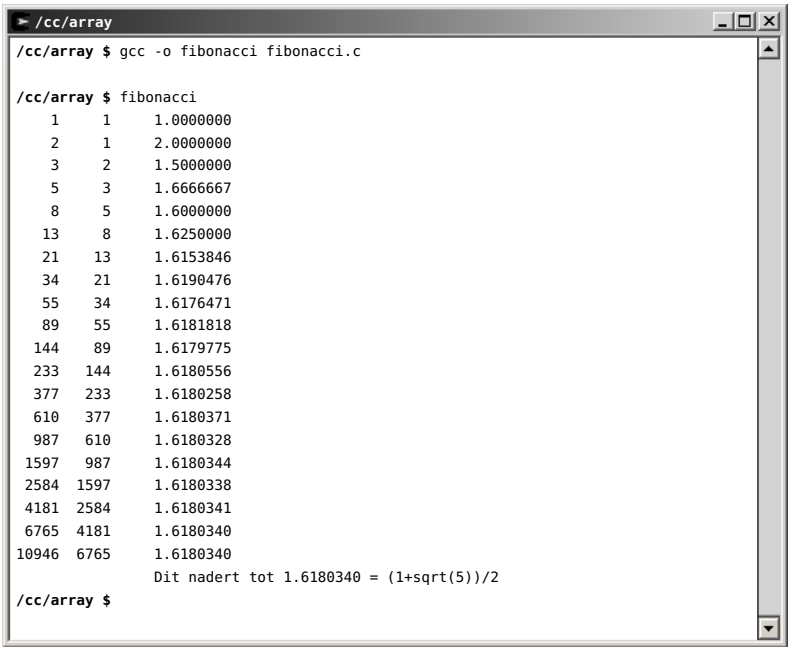
**Code 14.1 : Het verband tussen de getallen van Fibonacci en de Gulden Snede.**

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define MAX_NUMBER 21
5
6  int main(void)
7  {
8    int farray[MAX_NUMBER];
9    int i;
10
11   farray[0] = 1;
12   farray[1] = 1;
13   for (i=2; i<MAX_NUMBER; i++) {
14     farray[i] = farray[i-1] + farray[i-2];
15   }
16
17   for (i=1; i<MAX_NUMBER; i++) {
18     printf("%5d %5d %13.7f\n", farray[i], farray[i-1],
19                                (float) farray[i]/farray[i-1]);
20   }
21
22   printf("\t\tDit nadert tot %.7f = (1+sqrt(5))/2\n", (1+sqrt(5))/2);
23
24   return 0;
25 }
```

In code 14.1 wordt een array met de naam `farray` gebruikt voor getallen van Fibonacci. Dit array is op regel 8 gedeclareerd. De code van regel 11 tot en met regel 14 vult dit array met de getallen van Fibonacci.

De regels 17 tot en met 20 drukken steeds een getal van Fibonacci, het voorafgaande getal van Fibonacci en de verhouding tussen deze twee getallen af. Regel 22 berekent met vergelijking 14.1 Φ en drukt dit getal af.

De uitvoer van het programma staat in figuur 14.4 en laat zien dat de verhouding tussen twee opeenvolgende getallen van Fibonacci bij grote waarden de Gulden Snede steeds dichter nadert.




**Figuur 14.4 : De uitvoer van code 14.1 toont aan dat de verhouding tussen twee opeenvolgende getallen van Fibonacci nadert tot Φ.**

## 14.3 Declaraties van arrays

De volgende declaraties definiëren een **int**, een array van **int**'s, twee losse **char**'s, een array van zes **char**'s en een array van drie **char**'s.

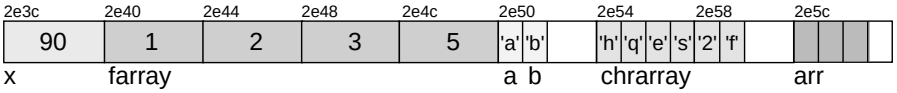
```
int  x = 90;
int  farray[4] = {1, 2, 3, 5};
char a = 'a';
char b = 'b';
char chrarray[] = {'h','q','e','s','2','f'};
char arr[3];
```

Een array wordt gedefinieerd door twee rechte haken achter de naam van de variabele. Het getal tussen `[]` is het aantal elementen waarvoor er in het geheugen plaats gereserveerd wordt. De grootte van deze geheugenruimte hangt af van het type dat gebruikt wordt. Bij een 32-bits machine zal er voor een **int** vier bytes worden gereserveerd en voor een char een byte.

Tussen de accolades staan de waarden die worden toegekend. Als er bij de declaratie waarden worden toegekend, dan mag het getal tussen de rechte haken worden weggelaten. Er wordt dan precies genoeg plaats gereserveerd voor de elementen die er tussen de accolades staat.

Staan er bij de declaratie geen initiële waarden, dan moet het aantal elementen tussen de rechte haken staan. De inhoud van een niet geïnitialiseerd array is ongedefinieerd.

Figuur 14.5 toont hoe de bovenstaande declaraties in het geheugen *kunnen* staan. In de praktijk kan dat er heel anders uitzien. Dit hangt onder andere af van de geheugenorganisatie van de processor en hoe de compiler daarmee omgaat.



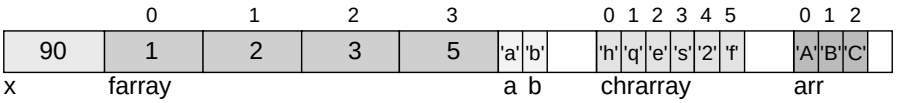

**Figuur 14.5** : **De geheugenorganisatie bij de declaratie van arrays.** De **int** variabele x neemt vier bytes in beslag. De array farray heeft 24 bytes nodig. De variabelen a en b zijn elk een byte groot. Voor chrarray zijn zes bytes nodig en voor arr worden drie bytes gereserveerd.

## 14.4 Toewijzingen bij arrays

De inhoud van de array kan worden gewijzigd door expliciet de inhoud van een enkele cel aan te passen. De elementen van een array worden altijd genummerd vanaf 0. Het eerste element heeft een index 0, het tweede element heeft de index 1 enzovoorts. Array arr kan dan op deze manier gevuld worden:

```
arr[0] = 'A';
arr[1] = 'B';
arr[2] = 'C';
```

Na deze toewijzingen zijn de geheugenplaatsen van arr gevuld met de letters 'A', 'B' en 'C'. Figuur 14.6 laat dit zien en toont de indices van de drie arrays.




**Figuur 14.6** : **De nummering van de array-elementen.** De elementen van array arr hebben nu de waarden 'A', 'B' en 'C'.

De waarden van elementen uit arrays worden opgevraagd door de naam op te geven met daar achter tussen rechte haken de betreffende index:

```
x = farray[2];               // x get value 3
b = chrarray[4];             // b become '2'
printf("%d\n", farray[0]);   // print 1
printf("%c\n", chrarray[1]); // print character 'q'
```

De index mag natuurlijk ook een variabele zijn. Onderstaande code vult de array arr eveneens met de letters 'A', 'B' en 'C':

```
int i;
for (i=0; i<3; i++) {
  arr[i] = 'A' + i;
}
```

Als i nul is krijgt arr[0] de waarde 'A'. Als i een is, wordt bij 'A' (ASCII-waarde 97) er een opgeteld en krijgt arr[1] de waarde 'B' (ASCII-waarde 98). Op dezelfde manier wordt, als i twee is, arr[2] gelijk aan 'C'.
Deze code drukt het derde en het tweede element van chrarray af:

```
x = farray[2];                       // x get value 3
printf("%c\n", chrarray[x]);         // print character 's'
printf("%c\n", chrarray[farray[1]]); // print character 'e'
```

## 14.5 Lezen buiten het bereik van een array

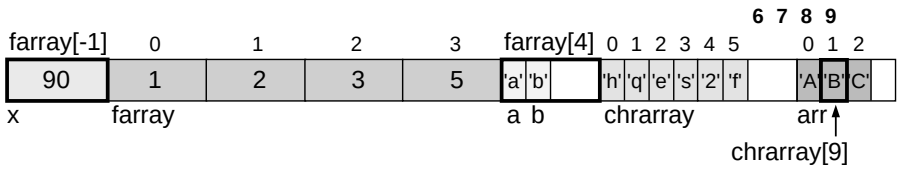
De hoeveelheid geheugen, die nodig is voor een array, wordt bij de declaratie vastgelegd. De taal C houdt op geen enkele manier bij of er ook netjes binnen

dit deel van het geheugen gewerkt wordt. Er kan daardoor op geheugenplaatsen buiten de array worden gelezen.

```
printf("%d\n", chrarray[9]);             // print character 'B'
printf("%d\n", farray[-1]);              // print value 90
printf("%#x\n", farray[4] & 0xffff0000); // print 0x61620000
```

Figuur 14.7 laat zien dat de eerste regel vier posities voorbij het laatste element van `chrarray` kijkt. Toevallig is dat een element uit de array `arr`. Dit element (karakter `'B'`) wordt afgedrukt.

Programma's die bewust gebruik maken van het lezen of het schrijven buiten een array kunnen nooit erg betrouwbaar zijn. De compiler kent de geheugenlocaties toe. De programmeur heeft daar geen invloed op. De geheugentoewijzing in figuren 14.5 tot en met 14.8 is geheel fictief. De figuren geven alleen aan hoe het er uit zou kunnen zien. Waarden lezen en schrijven buiten het bereik van een array, levert in het algemeen onzin op. De programmeur moet er op letten dat de index niet buiten het bereik valt.




**Figuur 14.7** : **Lezen buiten het bereik van een array.** Met een vette rechthoek zijn de elementen aangegeven, die in bovenstaand voorbeeld worden afgedrukt.

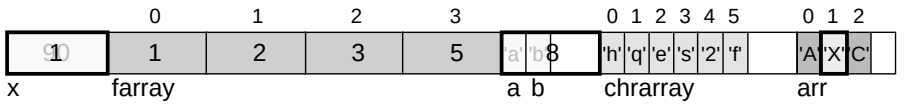
De tweede regel drukt een element af dat precies voor de beginpositie van `farray` staat. In dit geval is dat de variabele `x`. De derde regel drukt de inhoud van een cel ter grootte van een **`int`** af een positie na de array. In dit geval zijn dat de **`char`**'s `a` en `b` en wat rommel uit de rest van deze geheugencel. Deze rommel wordt gemaskeerd met `& 0xffff0000`. De hexadecimale ASCII-waarden 61 en 62 van de karakters `'a'` en `'b'` worden afgedrukt aangevuld met vier nullen.

## 14.6 Schrijven buiten het bereik van een array

Nog erger dan het lezen van een positie buiten een array is het schrijven naar een positie buiten de array. De hoeveelheid geheugen, die nodig is voor een array, is bij de declaratie vastgelegd. De taal C houdt op geen enkele manier bij of er ook netjes binnen dit deel van het geheugen gewerkt wordt. Er kan dus ook op geheugenplaatsen buiten de array worden geschreven.

```
chrarray[9] = 'X';      // overwrite arr[1]
farray[4]   = 8;        // overwrite a and b
farray[-1]  = 1;        // overwrite x
```

De eerste regel overschrijft het tweede karakter van array `arr`. De tweede regel overschrijft de variabelen `a` en `b`. De derde regel overschrijft de variabelen `x`.




**Figuur 14.8** : **Schrijven buiten het bereik van een array.** Met een vette rechthoek zijn de elementen aangegeven, die in bovenstaand voorbeeld worden overschreven.

## 14.7 Meerdimensionale arrays

C kent ook meerdimensionale arrays. Een voorbeeld van een tweedimensionaal array, dat veel gebruikt wordt, is een array van strings. Een string is een array van karakters. Een array van strings is daarom een tweedimensionaal array van **`char`**. Tweedimensionale arrays zijn vooral handig bij matrixberekeningen.

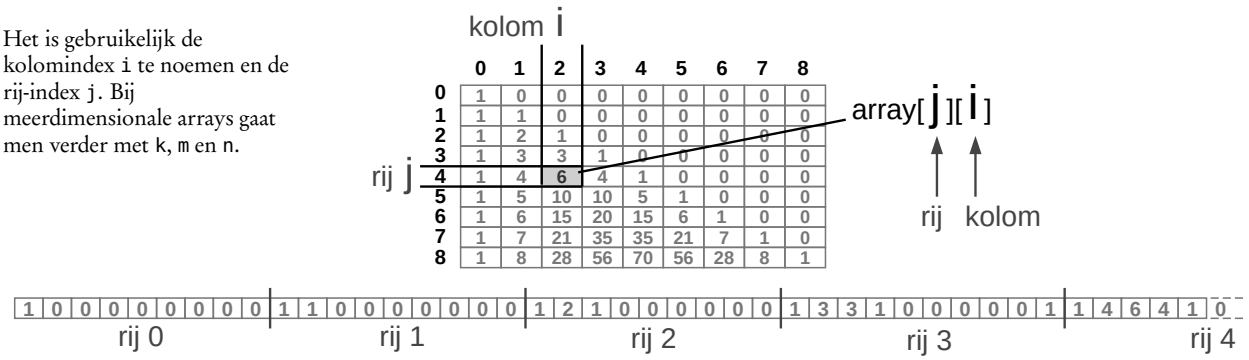
## 14.8 De declaratie van een multidimensionaal array

De nummering van dimensies is van rechts naar links. Bij een tweedimensionaal array geeft de rechter index het kolomnummer en de linker index het rijnummer. Hieronder staan een aantal declaraties van arrays. De variabele `z` is een tweedimensionaal array met twee rijen en drie kolommen. Bij de declaratie mogen initiële waarden staan. De variabelen `cub` en `hypercub` zijn respectievelijk drie- en vier dimensionale arrays. De andere variabelen zijn tweedimensionaal.

```
int   array[9][9];
int   z[2][3] = { {4, 5, 6}, {3, 2, 1} };
char  c[5][6];
float f[2][2] = { {3.14, 1.41}, {0.00, 2.72} };
int   cubic[5][5][5];
int   hypercub[8][7][6][5];
```

In het geheugen staan de elementen van een tweedimensionaal array rij voor rij naast elkaar, zie figuur 14.9. Het gevolg is dat twee elementen met een zelfde rijnummer en een opeenvolgend kolomnummer naast elkaar staan. Daarentegen staan twee elementen met eenzelfde kolomnummer en een opeenvolgend rijnummer juist niet naast elkaar.

Het is gebruikelijk de kolomindex `i` te noemen en de rij-index `j`. Bij meerdimensionale arrays gaat men verder met `k`, `m` en `n`.




**Figuur 14.9** : **De indices bij tweedimensionaal array.** Bovenaan staat de indeling van de array zoals wij er naar kijken. Het is tweedimensionale matrix met negen kolommen en negen rijen. Met de twee indices `i` en `j` kan elk hokje worden gevonden. Onderaan staat de indeling van de array zoals deze in het geheugen staat.

## 14.9 Toewijzingen bij een multidimensionaal array

De toewijzing aan een element van een meerdimensionale array gaat op dezelfde manier als bij eendimensionaal array:

Meerdimensionale arrays zijn altijd lastig. Zorg dat de methode van indexeren altijd consequent is.

```
z[0][0] = 9;
c[4][5] = 'a';
f[1][0] = 1.62;
```

Het afdrukken van een array-element of het toekennen van een array-element aan een variabele of aan een ander array-element gaat op een gelijke wijze:

```
x = z[1][2];
m = c[0][0];
c[2][0] = c[0][2];
printf("%d %f\n", z[0][0], f[1][0]);
```

De kolomindex staat bij het aanroepen van een array rechts: `cubic[k][j][i]`.

In code 14.2 wordt een variabele `i` als index voor de kolommen gebruikt en een variabele `j` als index voor de rijen. Vanwege de declaratie van `z` kan `j` 0 of 1 zijn en kan `i` 0, 1 of 2 zijn. Het resultaat van code 14.2 staat in figuur 14.10.

```
/cc/array
/cc/array $ gcc -o multiarray1 multiarray1.c

/cc/array $ multiarray1
4 5 6
3 2 1

/cc/array $
```

**Figuur 14.10 : De uitvoer van code 14.2.**

**Code 14.2 : Het afdrukken van een tweedimensionaal array.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    int i, j;
 6    int z[2][3] = { {4, 5, 6}, {3, 2, 1} };
 7
 8    for (j=0; j<2; j++) {
 9      for (i=0; i<3; i++) {
10        printf("%d ", z[j][i]);
11      }
12      printf("\n");
13    }
14
15    return 0;
16  }
```

**Code 14.3 : Het vullen en afdrukken van een array.**

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    int   i, j;
 6    char  x;
 7    char  c[5][6];
 8
 9    x = 32;
10    for (j=0; j<5; j++) {
11      for (i=0; i<6; i++) {
12        c[j][i] = x + i;
13      }
14      x += 16;
15    }
16
17    for (j=0; j<5; j++) {
18      for (i=0; i<6; i++) {
19        printf("%c ", c[j][i]);
20      }
21      printf("\n");
22    }
23
24    return 0;
25  }
```

In code 14.3 wordt een character array `c` gevuld met waarden. De variabele `i` is weer — zoals gebruikelijk — de index voor de kolommen en `j` de index voor de rijen. Uit de declaratie van `c` blijkt dat `j` 0 tot en met 4 en `i` 0 tot en met 5 kan zijn. De uitvoer van dit programma staat in figuur 14.11

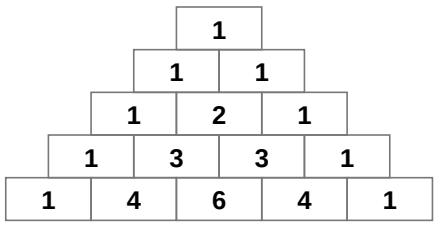
```
/cc/array
/cc/array $ gcc -o multiarray2 multiarray2.c

/cc/array $ multiarray1
  ! " # $ %
0 1 2 3 4 5
@ A B C D E
P Q R S T U
‘ a b c d e

/cc/array $
```

**Figuur 14.11 : De uitvoer van code 14.3.**

## 14.10 De driehoek van Pascal

De driehoek van Pascal is opgesteld door Blaise Pascal en geeft de factoren van het binomium van Newton. Dit binomium zegt dat de macht van de som van twee grootheden in een som van termen met de afzonderlijke machten kan worden geschreven. De factoren bij het ontbinden van de macht van de som van twee getallen $(a+b)^n$ worden gegeven door de driehoek van Pascal. Het binomium van Newton — en daarmee ook de driehoek — speelt een belangrijke rol in de kansberekening. Figuur 14.13 toont voor 0 tot en met 4 de macht van de som van twee getallen en het bijbehorende deel van de driehoek van Pascal.

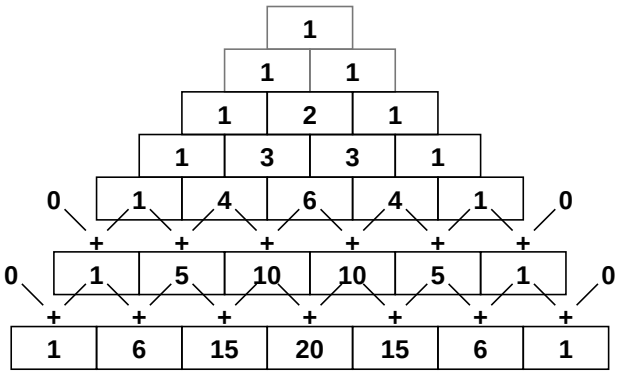
$$
\begin{aligned}
(a+b)^0 &= 1 \\
(a+b)^1 &= a+b \\
(a+b)^2 &= a^2+2ab+b^2 \\
(a+b)^3 &= a^3+3a^2b+3ab^2+b^3 \\
(a+b)^4 &= a^4+4a^3b+6a2b^2+4ab^3+b^4
\end{aligned}
$$




**Figuur 14.13 : Het binomium van Newton en de driehoek van Pascal.** De driehoek van Pascal bevat de coëfficiënten die voor de machten staan in het binomium van Newton.

**Figuur 14.12 : Blaise Pascal.** Deze Franse geleerde leefde van 1623 tot 1662 en heeft veel bijgedragen aan de wiskunde en de natuurkunde. De SI-eenheid voor druk (Pa = $Nm^{-2}$) is naar hem genoemd. Pascal hield zich ook bezig met het bedenken van rekenapparaten. De programmeertaal Pascal is naar hem genoemd.

De driehoek van Pascal wordt samengesteld door steeds de som van de twee bovenliggende elementen uit de driehoek te nemen, zie ook figuur 14.14. De linker en rechter cellen van elke regel zijn altijd 1. Toch is dit ook steeds de som van de twee bovenliggende elementen. Het is namelijk de som van een element met de waarde 0 en een element met de waarde 1.

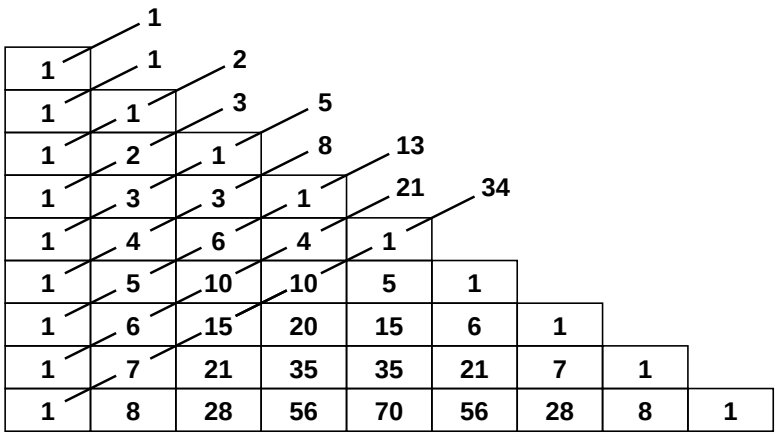



**Figuur 14.14 : De opbouw van de driehoek van Pascal.** Een getal uit de driehoek is de som van de twee bovenliggende getallen.

Er zijn meerdere manieren om de driehoek van Pascal op te schrijven. In figuur 14.15 zijn alle elementen links uitgelijnd. De som van de getallen op de diagonalen levert dan de reeks van Fibonacci op.

## 14.11 Berekening driehoek van Pascal en getallen van Fibonacci

Het programma van code 14.4 berekent de eerste negen rijen van de driehoek van Pascal en berekent voor elke regel de waarde langs de diagonaal. Het programma drukt de driehoek af en geeft voor elke regel de waarde van de bijbehorende diagonaal.




**Figuur 14.15 : De driehoek van Pascal en de getallen van Fibonacci.** Als de driehoek van Pascal links uitgelijnd is, zijn de sommen op de diagonalen getallen van Fibonacci.

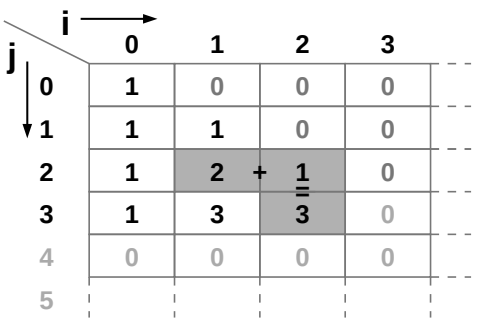
De getallen van de driehoek worden in een tweedimensionaal array `triangle` bewaard. In dit geval heeft dit array evenveel rijen als kolommen. Omdat het over een driehoek gaat, worden niet alle hokjes uit de array gebruikt. Voor de som van de diagonalen is een eendimensionaal array `diagonal` nodig. De afmetingen van triangle en de afmeting van `diagonal` zijn vastgelegd met de constante `DIMENSION`. In dit voorbeeld is deze constante negen.

**Uitleg code 14.4 regel 12-16**

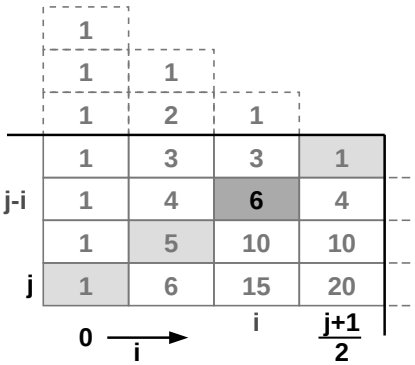
Het programma bestaat uit vier delen. Omdat bij het vullen van een rij de getallen uit de vorige rij gebruikt worden, wordt eerst de array `triangle` met nullen gevuld. Bij de declaratie wordt een array immers niet automatisch geïnitialiseerd. Er zou dus van alles in het geheugen kunnen staan.

**Regel 19-24**

De getallen worden links uitgelijnd in de array geplaatst. Rij voor rij worden de getallen uitgerekend. Het eerste getal van een rij is altijd 1 (`triangle[j][0]=1;`).




**Figuur 14.16 : De berekening van de driehoek van Pascal.** Element `triangle[j][i]` is de som van `triangle[j-1][i-1]` en `triangle[j-1][i]`. In dit voorbeeld is `triangle[3][2]` gelijk aan `triangle[2][1]` + `triangle[2][2]`




**Figuur 14.17 : De berekening van een diagonaal in de driehoek van Pascal.**

Daarna moeten er per rij nog eens `j` getallen worden uitgerekend. Het aantal getallen per rij hangt af van het rijnummer. De rest van de getallen blijft nul. Figuur 14.16 laat zien dat het getal `triangle[j][i]` gelijk is aan de som van de getallen `triangle[j-1][i-1]` en `triangle[j-1][i]` uit de bovenliggende rij `j-1`.

**Code 14.4 : De driehoek van Pascal en de getallen van Fibonacci.**

```
1  #include <stdio.h>
2
3  #define DIMENSION 9
4
5  int main(void)
6  {
7    int triangle[DIMENSION][DIMENSION];
8    int diagonals[DIMENSION];
9    int i,j;
10
11   // Fill array with zero's
12   for (j=0; j<DIMENSION; j++) {
13     for (i=0; i<DIMENSION; i++) {
14       triangle[j][i] = 0;
15     }
16   }
17
18   // Calculate values triangle of Pascal
19   for (j=0; j<DIMENSION; j++) {
20     triangle[j][0] = 1;
21     for (i=1; i<=j; i++) {
22       triangle[j][i] = triangle[j-1][i-1] + triangle[j-1][i];
23     }
24   }
25
26   // Calculate the sum along the diagonals (Fibonacci)
27   for (j=0; j<DIMENSION; j++) {
28     diagonals[j] = 1;
29     for (i=1; i<=(j+1)/2; i++) {
30       diagonals[j] += triangle[j-i][i];
31     }
32   }
33
34   // Print the sum along the diagonals and print the triangle of Pascal
35   for (j=0; j<DIMENSION; j++) {
36     printf("%5d |", diagonals[j]);
37     for (i=0; i<=j; i++) {
38       printf("%3d", triangle[j][i]);
39     }
40     printf("\n");
41   }
42
43   return 0;
44 }
```

**Uitleg code 14.4 regel 27-32**

Nadat de driehoek van Pascal is gevuld, worden de sommen van de diagonalen bepaald. Voor het berekenen van de som van een bepaalde rij `j` moeten de elementen van linksonder naar rechtsboven worden opgeteld, zoals in figuur 14.17 is weergegeven. De index van de rij moet steeds met één verlaagd worden en de index van de kolom met één verhoogd.

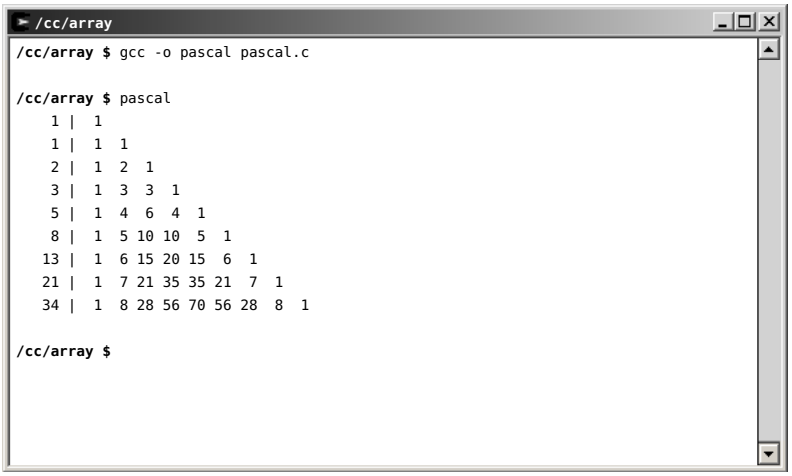
Bij kolom `i` is de index `j` van de rij `i` lager, dus: `j-i`. Het eerste element is altijd 1. De som van de diagonaal voor rij `j` is zodoende:

```
1 + triangle[j-1][1] + triangle[j-2][2] + ···  + triangle[j-i][i] + ···
```

Het herhaald optellen kan stoppen als `i` gelijk is aan `(j+1)/2`.

**Regel 35-41**

Tenslotte wordt per rij de som van de diagonalen en de betreffende rij van de driehoek afgedrukt met het resultaat van figuur 14.18




**Figuur 14.18 : De uitvoer van code 14.4 geeft de driehoek van Pascal en geeft voor elke regel de som op de diagonaal.**

# 15

# Pointers

| | |
|---|---|
| Doelstelling | Je leert in dit hoofdstuk wat een pointer is, hoe je een pointer toepast in C en waar je pointers voor kunt gebruiken. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De declaratie van een pointer.<br>▪ De toewijzing aan een pointer.<br>▪ Het overnemen van de inhoud van de plaats waar de pointer naar wijst.<br>▪ De adresoperator (&) en de dereferentie-operator (*).<br>▪ Rekenen met pointers.<br>▪ Fouten bij pointers.<br>▪ Toepassingen met pointers.<br><br>Voorbeelden met pointers zijn:<br>▪ Berekenen van de getallen van Fibonacci en de Gulden Snede.<br>▪ De functies `strcpy` en `strcmp`. |

In eerdere hoofdstukken zijn op verschillende plaatsen al pointers gebruikt. Het begrip pointer en de begrippen arrays en strings hebben veel gemeen en worden vaak door elkaar gebruikt. Arrays zijn besproken in hoofdstuk 14 en strings komen aanbod in hoofdstuk 16.

Programmeurs, die C niet goed kennen, vinden pointers vaak lastig. Het mechanisme van pointers en pointerbewerkingen is heel krachtig en kenmerkend voor C. Juist bij microcontrollers zijn pointers essentieel. De adressering van de registers is hierop gebaseerd.

Pointers zijn al eerder aan de orde gekomen bij de uitleg over *call by reference* in paragraaf 4.3, bij de uitleg van de functie `scanf` in paragraaf 5.1 over de geformatteerde in- en uitvoer en bij de bespreking van de in- en uitvoer van de microcontroller in paragraaf 11. De dereferentie-operator `*` en de adresoperator `&` zijn daar al besproken. Een pointer is een verwijzing naar een geheugenlocatie en de waarde van een pointer is dus altijd een geheugenadres.

Deze paragraaf behandelt het begrip pointers uitgebreider. Er wordt aandacht besteed aan het rekenen met pointers en voorbeeld 14.1 wordt herschreven met pointers in plaats van arrays.

## 15.1 Declaraties van pointers

Een pointer wordt gedeclareerd door een asterisk (*) voor het type te zetten. Onderstaande declaraties definiëren achtereenvolgens een pointer naar een **int**, een pointer naar een **char**, twee pointers naar een **char** en een pointer naar een **float**.

```
int   *x;
char  *c;
char  *s1, *s2;
float *f;
```

In het geheugen wordt alleen een stukje geheugenruimte gereserveerd waar een adres kan worden bewaard. Afhankelijk van het systeem is dat bijvoorbeeld een 32-bits getal. De compiler moet bij een toekenning altijd weten wat het type van de gegevens is waar de pointer naar wijst. Vandaar dat bij een pointerdeclaratie altijd een type staat. Alleen bij een **void**-declaratie is het type waar de pointer naar wijst nog onbekend.

```
void  *v;
```

Lege pointerdeclaraties worden gebruikt bij functies. De functie `malloc`, waarmee geheugenruimte gereserveerd kan worden, maakt hier gebruik van. Het prototype van deze functie, die bij code 15.1 verder aan de orde komt, luidt:

```
void *malloc(long unsigned int Nbytes);
```

Bij de aanroep wordt met een cast-operator aangegeven dat de pointer naar een geheugenplek met **int**'s wijst:

```
p = (int *) malloc(200*sizeof(int));
```

## 15.2 Toewijzingen met pointers

Een pointer bevat het adres van een geheugenplaats. In onderstaande code wordt aan de pointer `ptr` het adres van **int** `x` toegekend. Dit wordt aangeven met het ampersand-teken `&`. De uitdrukking `&x` betekent letterlijk het adres van `x`. Het `&`-teken wordt in dit verband de adresoperator genoemd.

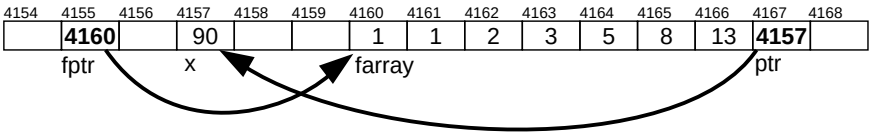
In C wordt & gebruikt als bitwise and-operator `z=a&b`, als logische and-operator `c&&d` en als adresoperator `ptr=&x`.

```
int *ptr;
int  x = 90;
int  farray = {1, 1, 2, 3, 5, 8, 13};
int *fptr;

ptr  = &x;
fptr = farray;
```

De array `farray` kan worden beschouwd als een pointer naar de geheugenplaats waar de elementen van de array staan. De variabele `farray` is het adres van de eerste `1` uit de array. Daarom staat er bij de toekenning aan `fptr` voor `farray` geen ampersand.
Een alternatieve schrijfwijze voor de toekenning aan `fptr` is:

```
fptr = &farray[0];
```

Hier wordt aan `fptr` het adres `&` van het eerste element van de array `farray[0]` toegekend.
Figuur 15.1 illustreert de toewijzingen aan `ptr` en aan `fptr`. Het geheugengebruik en de adressering in deze figuur is fictief. In werkelijkheid kan de compiler een heel andere indeling maken.




**Figuur 15.1** : **Toewijzing en adressering bij pointers.** De toewijzing `ptr=&x` kent aan `ptr` het adres van `x` toe. In dit geval is dat adres 4157. De toewijzing `fptr=array` kent aan `fptr` het adres van `array` toe. In dit geval is dat adres 4160.

De inhoud van de geheugenplek, waar de pointer naar wijst, wordt verkregen door een asterisk voor de pointer te zetten. De asterisk — het ∗-teken — heet in deze context de dereferentie-operator. In onderstaand voorbeeld wordt allereerst de inhoud waar `ptr` en `fptr` naar wijzen afgedrukt. Daarna wordt aan `x` de inhoud waar `fptr` naar wijst toegekend en afgedrukt. Tenslotte wordt 90 toegekend aan de inhoud waar `ptr` naar wijst.

In C wordt ∗ gebruikt als vermenigvuldingsoperator `z=a*b`, als pointerdeclaratie **`char`** `*s` en als operator om de inhoud van een pointer te benaderen `*ptr=90`.

De dereferentie-operator heet in het Egels *dereference operator*. In het Nederlands wordt dit soms vertaald met *dereference-* of indirectie-operator.

```
printf("%d %d\n", *ptr, *fptr);  // print 90 and 1
x = *fptr;
printf("%d\n", x);               // print 1
*ptr = 90;
printf("%d\n", x);               // print 90
```

## 15.3 Rekenen met pointers

Pointers zijn gewone getallen waar mee gerekend kan worden. Bewerkingen als vermenigvuldigingen of delen van pointers zijn weinig zinvol, maar het optellen en aftrekken van een constante bij een pointer of het aftrekken van twee pointers is wel zinvol. Dit optellen en aftrekken van pointers wordt pointer rekenen (*pointer arithmetic*) genoemd. Hier staan een aantal voorbeelden

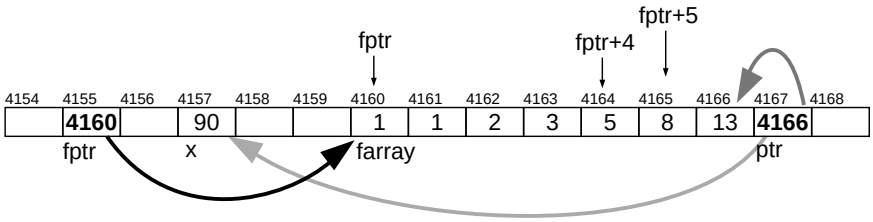
```
printf("%d\n", *(fptr+4));   // print 5
printf("%d\n", *(fptr+5));   // print 8
printf("%d\n", *fptr+5);     // print 6, the value of *fptr, which is 1, raised with 5
ptr = fptr+6;
printf("%d\n", *ptr);        // print 13, is the last item of farray
printf("%d\n", fptr-ptr);    // print 6, the number of items minus 1 of farray
```

In de eerste regel wijst `fptr+4` naar het vijfde element uit `farray`, zie figuur 15.2. De inhoud (∗) van dit element is 5 en daarom wordt er 5 afgedrukt. Het verschil tussen de tweede en derde regel is dat de tweede regel de inhoud van `fptr+5` afdrukt en dat de derde regel vijf bij de inhoud van `*fptr` optelt. Pointer `ptr` wijst zes elementen verder dan `fptr` en wijst naar het laatste element van `farray`. De inhoud waar `ptr` nu naar wijst, is 13. En het verschil tussen de twee pointers `ptr` en `fptr` is zes.

Stel dat deze bewerkingen na elkaar worden uitgevoerd:

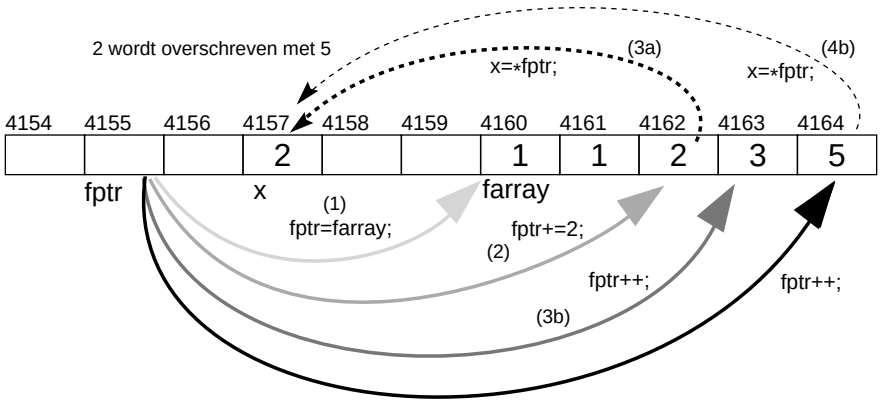
```
fptr  = farray;
fptr += 2;
x     = *fptr++;
x     = *++fptr;
```

Figuur 15.3 toont het effect. Eerst wijst `fptr` naar het begin van `farray`, dan wordt er twee bijgeteld en wijst `fptr` naar het derde element van `farray`. De derde regel




**Figuur 15.2** : **Rekenen met pointers.** De pointer `fptr+4` wijst naar het vijfde element van `farray`. In dit geval is dat adres 4164. De pointer `fptr+5` wijst naar het zesde element van `farray`. In dit geval is dat adres 4165. De pointer `ptr` wijst naar het laatste element van `farray`. In dit geval is dat adres 4166.

bestaat uit twee bewerkingen: eerst (`x=*fptr`) wordt de inhoud waar de pointer naar wijst aan `x` toegekend en daarna wordt er een bij opgeteld (`fptr++`). De pointer wijst na de derde bewerking dus naar het vierde element van `farray` en `x` heeft de waarde 2 gekregen. De laatste regel doet deze bewerkingen in omgekeerde volgorde: eerst schuift de pointer een positie op en wordt de inhoud waar deze naar wijst aan `x` toegekend. Na deze bewerking heeft `x` de waarde `5` en wijst de `fptr` naar het vijfde element.




**Figuur 15.3** : **Nog meer rekenen met pointers.** Pointer `fptr` wijst (a) naar `farray`; `fptr` schuift twee positie op (b); `x` krijgt waarde twee (3a) en `fptr` schuift een positie (3b) op en tenslotte schuift `fptr` nog een positie op (4a) en zal `x` vijf worden (4b).

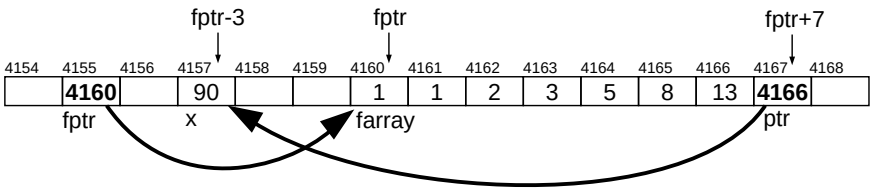
## 15.4 Fouten met pointers

Bij pointers is het nog eenvoudiger dan bij arrays om op een verkeerde plaats iets uit het geheugen te lezen of naar het geheugen te schrijven. De programmeur moet bij het gebruik van pointers precies weten wat er gebeurt, anders is de kans groot dat deze iets maakt dat vroeg of laat vastloopt.

```
ptr  = &x;
fptr = farray;
printf("%d\n", *(fptr-3));     // print 90
printf("%d\n", *(fptr+7));     // print 4166
```

In bovenstaand voorbeeld wijst pointer `ptr` weer naar het adres van variabele `x`. Doordat `fptr-3` toevallig naar `x` wijst, wordt de waarde van `x` afgedrukt. Omdat

`fptr+7` toevallig naar `ptr` wijst, wordt de inhoud van `ptr` afgedrukt en is dit toevallig het adres van `x`. Figuur 15.4 laat zien dat met pointerrekenen het eenvoudig is buiten de array te lezen. Dat kan nooit de bedoeling zijn. Het programma wordt afhankelijk van toevalligheden bij het compileren. Als de compiler een iets andere geheugenindeling kiest, klopt het niet meer.

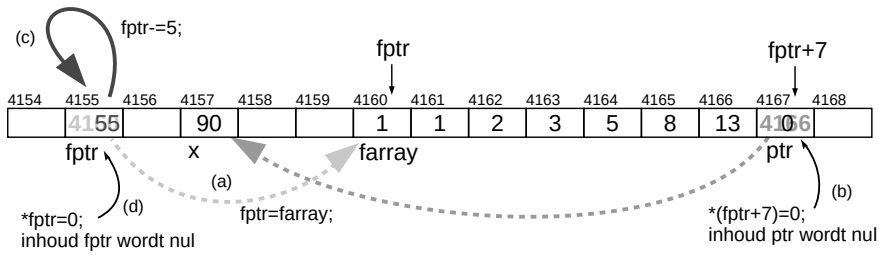



**Figuur 15.4** : **Leesfouten met pointers.** De pointer `fptr-3` wijst naar `x` en de pointer `fptr+7` wijst naar `ptr`.

Nog erger wordt het als de inhoud waar de pointers naar wijzen wordt veranderd. Figuur 15.5 laat zien dat deze code allerlei essentiële geheugenplaatsen overschrijft:

```
fptr       = farray;
*(fptr+7)  = 0;
fptr      -= 5;
*fptr      = 0;
```

De tweede regel overschrijft de inhoud van `ptr`. Deze pointer wijst nu nergens naar. De derde regel schuift de pointer vijf posities naar links. Toevallig is dat de pointer zelf. De laatste maakt ook de inhoud van `fptr` nul. De bovenstaande code overschrijft dus de pointers `ptr` en `fptr`. Als dit de enige mogelijkheid is om bij deze gegevens te komen, zijn de gegevens op deze geheugenlocaties definitief verloren. Omdat in dit geval `farray` en `x` als variabelen nog steeds bekend zijn, zijn de gegevens op deze geheugenlocaties wel terug te vinden.




**Figuur 15.5** : **Fouten bij toewijzingen met pointers.** Pointer `fptr` wijst (a) naar `farray`. Het niet bestaande element acht van `farray` wordt nul gemaakt (b). Dit is toevallig pointer `ptr` en wordt overschreven. Pointer schuift vijf posities naar links (c). Toevallig is dat het adres van de pointer zelf. Tenslotte wordt `fptr` ook overschreven (d) met nul.

## 15.5 Berekenen getallen van Fibonacci en Gulden Snede met pointers

Het programma uit code 15.1 onderzoekt opnieuw de relatie tussen de reeks van Fibonacci en de Gulden Snede. Net als het programma van code 14.1 uit para-

graaf 14 berekent het de eerste eenentwintig getallen van de reeks van Fibonacci en slaat deze op in een array. Ook nu wordt de reeks afgedrukt en wordt er weer aangetoond dat het quotiënt tussen twee opeenvolgende Fibonacci-getallen de Gulden Snede benadert. Alleen gebruikt code 15.1 een dynamisch array en worden er pointers gebruikt bij het vullen en bij het afdrukken van de array.

**Code 15.1 : De berekening van de getallen van Fibonacci met behulp van pointers.**

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <math.h>
 4
 5  #define MAX_NUMBER 21
 6
 7  int main(void)
 8  {
 9    int *farray;
10    int *p,*ep;
11
12    if ( (farray = (int *) malloc(MAX_NUMBER*sizeof(int))) == NULL ) {
13      printf("error: not enough memory\n");
14      return 1;
15    }
16    p  = farray;
17    ep = farray+MAX_NUMBER-1;
18
19    *p++ = 1;
20    *p++ = 1;
21    while (p <= ep) {
22      *p = *(p-1) + *(p-2);
23      p++;
24    }
25
26    p = farray+1;
27    while (p <= ep) {
28      printf("%5d %5d %13.7f\n", *p, *(p-1), (float) *p / *(p-1));
29      p++;
30    }
31
32    printf("\t\tDit nadert tot %.7f = (sqrt(5)+1)/2\n", (sqrt(5)+1)/2);
33
34    return 0;
35  }
```

**Uitleg code 15.1 regel 12-15**

In dit voorbeeld is `farray` een pointer. Met de functie `malloc` wordt er een rij van `MAX_NUMBER` integers gealloceerd. Pointer `farray` wijst naar het begin van deze rij. Het argument dat aan `malloc` wordt meegegeven is de grootte van de geheugenruimte die nodig is. In dit geval zijn dat `MAX_NUMBER*sizeof(int)` bytes, namelijk het product van het aantal integers en de grootte van een integer.

**Regel 12**
*dynamic allocation*
`malloc()`
`calloc()`
`realloc()`
`free()`

```
void *malloc(int n);
void *calloc(int n, int s);
void *realloc(void *p, int n);
void free(void* p)
```

Met `malloc` wordt geheugen dynamisch gereserveerd. Dynamisch betekent dat er pas geheugenruimte wordt gereserveerd op het moment dat deze nodig is. In het

Engels heet dit *dynamic memory allocation*. Het argument bevat het aantal bytes dat gereserveerd moet worden. De functie `malloc` geeft een pointer terug naar de gereserveerde geheugenruimte en `NULL` als het reserveren niet gelukt is.

```
int  *p = (int *) malloc(10*sizeof(int));   // allocate 10 int's
char *s = (char *) malloc(64*sizeof(char)); // allocate 64 char's
```

Met `calloc` wordt geheugen gereserveerd voor een array van `n` elementen met een grootte `s`. De functie `malloc` initialiseert het gereserveerde geheugen niet, daarentegen maakt `calloc` het geheugen wel leeg. Met `realloc` kan het gealloceerde geheugen uitgebreid worden. Gereserveerde geheugenruimte, die niet meer gebruikt wordt, kan met de functie `free` worden vrijgegeven.

```
free(p);          // free previous allocated memory space
```

**Regel 12**
**`sizeof()`**

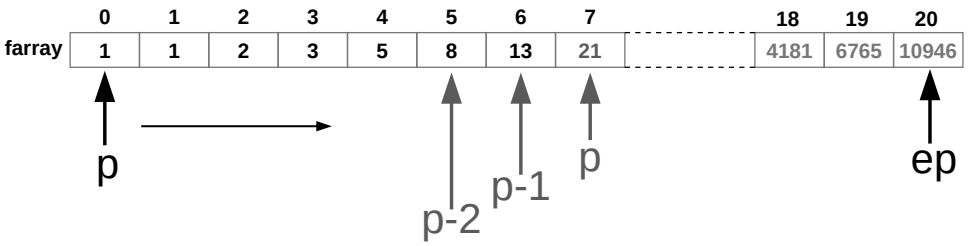
De operator sizeof() geeft de grootte van een bepaald datatype weer:

```
printf("%d\n", sizeof(char));          // print 1
printf("%d\n", sizeof(short));         // print 2
printf("%d\n", sizeof(int));           // print 4
printf("%d\n", sizeof(long));          // print 4
printf("%d\n", sizeof(double));        // print 8
printf("%d\n", sizeof(long double));   // print 12
printf("%d\n", sizeof(FILE));          // print 104
```

Deze operator wordt voornamelijk gebruikt bij reserveren van geheugenruimte om te weten hoeveel plaats een bepaald type inneemt.

**Regel 16-17**

Aan pointer `p` wordt `farray` toegekend. Pointer `p` wijst naar het eerste element van de array. Later wordt `p` gebruikt om langs de array te gaan. Aan pointer `ep` wordt `farray+MAX_NUMBER` toegekend. Pointer `ep` wijst naar het laatste element van `farray`.




**Figuur 15.6 :** **Pointer p schuift langs de array.** Aanvankelijk wijst `p` naar het begin van `farray`. Elke iteratie wordt de som van de twee voorafgaande elementen berekend en aan het element waar `p` naar wijst toegekend. Het schuiven gaat door zolang `p` kleiner of gelijk is aan `ep`.

**Uitleg code 15.1 regel 19-24**

De bewerking `*p++ = 1` op regel 19 bestaat in feite uit twee bewerkingen: `*p=1` en `p++`. Aan de inhoud van het adres waar `p` naar wijst wordt 1 toegekend en daarna wordt er bij `p` 1 opgeteld. De pointer is na deze bewerking een positie opgeschoven. Na de regel 20 hebben de eerste twee elementen van `array` de waarde 1 en wijst `p` naar het derde element.
Vervolgens worden de volgende elementen van de array gevuld met de som van de voorgaande elementen `*p= *(p-1) + *(p-2);` en schuift de pointer weer een positie verder `p++`. Dit wordt gedaan zolang `p` kleiner of gelijk is aan `ep`. Zie ook figuur 15.6.

**Regel 26-32**

Regel 26 tot en met 32 drukken de getallen van Fibonacci af. Pointer `p` wordt daarvoor eerst teruggezet naar het tweede element van de array: `p = farray + 1;`.

Vervolgens wordt er opnieuw langs de array gegaan en wordt steeds de inhoud waar de pointers `p` en `p-1` naar wijzen en het quotiënt van deze waarden afgedrukt.

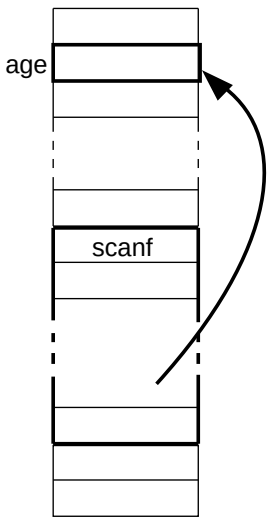
## 15.6 Toepassingen pointers

Het programma van code 15.1 laat het gebruik van een pointer bij een array zien. De pointers nemen daarbij de rol over van de array-indices. Dit suggereert dat een pointer een soort array-index is. Dat is niet zo. Een pointer is veel algemener. Pointers worden gebruikt om allerlei datagegevens te manipuleren. Pointers kom je tegen bij:

- *IO van functies*
  Via de parameterlijst kunnen functies waarden inlezen. Een functie kan alleen met de return data teruggeven. Door aan de parameterlijst een pointer mee te geven, heeft de functie een adres beschikbaar, waar deze de informatie weg kan plaatsen. Een voorbeeld is de standaardfunctie `scanf`, die geïntroduceerd is in paragraaf 5.1

  ```
  printf("What is your age? ");
  scanf("%d", &age);
  ```

  Aan de functie `scanf` wordt het adres `&age` van de variabele `age` meegegeven. Zo weet `scanf` waar het resultaat, de waarde van `age`, moet worden weggeschreven, zie figuur 15.7.

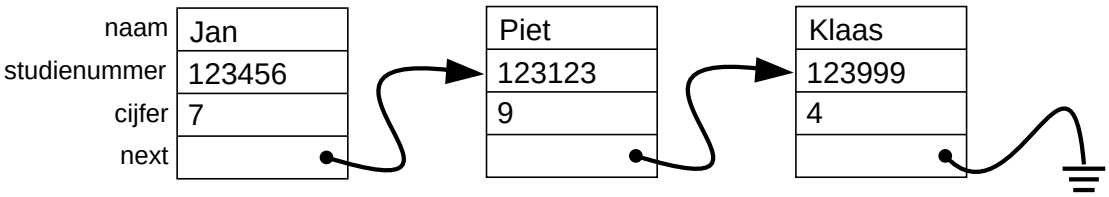



**Figuur 15.7 : Pointers bij functies.** Door aan de functie `scanf` het adres van `age` mee te geven, vult `scanf` het resultaat op de locatie `age` in.

- *Arrays*
  Zoals voorbeeld 15.1 uit dit hoofdstuk laat zien, kan bijna alles wat met array-indices wordt gedaan ook met pointers worden uitgevoerd.
- *Strings*
  Een string is een array van **`char`**'s, die afgesloten wordt met de waarde nul `'\0'`. Bij het manipuleren van strings worden daarom vaak pointers gebruikt. Hoofdstuk 16 behandelt de strings. Paragraaf 15.7 bespreekt het gebruik van pointers bij de stringfuncties `strcpy` en `strcmp`.
- *Datastructuren*
  In C is het mogelijk om met een **`struct`** een eigen type te maken, dat opgebouwd is uit andere typen. Het meegeven en teruggeven van dit soort typen aan functies wordt gedaan met pointers. Een voorbeeld van een dergelijke datastructuur is de filepointer. Als er een file wordt geopend met de functie `fopen` geeft deze een filepointer (`FILE *`) terug die naar een datastructuur wijst met alle informatie van het bestand.
- *Lijsten en bomen*
  Met C kunnen lijsten en bomen worden gemaakt. Een lijst bestaat uit een serie objecten van een bepaalde datastructuur die aan elkaar gekoppeld zijn. Figuur 15.8 geeft een voorbeeld van een lijst met objecten van het type `STUDENT`. Deze datastructuur heeft vier velden: `naam`, `studienummer` en `cijfer` en een pointer.

  ```
  typedef struct student {
    char            naam[64];
    char            studienummer[8];
    int             cijfer
    struct student *next;
  } STUDENT;
  ```

De objecten zijn aan elkaar gekoppeld met de pointer `next`. Deze pointer wijst steeds naar het volgende object. De pointer van het laatste object wijst naar `NULL`. Lijsten en bomen zijn flexibele datastructuren waaraan makkelijk elementen kunnen worden toegevoegd en verwijderd. Paragraaf 17.3 laat zien hoe dat gaat bij een lijst.




**Figuur 15.8 : Voorbeeld van een lijst met datastructuren.**

## 15.7 Voorbeelden met pointers

Vooral bij strings worden pointers veel gebruikt. De functies uit het headerbestand `string.h` zijn allemaal met een paar regels code te beschrijven. Het is interessant om een aantal van deze functies nader te bekijken.
De functie `strcpy` kopieert een string van de ene geheugenplaats naar de andere. Pointer `s` wijst naar de bron (*source*) en `d` wijst naar de bestemming (*destination*). De functie `strcpy` begint vooraan bij beide geheugenplaatsen en kopieert de bron dan karakter voor karakter naar de bestemming. Dit gaat door totdat `s` de *end-of-string* heeft bereikt. Deze *end-of-string* wordt dus niet meer gekopieerd en wordt daarom apart toegevoegd.

```
void strcpy(char *d, char *s)
{
  while (*s != '\0') {         // while not end-of-string
    *d = *s;                   //   copy character
    d++;                       //   move d to next character
    s++;                       //   move s to next character
  }
  *d = '\0';                   // add end-of-string
}
```

De toekenning van `*s` aan `*d` kan al bij de test van de **`while`**-lus worden gedaan. De *end-of-string* wordt in dit geval ook gekopieerd en hoeft niet apart toegevoegd te worden.

```
void strcpy(char *d, char *s)
{
  while ((*d = *s) != '\0') {
    d++;
    s++;
  }
}
```

`*s++` is equivalent met `*(s++)` en niet met `(*s)++`. De associatie van `*` is immers van rechts naar links. Het is dus `s` die verhoogd wordt. Maar de waarde van `*(s++)` is wel de waarde waar de huidige `s` naar wijst. Na de verwerking van deze waarde wordt `s` pas opgehoogd.

Het ophogen van de pointers is in de volgende `strcpy` ook aan de test van de **`while`** toegevoegd. De waarde van `*s++` is het karakter waar `s` naar wijst voordat `s` wordt opgehoogd. Op dezelfde manier wordt deze waarde op de plaats waar `d` naar wijst ingevuld voordat `d` wordt opgehoogd.

```
void strcpy(char *d, char *s)
{
  while ((*d++ = *s++) != '\0') {}          // {} is an empty statement
}
```

Zolang de *end-of-string* niet bereikt is, is de waarde van de toekenning ongelijk aan nul. Er kan dus direct op de waarde van de toekenning worden getest:

De **while**-lus voert een leeg statement {} uit. Dit kan ook met een ; worden beschreven.

```
void strcpy(char *d, char *s)
{
  while (*d++ = *s++) ;                    // ; is an empty statement
}
```

De compiler waarschuwt hierbij dat er beter haakjes om de testwaarde kunnen staan:

```
void strcpy(char *d, char *s)
{
  while ((*d++ = *s++)) ;
}
```

De functie `strcmp` vergelijkt een string met een andere string. De pointers `a` en `b` wijzen naar de twee strings. De functie `strcmp` begint vooraan bij beide strings en geeft nul terug als de *end-of-string* van string `a` is bereikt. De karakters van `a` en `b` zijn dan allemaal gelijk. Als de karakters waar `a` en `b` naar wijzen ongelijk zijn, stopt het zoeken en geeft `strcmp` het verschil tussen de waarden terug. Als het verschil positief is, is `a` alfabetisch groter dan `b`. Als het negatief is, is `a` alfabetisch kleiner dan `b`.

De string `abcdz` is alfabetisch groter dan `abcdefg`. In een alfabetisch geordende lijst komt de string `abcdz` na `abcdefg`.

```
int strcmp(char *a, char *b)
{
  while (*a == *b) {               // while equal characters
    if (*a == '\0') return 0;      //   return if end-of-string
    a++;                           //   move a to next character
    b++;                           //   move b to next character
  }
  return  (*a - *b);
}
```

Het ophogen van de pointers kan ook op een andere plaats gebeuren. In dit geval is ook de retourwaarde aangepast, omdat `b` al opgehoogd is:

```
int strcmp(char *a, char *b)
{
  while (*a == *b++)
    if (*a++ == '\0') return 0;
  return  (*a - *(b-1));
}
```

Met een **for**-lus kan het ook zeer compact worden beschreven. De **for**-lus test op `*a == *b` en beide pointers worden opgehoogd nadat de actie is uitgevoerd:

```
int strcmp(char *a, char *b)
{
  for (; *a == *b; a++, b++)
    if (*a == '\0') return 0;
  return  (*a - *b);
}
```

De beschrijvingen van `strcpy` en `strcmp` maken duidelijk dat dit soort stringbewerkingen zeer beknopt beschreven kan worden.

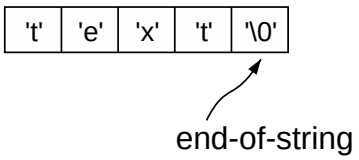
# 16

# Strings

| | |
|---|---|
| Doelstelling | Je leert in dit hoofdstuk wat een string is, hoe je in C een string toepast en waar je strings voor kunt gebruiken. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ Het *end-of-string*-teken.<br>▪ Het verschil tussen een **`char`** en een string.<br>▪ De declaraties van strings.<br>▪ Het toekennen aan en het overnemen van strings.<br>▪ De stringfuncties uit `string.h`.<br>▪ Arrays van strings.<br>▪ De argumentenlijst bij `main`.<br>Voorbeeldprogramma's voor strings tonen:<br>▪ Het omzetten van jaar, maand en dag in een datumcode.<br>▪ Het verschil tussen `strncpy` en `strlcpy`. |

C kent geen speciaal stringtype. Een string is niets anders dan een array van **`char`**'s, die afgesloten wordt met een *end-of-string*-teken. Dat is het karakter nul — de ASCII-waarde nul — en wordt in C aangegeven als `'\0'`. Bij het manipuleren van strings worden — net zoals bij arrays — vaak pointers gebruikt. Strings hebben zodoende veel gemeen met de arrays en pointers, die in hoofdstuk 14 en 15 zijn behandeld.

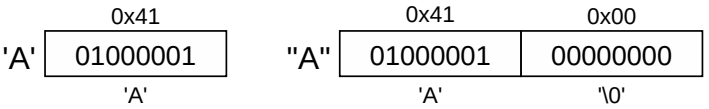
Figuur 16.1 visualiseert de string `"text"`. De string `"text"` bestaat uit de vier karakters van het woord *text* en het *end-of-string*-teken. Om het woord *text* te bewaren is dus een array van minimaal vijf **`char`**'s nodig.




**Figuur 16.1 : Een string is een array van char afgesloten met een `'0'`.** Voor de string `"text"` is een array met minimaal vijf karakters nodig. Het vijfde karakter is de *end-of-string*.

Een string wordt aangegeven met dubbele aanhalingstekens `""`. Enkele aanhalingstekens `''` worden gebruikt om karakters aan te duiden. `'A'` is een **`char`** met

de ASCII-waarde van A. Dat is hexadecimaal $0x41$ of decimaal 65. `"A"` is een array van twee **`char`**'s met de waarde `'A'` en `'\0'`. Het karakter `'\0'` is de ASCII-waarde nul en geeft het einde van de string aan. In figuur 16.2 is het geheugengebruik van `'A'` en `"A"` getekend.




**Figuur 16.2** : **Het verschil tussen een char en een string.** `'A'` is een **`char`** met de ASCII-waarde van A. `"A"` is een array van twee char's met ASCII-waarde `'A'` en `'0'`.

Aan het programma 16.1 worden drie argumenten meegegeven: jaar, maand en dag. Deze argumenten worden met `strcpy()` gekopieerd naar de stringvariabelen `year`, `month` en `day`. De strings `year` en `day` stellen getallen voor, die met `atoi()` worden omgezet in een integer. Met de functie `getmonth()` wordt de string `month` omgezet in een maandnummer. Het programma drukt tenslotte de datum in een zescijferig formaat (*yymmdd*) af.

De functie `getmonth()` zet een string om naar een maandnummer. Als de string een Engelstalige maand in kleine letters is en met een hoofdletter begint, wordt het maandnummer geretourneerd: 1 voor January, 2 voor February, tot en met 12 voor December. Met een **`if else if`**-constructie wordt met `strcmp()` steeds de string met de verschillende namen van de maanden vergeleken.

Voor de array `month` zijn tien karakters gereserveerd. Als het tweede argument uit meer dan negen letters bestaat past het bij het kopiëren niet meer in `month`. In een array van tien karakters is slechts plaats voor negen letters en de *end-of-string*. Een probleem is dat de gebruiker veel grotere strings kan invoeren, bijvoorbeeld: `sprokkelmaand` of `JanuaryorFebruary`. Een oplossing hiervoor is de arrays groter te maken. Vaak maakt men voor de veiligheid de array 64, 256 of bijvoorbeeld 1000 karakters groot. Het nadeel is dat heel veel geheugenruimte verloren gaat. Andere alternatieven zijn om dynamische geheugenallocatie te gebruiken of de invoer eerst te controleren of deze in de array past.

## 16.1 Declaratie van en toekenningen aan strings

Er zijn vele manieren om strings te declareren en te initialiseren. In onderstaand fragment wordt voor `t1` en `t2` in beide gevallen een geheugenplaats gereserveerd ter grootte van vijf **`char`**'s. Deze beide geheugenplaatsen worden gevuld met de vijf karakters `'t','e','x','t'` en `'\0'`. Er wordt precies zoveel ruimte gereserveerd als er nodig is voor deze tekst.

```
char  t1[] = "text";
char  t2[] = {'t','e','x','t','\0'};
char *t3   = "text";
```

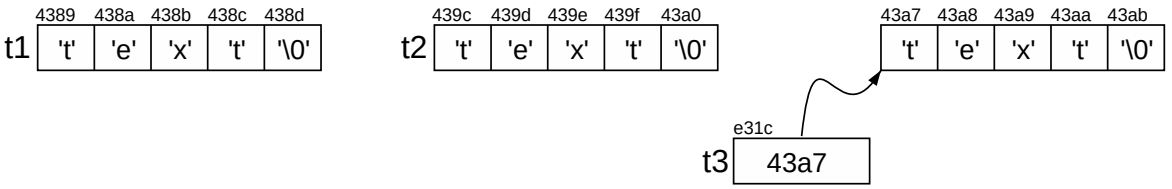
Bij de declaratie van `t3` reserveert de compiler geheugenruimte voor de pointer `t3` en ruimte voor de string `"text"`. Het beginadres van de string wordt aan de pointer `t3` toegekend. Figuur 16.3 visualiseert het geheugengebruik bij de declaratie van `t1`, `t2` en `t3`.

**Code 16.1 : Het omzetten van het jaar, de maand en de dag in een datumcode.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MSG_USAGE  "usage: %s <year> <month> <day>\n"
6  #define MSG_YEAR   "       <year> must be a number between 2000 and 2099\n
7  #define MSG_MONTH  "       <day> must be a number between 1 and 31\n"
8  #define MSG_DAY    "       <month> must the name of a month\n"
9
```

```
10 int getmonth(char *m)
11 {
12   if (strcmp(m, "January") == 0) {
13     return 1;
14   } else if (strcmp(m, "February") == 0) {
15     return 2;
16   } else if (strcmp(m, "March") == 0) {
17     return 3;
18   } else if (strcmp(m, "April") == 0) {
19     return 4;
20   } else if (strcmp(m, "May") == 0) {
21     return 5;
22   } else if (strcmp(m, "June") == 0) {
23     return 6;
24   } else if (strcmp(m, "July") == 0) {
25     return 7;
26   } else if (strcmp(m, "August") == 0) {
27     return 8;
28   } else if (strcmp(m, "September") == 0) {
29     return 9;
30   } else if (strcmp(m, "October") == 0) {
31     return 10;
32   } else if (strcmp(m, "November") == 0) {
33     return 11;
34   } else if (strcmp(m, "December") == 0) {
35     return 12;
36   } else {
37     return 0;
38   }
39 }
40
```

```
41 int main (int argc, char **argv)
42 {
43   char exename[32];
44   char year[5], month[10], day[3];
45   int  y,m,d;
46
47   strcpy(exename, *argv++);
48   if ( argc < 4 ) {
49     printf(MSG_USAGE, exename);
50     return 1;
51   }
52
53   strcpy(year, *argv++);
54   strcpy(month, *argv++);
55   strcpy(day, *argv++);
56
57   y = atoi(year);
58   if ( (y < 2000) || (y > 2099) ) {
59     printf(MSG_USAGE, exename);
60     printf(MSG_YEAR);
61     return 1;
62   }
63   y -= 2000;
64
65   d = atoi(day);
66   if ( (d < 1) || (d > 31) ) {
67     printf(MSG_USAGE, exename);
68     printf(MSG_DAY);
69     return 1;
70   }
71
72   if ( ( m = getmonth(month)) == 0) {
73     printf(MSG_USAGE, exename);
74     printf(MSG_MONTH);
75     return 1;
76   }
77
78   printf("%s %s %s is %02d%02d%02d",
79              day, month, year, y, m, d);
80
81   return 0;
82 }
```




**Figuur 16.3 : Geheugengebruik bij strings.**
Voor de strings `t1` en `t2` worden vijf karakters in het geheugen gereserveerd. De variabele `t1` verwijst naar het begin de array. Dat geldt ook voor `t2`. Voor pointer `t3` wordt een geheugenplek gereserveerd. Daar staat dan een pointer, die naar de geheugenplek wijst waar `"text"` staat.

```
char t4[32] = "text";
char t5[32] = {'t','e','x','t','\0'};
```

Bij `t4` en bij `t5` wordt ruimte gereserveerd voor 32 karakters. In beide situaties worden de eerste vijf **`char`**'s met de vijf karakters `'t'`, `'e'`, `'x'`, `'t'` en `'\0'` gevuld. De andere 27 **`char`**'s blijven ongedefinieerd. Als de string langer is dan 31 karakters, komt een deel van de string buiten de gedeclareerde geheugenruimte te staan. Het programma kan dan crashen.

```
char t6[32];

strcpy(t6, "text");
```

Het kopiëren van een string naar een pointer, die naar niets (NULL) wijst of die naar een te kleine geheugenruimte wijst, is een fout die door beginnende C-programmeurs vaak gemaakt wordt. Zelfs ervaren programmeurs maken hier fouten mee.

In bovenstaand voorbeeld wordt er een array `t6` gedeclareerd van 32 **`char`**'s. Met `strcpy` worden de eerste vijf plaatsen gevuld met de karakters `'t'`, `'e'`, `'x'`, `'t'` en `'\0'`. De rest van de array `t6` blijft ongedefinieerd. Als de string meer dan 31 karakters bevat, schrijft `strcpy` buiten de gedeclareerde geheugenruimte en kan het programma crashen.

```
char *t7;

strcpy(t7, "text");
```

Pointer `t7` wijst naar een willekeurige plek in het geheugen. Met `strcpy` wordt de string "text"naar deze willekeurige plek gekopieerd. Het programma zal crashen.

```
char  t8[]="text";
char *t9;

t9 = (char *)malloc( sizeof(char)*(strlen(t8)+1)) );
strcpy(t9, t8);
```

Hierboven staat een voorbeeld van dynamische geheugenallocatie. Pas tijdens de uitvoering van het programma wordt de geheugenruimte toegewezen. In dit voorbeeld is `t9` een pointer en `t8` een array. Met de functie `malloc` wordt een stuk geheugenruimte gealloceerd waar string `t8` precies in past (`strlen(t8)+1`). Vervolgens wordt string `t8` met `strcpy` gekopieerd naar de geheugenplaats waar `t9` naar wijst.

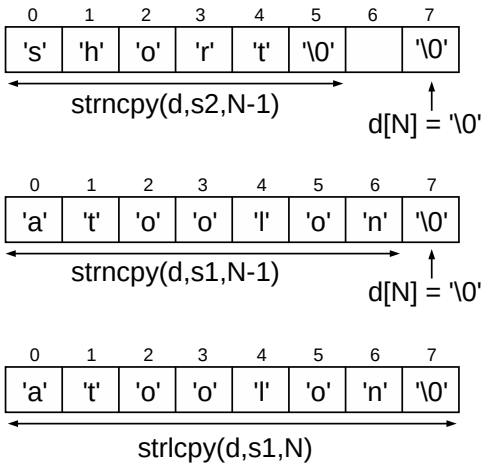
## 16.2 Op veilige wijze strings gebruiken

De functie `strcpy` kan, net als bijvoorbeeld de functie `strcat`, op ongewenste plaatsen in het geheugen schrijven.
Deels kan dit worden voorkomen door in plaats van `strcpy` de functie `strncpy` of `strlcpy` te gebruiken. In code 16.2 heeft de string `d` plaats voor acht karakters

inclusief de *end-of-string*. De functie `strncpy` kopieert maximaal N-1 karakters van `src` naar `des`. Figuur 16.4 laat zien dat het noodzakelijk is om op de laatste positie een *end-of-string* te plaatsen.
De functie `strlcpy` doet in principe hetzelfde als de functie `strncpy`, maar voegt zelf een *end-of-string* toe. Mits de correcte lengte van de bestemmingsstring wordt meegegeven kan er niet buiten de bestemmingsstring worden geschreven.

**Code 16.2 : Voorbeeld met strncpy en strlcpy.**

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  #define  N  8
 5
 6  int main (void)
 7  {
 8    char d[N];
 9    char s1[]="atoolongstring";
10    char s2[]="short";
11
12    strncpy(d, s1, N-1);
13    d[N] = '\0';
14    printf("%s\n", d);
15
16    strncpy(d, s2, N-1);
17    d[N] = '\0';
18    printf("%s\n", d);
19
20    strlcpy(d, s1, N);
21    printf("%s\n", d);
22    return 0;
23  }
```




**Figuur 16.4 : Het gebruik van de functies strncpy() en strlcpy().** String `s2` wordt door `strncpy` helemaal, inclusief de *end-of-string*, naar `d` gekopieerd. Het toevoegen van een `'\0'` op de laatste positie is overbodig, maar kan geen kwaad. Van string `s1` worden alleen de eerste zeven karakters naar `d` gekopieerd. De toevoeging van een `'\0'` op de laatste positie is hier noodzakelijk. De functie `strlcpy` kopieert maximaal zeven (N-1) karakters en plakt er zelf `'\0'` achter.

## 16.3 Stringfuncties

Er bestaan heel veel bewerkingen voor strings. Sommige stringfuncties horen bij de standaardbibliotheek en zijn bij alle C-compilers bekend. Anderen zijn alleen bekend bij een bepaalde groep compilers of vanaf een bepaalde versie. Oorspronkelijk kende C alleen `strcpy`. Tegenwoordig maakt `strncpy` deel uit van ANSI C. De functie `strlcpy` maakt daar nog geen deel van uit. Dit heeft gevolgen voor de overdraagbaarheid. Programma's met `strcpy` en `strncpy` zijn altijd te compileren. Een programma met `strlcpy` hoeft niet compileerbaar te zijn.
De meeste compilers kennen veel meer stringbewerkingen dan er standaard in ANSI C staan. Tabel 16.1 geeft een overzicht van de meest gebruikte stringfuncties. Het headerbestand voor deze functies is `string.h`.

**Tabel 16.1 : De belangrijkste stringfuncties uit string.h.**

| functie | omschrijving |
|---|---|
| strcpy | `char *strcpy(char *d, char *s);` |
| strncpy | `char *strncpy(char *d, char *s, size_t n);` |
| strlcpy | `size_t strlcpy(char *d, char *s, size_t n);` Deze functies kopiëren een string, waar `s` naar wijst, naar de geheugenplaats waar `d` naar wijst. Op pagina 170 zijn deze drie functies uitgebreid besproken. |
| strcat | `char *strcat(char *d, char *s);` |
| strncat | `char *strncat(char *d, char *s, size_t n);` |
| strlcat | `size_t strlcat(char *d, char *s, size_t n);` De functie `strcat` voegt een kopie van de string, waar `s` naar wijst, toe aan de string, waar `d` naar wijst. De functies `strncat` en `strlcat` werken hetzelfde als `strcat`, maar kopiëren maximaal `n` karakters naar de bestemmingsplek. De verschillende functies tussen `strcat`, `strncat` en `strlcat` komen overeen met die bij `strcpy`, `strncpy`, `strlcpy`. |
| strcmp | `int strcmp(char *a, char *b);` |
| strncmp | `int strncmp(char *a, char *b, size_t n);` De functie `strcmp` vergelijkt de string `a` met string `b`. Als `a` alfabetisch voor `b` komt, retourneert de functie een waarde kleiner dan nul. Als `a` alfabetisch na `b` komt, retourneert de functie een waarde groter dan nul. Als `a` en `b` identiek zijn, retourneert de functie nul. De functie `strncmp` doet hetzelfde als `strcmp`, maar vergelijkt maximaal `n` karakters. |
| strchr | `char *strchr(char *s, int c);` |
| strstr | `char *strstr(char *s, char *sub);` |
| strrchr | `char *strrchr(char *s, int c);` Functie `strchr` zoekt in string `s` naar de eerste maal dat het karakter `c` voorkomt. Functie `strstr` zoekt in string `s` naar de eerste maal dat de substring `sub` voorkomt. Functie `strrchr` zoekt in string `s` naar de laatste maal dat het karakter `c` voorkomt. Al deze drie functies retourneren een pointer naar de plaats van het gevonden resultaat of retourneren de nullpointer `NULL` als het gezochte niet voorkomt in `s`. |
| strlen | `int strlen(char *s);` De functie `strlen` retourneert het aantal karakters van de string, waar de pointer *s* naar wijst. Dat is de lengte van de string zonder de *end-of-string*. |
| strtok | `char *strtok(char *s, char *delimiters);` De string `delimiters` bevat een of meer scheidingstekens. De functie `strtok` haalt tekens, die gescheiden zijn met karakters uit `delimiters`, uit de string `s`. Het eerste teken wordt gevonden met `s` als eerste parameter. De volgende tekens worden gevonden als `strtok` opnieuw aangeroepen wordt nu met `NULL` als eerste parameter. De tekens is in `delimiters` kunnen bij elke volgende aanroep anders zijn. |
| strlwr | `char *strlwr(char *s);` |
| strupr | `char *strupr(char *s);` De functies `strlwr` en `strupr` converteren alle karakters van string `s` naar respectievelijk kleine letters (onderkast) of grote letters (bovenkast). Deze functies retourneren `s`. |

Bij verschillende stringfuncties wordt in de functieheader het type `size_t` gebruikt. Dat is een speciaal type om de grootte van een array aan te geven. `size_t` is gedefinieerd als een **`unsigned long`**:
**`typedef unsigned long`** `size_t;`

Van alle standaardfuncties bestaan beknopte handleidingen: de zogenoemde *man pages*. Figuur 16.5 toont een deel van de handleiding van de functie `strncpy`. Deze handleiding wordt getoond door in de Cygwin-omgeving achter de prompt `man strncpy` op te geven. Als slechts een deel van de handleiding zichtbaar is, kan met de spatiebalk vooruit worden gegaan, met *b* achteruitgegaan, met *j* en *k* respectievelijk een regel voor of achteruit. Met *q* wordt de handleiding afgesloten.

De *man pages* zijn ook op internet te vinden. Googelen op *man strncpy* levert een groot aantal hits op. De handleidingen van de C-functies verschillen bij de diverse Unix-varianten wel in opmaak en in formulering.

Het commando `info strncpy` toont ook de handleiding. De opdracht `info libc strings` geeft een overzicht van alle functies uit string.h.

```
/cc/string
NAME
   6.27 'strncpy'-counted copy string

SYNOPSIS
        #include <string.h>
        char *strncpy(char *DST, const char *SRC, size_t LENGTH);

DESCRIPTION
   'strncpy' copies not more than LENGTH characters from the string
   pointed to by SRC (including the terminating null character) to the
   array pointed to by DST. If the string pointed to by SRC is shorter
   than LENGTH characters, null characters are appended to the destination
   array until a total of LENGTH characters have been written.

RETURNS
   This function returns the initial value of DST.

PORTABILITY
   'strncpy' is ANSI C.
```

**Figuur 16.5 : Een deel van de *man page* van de stringfunctie strncpy. Deze handleiding wordt getoond als achter de prompt de opdracht `man strncpy` te geven.**

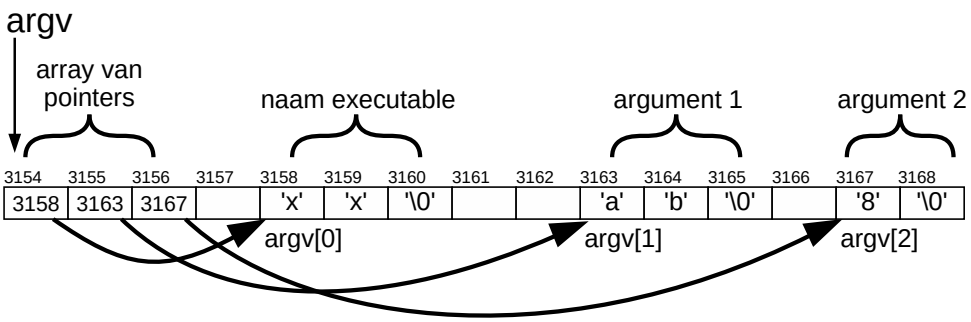
## 16.4 Array van strings

Een string is een array van **`char`**'s. Een array van strings lijkt daarom op een tweedimensionaal array. Bij het gebruik van de functie `main` met een argumentenlijst wordt een array van strings gebruikt.

```
int main(int argc, char *argv[])
```

Eerder is bij code 5.3 uitgelegd dat `argv[0]` de naam van het programma is en dat `argv[1]` het eerste argument is dat bij de aanroep is meegegeven.

In de parameterlijst van `main` is **`char`** `*argv[]` een array met pointers. De twee rechte haken `[]` geven aan dat het een array is en **`char`** `*` geeft aan dat het een array van pointers is. De pointers uit de array wijzen naar de geheugenplaatsen waar de argumenten staan. Naar een array met pointers kan net als naar alle arrays verwezen worden met een pointer.




**Figuur 16.6 : De argumenten bij een aanroep van main.** Pointer `argv` wijst naar een array van pointers, die op hun beurt weer naar de strings wijzen.

Code 16.1 laat zien dat de argumentenlijst van `main` ook zo geschreven wordt:

```
int main(int argc, char **argv)
```

Argument `argv` is nu een pointer naar een array met pointers. In dit geval gaat het om een pointer naar een **`char`**-pointer, oftewel: **`char`** `**`. Figuur 16.6 toont de argumenten bij een aanroep van een applicatie met de naam `xx` en met twee argumenten `ab` en `8`.

De C-programmeurs, die **`char`** `**` gebruiken, zijn meestal meer gecharmeerd van pointers dan van arrays en zullen in de code ook pointers gebruiken om de argumentenlijst langs te lopen. In de code staat dan bijvoorbeeld: `*argv`, `*(argv+2)`, `argv++`, `*argv++`. In code 16.1 kan in plaats van:

```
strcpy(exename, *argv++);
        ⋮
strcpy(year, *argv++);
strcpy(month, *argv++);
strcpy(day, *argv++);
```

ook dit staan:

```
i = 0;
strcpy(exename, argv[i]);
i++;
        ⋮
strcpy(year, argv[i]);
i++;
strcpy(month, argv[i]);
i++;
strcpy(day, argv[i]);
```

# 17

# Advanced C

Doelstelling

Dit hoofdstuk behandelt een aantal geavanceerde onderwerpen, die voor het programmeren van microcontrollers minder belangrijk zijn. Je leert hoe je in C naar een bestand kunt schrijven en uit een bestand kunt lezen. Verder leer je wat recursie is, wat zoekalgoritmen zijn en wat datastructuren zijn.

Onderwerpen

De behandelde onderwerpen zijn:

- Het creëren van een filepointer (`FILE *`) om naar een bestand te schrijven of uit een bestand te lezen.
- Lezen uit een bestand met `fscanf`, `fgets`, `fgetc` en `fread`.
- Recursie.
- Een zelfgeschreven recursieve zoekfunctie `quicksort` .
- De standaard functie `qsort`.
- Pointers naar functies.
- Het dynamische alloceren van geheugenruimte met `malloc`.
- Het gebruik van structuren (**`struct`**).
- De lijst als datastructuur.

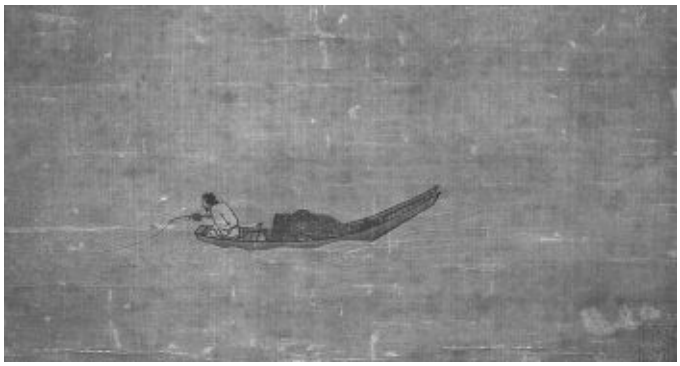
Voorbeelden zijn:

- Het lezen uit een bestand met `fscanf`.
- Het lezen uit een bestand met `fgets`.
- Het lezen uit een bestand met `fgetc`.
- Het lezen uit een bestand met `fread`.
- Het recursief berekenen van de faculteit van een getal.
- Het recursief genereren van de getallen van Fibonacci.
- Een iteratieve functie voor het berekenen van de faculteit van een getal.
- Een iteratieve functie voor het genereren van de getallen van Fibonacci.
- Het sorteren van een rij getallen met `quicksort`.
- Het sorteren van een verzameling gegevens met `qsort`.
- Het afdrukken van een array van gegevens met een bepaalde datastructuur.
- Het gebruik van een lijst met records.

## 17.1 Lezen en schrijven naar bestanden

Het lezen en het schrijven naar bestanden is bij microcontrollers niet erg belangrijk. Het schrijven naar en lezen uit het EEPROM is daar veel belangrijker. Een verhaal over C zonder bestandswerkingen is echter niet compleet. C wordt vaak gebruikt om gegevens met een bepaald dataformaat om te zetten in een ander dataformaat. Het lezen uit en het schrijven naar bestanden is daarvoor essentieel.

De voorbeelden uit dit hoofdstuk maken gebruik van een tekstbestand met het gedicht 'Visser van Ma Yuan' van Lucebert. Lucebert (1924-1994) is een Nederlands dichter, schilder, lithograaf en tekenaar die met zijn literaire werk gerekend wordt tot de Vijftigers. Zijn schilderwerk is sterk beïnvloed door Cobra. Veel Nederlanders kennen van Lucebert de tekst 'alles van waarde is weerloos'. Deze tekst staat in Rotterdam — in neonletters — op het gebouw van een verzekeringsmaatschappij. Het gedicht 'Visser van Ma Yuan' is geïnspireerd op een werk van de Chinese landschapsschilder Ma Yuan, die leefde rond 1200.



```
Visser van Ma Yuan

onder wolken vogels varen
onder golven vliegen vissen
maar daartussen rust de visser

golven worden hoge wolken
wolken worden hoge golven
maar intussen rust de visser

Lucebert
```

**Figuur 17.1** : **De Visser van Ma Yuan.** Links staat het schilderij van Ma Yuan en rechts het gedicht van Lucebert.

### Lezen uit een bestand met fscanf

Code 17.1 leest de tekst uit een bestand met de naam *ma_yuan.txt*. Het programma opent met de functie `fopen` het bestand; leest met `fscanf` de woorden uit de tekst; drukt deze onder elkaar af op het scherm en sluit tenslotte met `fclose` het bestand. Er is een buffer `buf` van 32 karakters. Dat betekent dat de woorden (strings) in het gedicht niet meer dan `31` letters mogen hebben.
Als in bestand *ma_yuan.txt* het gedicht van figuur 17.1 staat, is het resultaat de uitvoer van figuur 17.2.

```
/cc/files
/cc/files $ gcc -Wall -o yuan yuan.c

/cc/files $ yuan
Visser
van
Ma
Yuan
onder
wolken
vogels
varen
onder
golven
vliegen
vissen
maar
```

**Figuur 17.2** : **Een deel van de uitvoer van code 17.1.**

**Uitleg code 17.1 regel 7**
`FILE`

De datastructuur `FILE` is nodig bij bestandsbewerkingen. Een variabele van dit type bevat allerlei informatie over het bestand, zoals: de aanmaakdatum, de afmeting en de bestandsnaam.

**Code 17.1 : Het lezen uit een bestand met fscanf**

```
 1  #include <stdio.h>
 2
 3  char buf[32];
 4
 5  int main(void)
 6  {
 7    FILE * fp;
 8
 9    if ( (fp = fopen("ma_yuan.txt", "r")) == NULL ) {
10      printf("error: couldn't find or open file");
11      return 1;
12    }
13
14    while ( fscanf(fp, "%s", buf) > 0 ) {
15      printf("%s\n", buf);
16    }
17    fclose(fp);
18
19    return 0;
20  }
```

**Regel 9**
`fopen()`

`FILE *fopen(char *filename, char *mode);`
De functie `fopen` opent een bestand. Deze functie retourneert een filepointer `FILE *` naar een datastructuur met bestandsgegevens. In het geval dat het bestand niet geopend kan worden, wordt de nullpointer (`NULL`) geretourneerd. `fopen` heeft twee argumenten: de bestandsnaam `filename` en een string `mode`. Deze string bevat de zogenoemde mode en eventueel een paar opties. De bestandsnaam kan de naam van een bestand zijn of een bestandsnaam met het complete pad. Als de naam geen pad bevat zoekt `fopen` het bestand in de huidige map. Meestal zal dat de map zijn van waaruit het programma is gestart. Er zijn drie lees- en schrijfmodes:

- `r` lezen uit een bestand
- `w` schrijven uit een bestand
- `a` toevoegen aan een bestand

Aan deze mode kan een `b` of `t` en een `+` worden toegevoegd. De mode `"r+"` betekent dat er gelezen en geschreven kan worden. Een bestand lezen waarin ook geschreven kan worden is lastig en meestal niet nodig. Deze feature wordt weinig gebruikt. Een interessante optie is `a+`. Dit is de enige optie waarbij een bestand wordt gecreëerd als het nog niet bestaat. Sommige systemen maken onderscheid tussen twee soorten bestanden:

- `b` *binary* binair bestand
- `t` *text* tekstbestand (dit is de standaardinstelling)

Windows, Unix en Macintosh-systemen gebruiken afwijkende methoden om het einde van een regel of het einde van een bestand te beschrijven. Windows gebruikt twee karakters voor een *end-of-line* namelijk: `0x0D` en `0x0A`, oftewel de *carriage return* (<CR>) en de *linefeed* (<LF>). Unix gebruikt hiervoor maar een karakter: `0x0A` (<LF>). Het gevolg is dat Windows onderscheid moet maken tussen binaire en tekstbestanden. Unix heeft dit onderscheid niet nodig.

Bij het lezen op een Windows-systeem worden alle *end-of-lines* (<CR><LF>) geïnterpreteerd als nieuwe regels (`'\n'`). Schrijven van `'\n'` geeft op een Windows-systeem twee karakters (<CR><LF>) en op een Unix-systeem een karakter (<LF>). Het lezen van een tekstbestand, dat afkomstig is van ander systeem, kan daarom problemen geven. Alle schrijf- en leesfuncties hebben vaak eigen oplossingen voor het lezen en schrijven van bestanden van andere systemen. Dit verslechtert de overdraagbaarheid van een C-programma naar een ander systeem. Interpretatieverschillen zijn er eveneens bij de GNU-compilers op een Windows-systeem. De GNU-compiler van MinGW maakt onderscheid tussen tekst- en binaire bestanden. De GNU-compiler van Cygwin doet dat niet. Scott Brueckner heeft een goed artikel geschreven over het lezen en schrijven van bestanden in C. Hij adviseert om ook tekstbestanden altijd in de *binary mode* te openen.

**Regel 14**
`fscanf()`

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

De functie `fscanf` lijkt op `scanf`. Alleen leest `fscanf` niet van de standaardinvoer maar uit een bestand. De functie `scanf` is feitelijk een variant van `fscanf` met als invoerbestand de standaardinvoer `stdin`. De code

```
scanf("%d", &x);
```

is identiek aan:

```
fscanf(stdin, "%d", &x);
```

Een verschil met het lezen van de standaardinvoer, is dat het bestand eerst geopend moet worden met `fopen` en dat het na afloop weer gesloten wordt met `fclose`.

`fprintf()`

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

De functie `fprintf` lijkt op `printf`. Alleen schrijft `fprintf` niet naar de standaarduitvoer maar naar een bestand. De functie `printf` is feitelijk een variant van `fprintf` met als uitvoerbestand de standaarduitvoer `stdout`. De code

```
printf("%d", x);
```

is identiek aan:

```
fprintf(stdout, "%d", x);
```

**Regel 17**
`fclose()`

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

De functie `fclose` sluit, nadat eerst alle nog niet verwerkte data zijn weggeschreven, het bestand van de betreffende filepointer. Overigens worden alle gebruikte bestanden bij het afsluiten van een programma automatisch gesloten.
In veel gevallen is het niet noodzakelijk om het bestand expliciet te sluiten. Het is overigens wel een goede gewoonte om dat wel te doen.

### Overzicht van de lees- en schrijffuncties

C kent diverse functies om gegevens naar een bestand te schrijven en uit een bestand te lezen. De belangrijkste lees- en schrijffuncties staan in tabel 17.1.

**Tabel 17.1 : De lees- en schrijffuncties.**

| leesfuncties | |
|---|---|
| `fscanf()` | leest tekst met een gegeven formaat |
| `fgets()` | leest een regel (string tot `'\n'`) |
| `fgetc()` | leest een karakter |
| `fread()` | leest een gegeven aantal bytes |

| schrijffuncties | |
|---|---|
| `fprintf()` | schrijft tekst met een gegeven formaat |
| `fputs()` | schrijft een string |
| `fputc()` | schrijft een karakter |
| `fwrite()` | schrijft een gegeven aantal bytes |

Het schrijven naar bestanden levert meestal weinig problemen op. Het lezen uit bestanden is vaak lastig. Als er een fout gemaakt wordt, crasht het programma of blijft het in een lus hangen. De leesfuncties zijn bovendien niet in alle gevallen even geschikt. Tabel 17.2 geeft een overzicht van de kenmerken van de vier leesfuncties.

**Tabel 17.2 : Een overzicht van de kenmerken van de verschillende leesfuncties.**

| `fscanf()` | `fgets()` | `fgetc()` | `fread()` |
|---|---|---|---|
| • Leest tekst met een gegeven formaat. | • Leest een regel. | • Leest een karakter. | • Leest een gegeven aantal elementen van een gegeven aantal bytes. |
| • De exacte syntax van de data moet bekend zijn.<br>• Spaties in velden zijn lastig.<br>• Tab's en *end-of-lines* worden als white space gelezen.<br>• Uit een gewone tekst verdwijnt de regelopmaak.<br>• Als het misgaat is, gaat het goed mis. | • De overdraagbaarheid tussen de verschillende systemen (Windows, Unix) kan problemen geven, vooral met de *end-of-file* en de *end-of-lines*.<br>• Er is een buffer nodig voor de regels.<br>• De grootte van de buffer is vooraf moeilijk te bepalen. | • Alle karakters worden een voor een gelezen.<br>• Dat is niet erg efficiënt.<br>• Parsers gebruiken vaak `getc`, omdat het checken van de syntax van de data hiermee eenvoudig is. | • Leest grote datablokken in een keer binnen.<br>• Probleem is dat de grootte van de datablokken vaak moeilijk te bepalen is.<br>• Een oplossing is alle data (de hele file) in een keer te lezen.<br>• Wordt vaak gebruikt voor gestructureerde databestanden, waarvan het aantal velden vastligt. |
| • Geeft het aantal gelezen velden terug. | • Geeft `NULL` of een pointer naar de **`char`**-buffer terug. | • Geeft `EOF` of het gelezen karakter terug. | • Geeft het aantal gelezen elementen terug. |

Al de leesfuncties hebben hun eigen voor- en nadelen. Afhankelijk van het dataformaat dat gelezen wordt, zal de ene functie of de andere de voorkeur hebben. Verschillende compilers hebben extra mogelijkheden. De GNU-compiler kent bijvoorbeeld een routine `getline`, die hetzelfde werkt als `fgets` maar zelf ruimte alloceert als de buffer te klein is.

### Lezen uit een bestand met fgets

Stel dat het gedicht van Lucebert afgedrukt moet worden met regelnummers. Dan is het gebruik van `fscanf` niet mogelijk. Deze routine verwijdert immers alle *end-of-lines*. Een alternatief is om gebruik te maken van `fgets` zoals code 17.2 laat zien. Deze functie leest complete regels uit een bestand. Het programma drukt deze vervolgens met een regelnummer af. Integer `i` bevat het aantal regels. Elke keer als er een regel gelezen en afgedrukt is, wordt `i` met 1 opgehoogd.

Voor een Windows tekstbestand op een Unix-systeem is het maximale aantal zelfs 253, namelijk 256 min drie posities voor <CR>, <LF> en `'0'`.

In code 17.2 is de buffer voor het opslaan van een regel 256 karakters groot. De functie `fgets` leest een complete regel inclusief de *end-of-line*. De regels in het bestand mogen daarom niet meer dan 254 karakters hebben. De twee laatste posities moeten beschikbaar blijven voor een `'\n'` en een `'\0'`. Figuur 17.3 toont het resultaat van het programma van code 17.2.

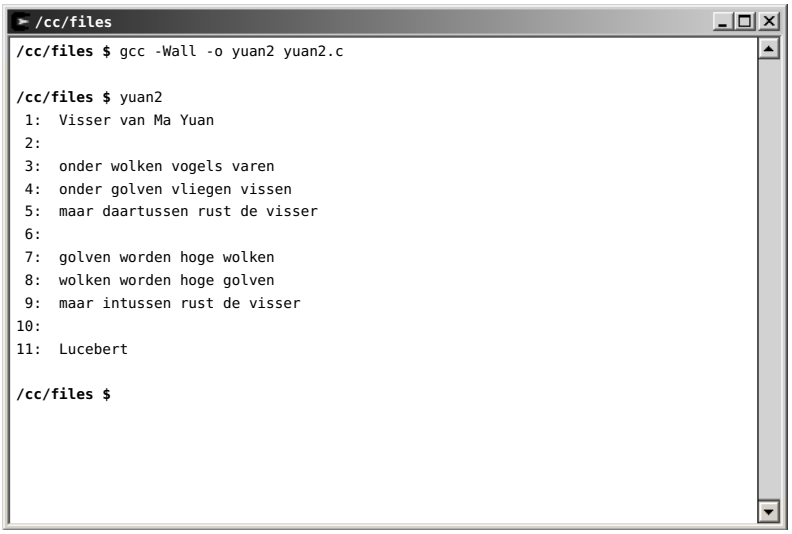
**Uitleg code 17.2 regel 18**
`fgets()`

```
char *fgets(char *b, int n, FILE *f);
```

De functie `fgets()` leest uit bestand `f` maximaal `n-1` karakters totdat er een *end-of-line* is gevonden. De karakters, inclusief `'\n'`, worden opgeslagen in buffer `b` en wordt afgesloten met `'\0'`. De routine retourneert de pointer naar de buffer of `NULL` als er helemaal geen data gelezen zijn. In het geval dat de `'\n'` niet gewenst

**Code 17.2 : Het lezen uit een bestand met fgets**

```
 1  #include <stdio.h>
 2
 3  #define  MAXBUF 256
 4
 5  char buf[MAXBUF];
 6
 7  int main(void)
 8  {
 9    FILE * fp;
10    int i;
11
12    if ( (fp = fopen("ma_yuan.txt", "r")) == NULL ) {
13      printf("error: couldn't find or open file");
14      return 1;
15    }
16
17    i=1;
18    while ( fgets(buf, MAXBUF-1, fp) != NULL ) {
19      printf("%2d: %s", i++, buf);
20    }
21    fclose(fp);
22
23    return 0;
24  }
```




**Figuur 17.3 : De uitvoer van code 17.2.**

is, kan deze vervangen worden door een `'\0'`:

```
fgets(buf, MAXBUF-1, fp);
buf[strlen(buf)-1] = '\0';
```

Het nadeel van deze oplossing is, dat dit compatibiliteitsproblemen kan geven. Een Windows tekstbestand heeft twee tekens als einde regel (<CR><LF>). Een Unix-programma verwacht er maar een teken (<LF>). Bij het lezen van een Win-

dows tekstbestand op een Unix-systeem blijft de carriage return gewoon staan. Een betere oplossing is om de *end-of-lines* op deze manier te verwijderen:

```
fgets(buf, MAXBUF-1, fp);
*strpbrk(buf, "\r\n") = '\0';
```

gets()

```
char *gets(char *b);
```

De functie `gets` leest direct van de standaardinvoer en lijkt op `fgets`. Omdat er geen maximaal aantal karakters wordt opgegeven kan er eenvoudig op andere plaatsen in geheugen gelezen worden. Programma's die `gets` gebruiken, hebben altijd een beveiligingslek. De leesroutine `gets` mag nooit gebruikt worden. Vervang `gets` altijd door `fgets`. Voor een buffer `buf` en met een lengte `BUFLENGTH` is dat:

```
fgets(buf, BUFLENGTH-1, stdin);
```

Alle *man pages* waarschuwen voor het gebruik van `gets`. Er staat bijvoorbeeld:

> *The `gets` function cannot be used securely. Because of its lack of bounds checking, and the inability for the calling program to reliably determine the length of the next incoming line, the use of this function enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. It is strongly suggested that the `fgets` function be used in all cases.*

Gebruik dus nooit de functie `gets`.

fputs()
puts()

```
int fputs(char *s, FILE *f);
int puts(char *s);
```

De functie `fputs` schrijft de karakters van string `s`, zonder `'\n'`, naar het bestand, dat bij filepointer `f` hoort. De functie `puts` schrijft de karakters van string `s` naar de standaarduitvoer. De volgende vier toewijzingen schrijven alle vier een string naar de standaarduitvoer:

```
fputs(buf,stdout);
puts(buf);
fprintf(stdout, "%s\n", buf);
printf("%s\n",buf);
```

De functies `fputs` en `puts` zijn nuttig als er alleen strings worden geschreven. Dat geeft snellere en kleinere programma's. De functies `fprintf` en `printf` hebben de voorkeur als er verschillende soorten dataformaten (**`int`**'s, **`float`**'s) geschreven moeten worden en als de uitvoer een bepaalde opmaak moet krijgen.

### Lezen uit een bestand met fgetc

Het Engelse *to parse* betekent ontleden. Een *parser* is een programma, dat een tekst of andere invoer analyseert volgens een vaste grammatica en deze vastlegt in een datastructuur.

Met `fgetc` wordt één karakter uit het bestand gelezen. Dat is handig bij het *parsen* van een stuk tekst.

Code 17.3 verandert alle beginletters van de woorden in hoofdletters. Voor alle beginletters, behalve de eerste, staat witte ruimte (*white space*). De hulpvariabele `lastc` bevat het vorige karakter. Als dit een *white space* is wordt het gelezen karakter een hoofdletter. In figuur 17.3 staat het resultaat.

**Uitleg code 17.3 regel 15**
`fgetc()`

```
int fgetc(FILE *f);
```

De functie `fgetc` leest het eerstvolgende karakter uit het bestand, dat bij filepointer `f` hoort. Deze functie retourneert een **`int`** en geen **`unsigned char`**. Het gelezen karakter wordt door fgetc gecast naar een **`int`**. Zijn geen er karakters meer, dan retourneert `fgetc` een `EOF`.

Code 17.3 : Het lezen uit een bestand met fgetc

```
 1  #include <stdio.h>
 2  #include <ctype.h>
 3
 4  int main(void)
 5  {
 6     FILE * fp;
 7     int c, lastc;
 8
 9     if ( (fp = fopen("ma_yuan.txt", "r")) == NULL ) {
10        printf("error: couldn't find or open file");
11        return 1;
12     }
13
14     lastc=' ';
15     while ( (c = fgetc(fp)) != EOF ) {
16        if ( isspace(lastc) ) {
17           printf("%c", toupper(c));
18        } else {
19           printf("%c", c);
20        }
21        lastc = c;
22     }
23     fclose(fp);
24
25     return 0;
26  }
```

getc()
getchar()

```
int getc(FILE *f);
int getchar(void);
```

Qua functionaliteit is `getc` identiek aan `fgetc`. Alleen is `getc` een macro en geen functie. De macro `getchar()` is identiek aan `getc(stdin)`.

fputc()
putc()
putchar()

```
int fputc(int c, FILE *f);
int putc(int c, FILE *f);
int putchar(int c);
```

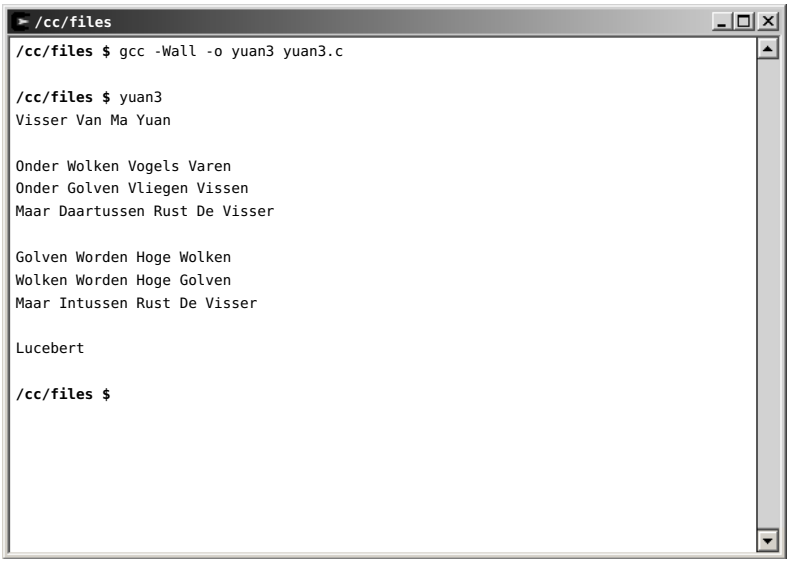
De functie `fputc` schrijft het karakter `c` naar het bestand, dat bij filepointer `f` hoort. De macro `putc()` is identiek aan de functie `fputc()`. De macro `putchar()` is identiek aan `putc(stdout)`.

### Lezen uit een bestand met fread

De functie `fread` kent niet de nadelen van `fscanf`, `fgets` en `fgetc`: alle karakters worden gelezen, de buffergrootte is goed te bepalen en het lezen van grote blokken gaat efficiënt. Het programma van code 17.4 bepaalt eerst de grootte van het het bestand, leest daarna het complete bestand en drukt tenslotte de tekst karakter voor karakter af.

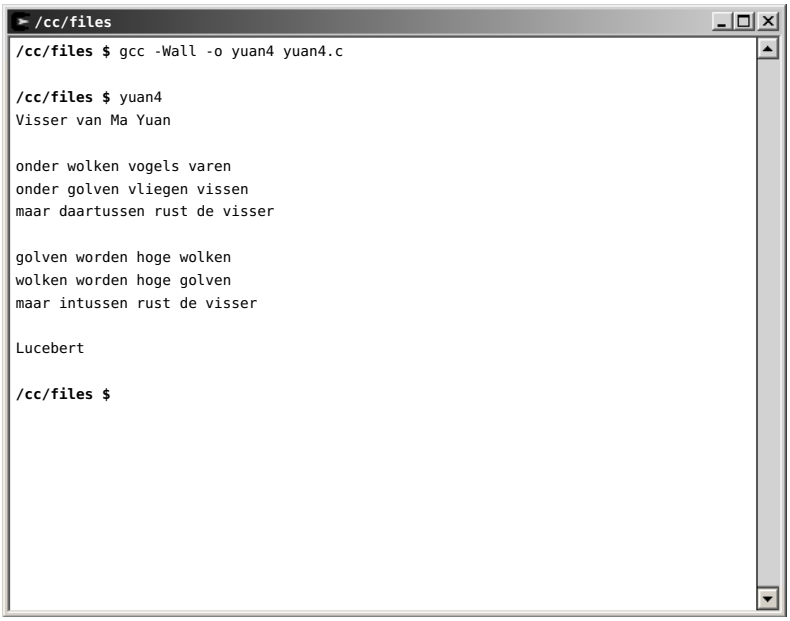
Voor het bepalen van de bestandsgrootte zijn drie functies nodig:

- `fseek`: zet de filepointer naar het einde van het bestand;
- `ftell`: geeft de positie ten opzichte van het begin van het bestand en dat is de bestandsgrootte;
- `rewind`: zet de filepointer weer terug naar het begin.




**Figuur 17.4 : De uitvoer van code 17.3.**

Nadat de grootte van het bestand bekend is, wordt met `malloc` geheugenruimte gealloceerd en wordt met `fread` het hele bestand naar deze geheugenplek gekopieerd.




**Figuur 17.5 : De uitvoer van code 17.4.**

**Uitleg code 17.4 regel 25**
`fread()`

```
size_t fread(void *buf, size_t size, size_t n, FILE *f);
```

De functie `fread` kopieert uit het bestand, dat bij filepointer `f` hoort, `n` elementen ter grootte van `size` naar een geheugenplaats waar de pointer `buf` naar wijst. De functie retourneert het aantal gelezen elementen.

**Regel 15**
`fseek()`

```
int fseek(FILE *f, long n, int pos);
```

De functie `fseek` verplaatst de filepointer `f` naar een plaats op `n` posities afstand van de startpositie `pos`. Deze laatste kan de volgende drie waarden hebben:

**Code 17.4 : Het lezen uit een bestand met fread**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6     FILE *fp;
7     int   size;
8     char *fbuf;
9
10    if ( (fp = fopen("ma_yuan.txt", "r")) == NULL ) {
11       printf("error: couldn't find or open file");
12       return 1;
13    }
14
15    fseek(fp, 0L, SEEK_END);       // search end of file
16    size = ftell(fp);              // get file size
17    rewind(fp);                    // back to start of file
18
19    if ( (fbuf = malloc((size + 1)*sizeof(char))) == NULL ) {
20       printf("error: not enough memory\n");
21       return 1;
22    }
23    fbuf[size] = '\0';             // append end of string
24
25    fread(fbuf, size, 1, fp);      // read the entire file
26
27    fclose(fp);
28
29    while (*fbuf != '\0') {
30       fputc(*fbuf++, stdout);
31    }
32
33    return 0;
34 }
```

**Regel 15**
`fseek()`

`SEEK_CUR` : de huidige positie van de filepointer
`SEEK_SET` : het begin van het bestand
`SEEK_END` : het einde van het bestand

Het einde van het bestand, dat hoort bij een filepointer `f`, wordt gevonden met:

```
fseek(f, 0L, SEEK_END);
```

**Regel 16**
`ftell()`

```
long ftell(FILE *f);
```

De functie `ftell` geeft de huidige positie van de filepointer in het bestand terug, gerekend vanaf het begin.

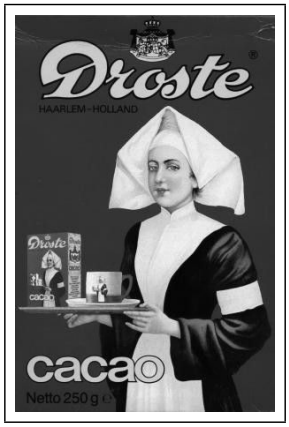
**Regel 17**
`rewind()`

```
void rewind(FILE *f);
```

De functie `rewind` zet de filepointer terug naar het begin van het bestand.

## 17.2 Recursie

Een recursieve functie is een functie die zichzelf aanroept. Deze functies geven vaak elegante oplossingen voor ingewikkelde problemen.




**Figuur 17.6 : Het droste-effect.**
Een voorbeeld van recursie is het droste-effect. Op de cacaobus staat een verpleegster, die een schaal vasthoudt met een cacaobus met daarop weer een verpleegster, die een schaal vasthoudt met ⋯

Sommige acroniemen (letterwoorden) zijn recursief. Voorbeelden zijn GNU (*GNU's Not Unix*) en PHP (*PHP: Hypertext Preprocessor*).

Een bekend voorbeeld is het berekenen van de faculteit. De faculteit $n!$ is gedefinieerd als:

$$n! \;=\; n * (n-1) * (n-2) * \cdots * 3 * 2 * 1$$

Formeel is dit te schrijven als:

$$n! \;=\; \begin{cases} 1 & \text{als} \quad n \;=\; 0 \\ n * (n-1)! & \text{als} \quad n \;>\; 0 \end{cases} \tag{17.1}$$

De faculteit van $n$ is $n$ maal de faculteit van $n-1$. Zo is 6! is $6 * 5!$. Om dit te berekenen moet de 5! berekend worden. Dat kan door $5 * 4!$ te berekenen. Dit gaat zo verder tot aan 1!.
Code 17.5 gebruikt een recursieve functie `fac` om 17! te berekenen. Deze functie is gebaseerd op de vergelijking 17.1. Omdat de faculteit van een getal snel heel groot wordt, is de retourwaarde van `fac` van het type **`unsigned long long`**.

**Code 17.5 : De faculteit van een getal.**

```
1  #include <stdio.h>
2
3  unsigned long long fac(int n)
4  {
5    if ( n==0 ) {
6      return 1 ;
7    } else {
8      return n * fac(n-1);
9    }
10 }
11
12 int main(void)
13 {
14   int i=17;
15
16   printf("%2d! is %llu\n", i, fac(i));
17
18   return 0;
19 }
```

Een ander voorbeeld is de reeks van Fibonacci. De getallen uit deze reeks voldoen aan deze recursieve vergelijking:

$$\text{fib}(n) \;=\; \begin{cases} 1 & \text{als } n \;=\; 0 \vee n \;=\; 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{als } n \;>\; 0 \end{cases} \tag{17.2}$$

Code 17.5 gebruikt een recursieve functie `fib` om de eerste twintig getallen uit de reeks van Fibonacci te berekenen. Deze functie is gebaseerd op vergelijking 17.2.
Recursieve oplossingen ogen vaak elegant. Ze zijn kort en lijken eenvoudig. Voor bepaalde problemen, zoals bij zoekalgoritmen en bij het werken met bomen en lijsten, zijn recursieve oplossingen ideaal.
In andere gevallen kan er evengoed — of zelfs beter — een iteratieve oplossing, dat is een oplossing met **`for`**- of **`while`**-lussen, worden gebruikt. In code 17.7 en in code 17.8 staan de iteratieve functies voor de faculteit en voor de getallen van Fibonacci.

**Code 17.6 : De reeks van Fibonacci als voorbeeld van recursie.**

```
 1  #include <stdio.h>
 2
 3  #define MAX_NUMBER 21
 4
 5  unsigned long long fib(int n)
 6  {
 7    if ( (n==0) || (n==1) ) {
 8      return 1;
 9    } else {
10      return ( fib(n-1) + fib(n-2) );
11    }
12  }
13
14  int main(void)
15  {
16    int i;
17
18    for (i=1; i<MAX_NUMBER; i++) {
19      printf("%2d! is %llu\n", i, fib(i));
20    }
21
22    return 0;
23  }
```

**Code 17.7 : Een iteratieve functie `fac`.**

```
 1  unsigned long long fac(int n)
 2  {
 3    unsigned long long f = 1;
 4
 5    for (;n>0; n--) {
 6      f *= n;
 7    }
 8
 9    return f;
10  }
```

**Code 17.8 : Een iteratieve functie `fib`.**

```
 1  unsigned long long fib(int n)
 2  {
 3    unsigned long long h, f1, f2;
 4    int i;
 5
 6    f1 = 1;
 7    f2 = 1;
 8    for (i=1; i<n; i++) {
 9      h  = f2;
10      f2 = f1 + f2;
11      f1 = h;
12    }
13
14    return f2;
15  }
```

Een belangrijk nadeel van recursieve functies is dat deze een groot beslag leggen op de performance van het systeem. Een aanroep van de recursieve functie `fib` met een getal $n$ uit code 17.6, geeft $2 * \texttt{fib}(n-1) - 1$ aanroepen van de functie `fib`. Voor het berekenen van `fib(8)` betekent het dat de functie `fib` 67 keer wordt aangeroepen. Dat kost erg veel rekentijd en geheugenruimte.
De iteratieve functie `fib` uit figuur 17.8 is vele malen sneller dan de recursieve functie `fib` uit figuur 17.6. De berekening van `fib(60)` duurt met de recursieve functie vele uren en met de iteratieve functie slechts een fractie van een seconde.
Een ander nadeel is dat, ondanks dat recursieve functies klein en overzichtelijk zijn, het schrijven en controleren van een recursieve functie lastig is.

**Code 17.9 : Sorteren van een rij getallen met quicksort.**

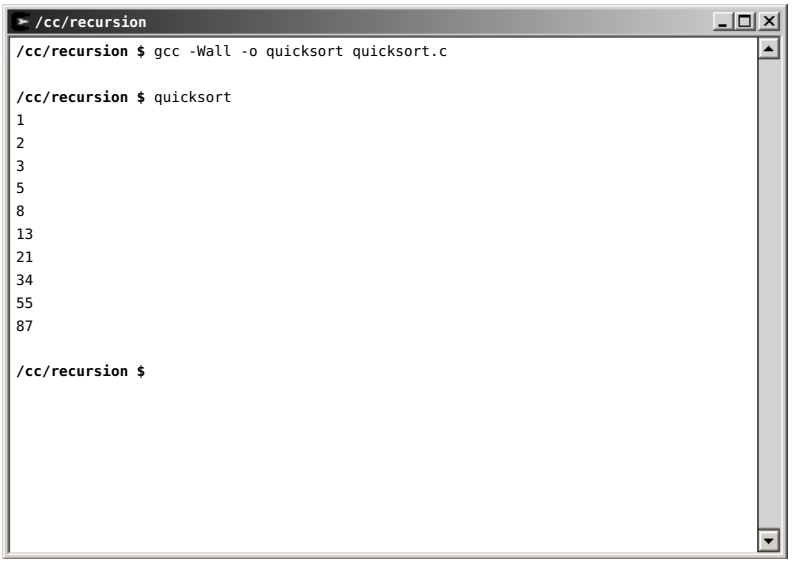
```
#include <stdio.h>

void swap(int a[], int i, int j)
{
  int tmp;

  tmp  = a[i];
  a[i] = a[j];
  a[j] = tmp;
}

void quicksort(int a[], int l, int r)
{
  int i,j;
  int v;

  if (r>l) {
    v = a[r];
    i = l-1;
    j = r;
    while (1)
    {
      while (a[++i] < v);
      while (a[--j] > v);
      if (i>=j) break;
      swap(a,i,j);
    }
    swap(a,i,r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
}

int main(void)
{
  int i;
  int a[]={87,2,1,13,21,8,55,5,3,34};

  quicksort(a, 0, 9);

  for(i=0; i<=9; i++) {
    printf("%d\n", a[i]);
  }

  return 0;
}
```

### Quicksort

Een voorbeeld waar recursie wel interessant is, is bij sorteeralgoritmen. Het sorteren van een array van gegevens is iets dat veel voorkomt. In de verkenner van Windows kunnen de bestanden bijvoorbeeld geordend worden op naam, grootte, type of wijzigingsdatum. Er bestaan veel algoritmen voor het sorteren, zoals: *bubble sort*, *merge sort*, *quick sort* en *heap sort*.

Een goed voorbeeld van recursie is quicksort, een sorteeralgoritme dat ontwikkelt is door Tony Hoare. In een rij gegevens wordt een element gekozen. De gegevens worden dan verdeeld in twee subgroepen. Links van het gekozen element komen alle gegevens te staan die kleiner zijn en rechts alle gegevens die groter zijn. Hetzelfde proces kan recursief worden toegepast op de subgroepen. De recursie stopt als een subgroep minder dan twee elementen heeft.

Code 17.9 sorteert met behulp van de functie `quicksort` een array van integers en drukt de gesorteerde rij af. Het resultaat staat in figuur 17.7.




**Figuur 17.7 : Het resultaat van de quicksort van code 17.9.**

De rij bestaat uit tien elementen en wordt helemaal gesorteerd van index `0` tot index `9`. De functie `quicksort` gebruikt een functie `swap` om twee elementen te verwisselen. De code van `quicksort` wordt hier niet volledig toegelicht. In de literatuur en op het internet zijn vele versies van het quicksort-algoritme te vinden. Het basis idee van quicksort is eenvoudig, maar de procedure om de waarden, die kleiner zijn dan het gekozen element, naar links te verplaatsen en alle andere elementen naar rechts is niet triviaal. De code tussen de regels 18 en 28 zorgt hiervoor.

### De standaard functie qsort

De methode quicksort is een van de beste algoritmen voor het sorteren van gegevens. Toch is de functie `quicksort` uit programma 17.9 niet de beste oplossing. Als de rij niet oplopend gerangschikt moet worden maar aflopend, moet de hele functie herschreven worden. Ook als er een array met woorden (strings) gesorteerd moet worden, moet de functie worden aangepast.

De standaardbibliotheek bevat een functie `qsort` die breder inzetbaar is dan de `quicksort` uit code 17.9. Het prototype van deze functie luidt:

```
void qsort(void *base, size_t n, size_t width,
                       int(*_compar)(const void *, const void*));
```

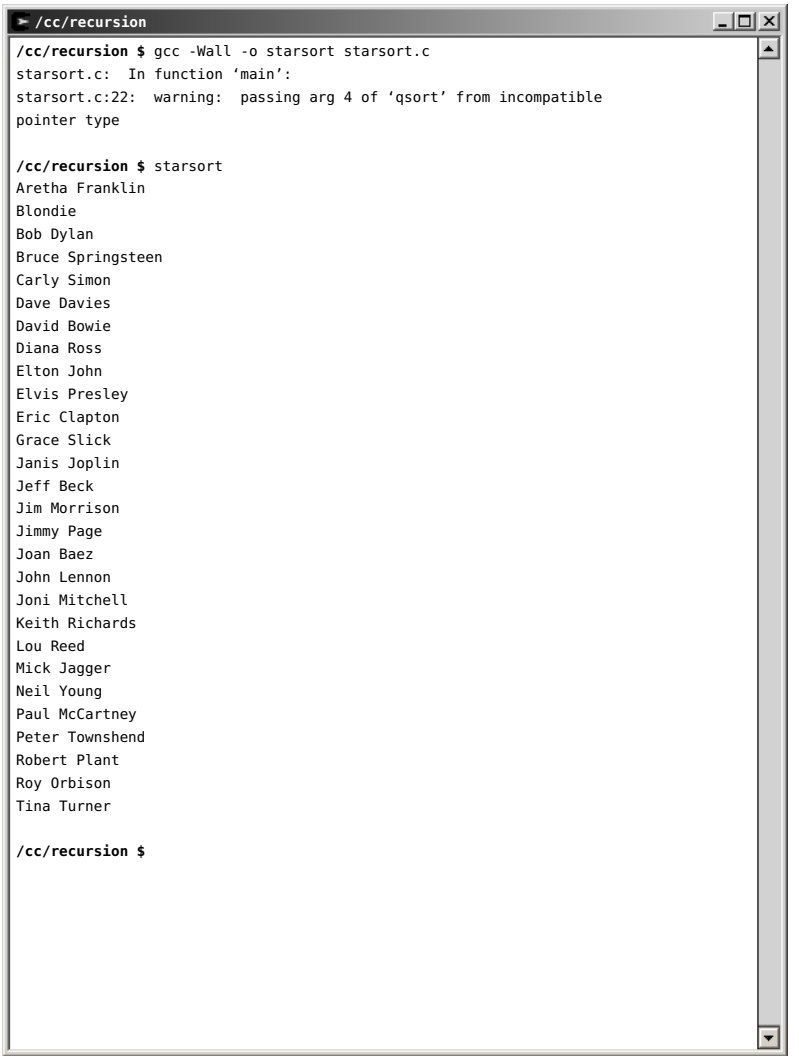
De functie `qsort` kan gebruikt worden voor elke regelmatige datastructuur. De pointer `base`, die naar de datastructuur wijst, is van het type **`void *`**. De datastructuur telt `n` elementen met een afmeting `width`. De vierde parameter is een pointer naar een functie. Deze functie heeft twee **`void`** pointers als ingangsparameters.

**Code 17.10 : Sorteren van een verzameling popsterren met `qsort`.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_LENGTH  32
6  #define NSTARS      (sizeof (stars)/sizeof(stars[0]))
7
8  char stars[][MAX_LENGTH] = {
9    "John Lennon", "Roy Orbison", "Elvis Presley", "Jim Morrison",
10   "Janis Joplin", "Aretha Franklin", "Paul McCartney", "Mick Jagger",
11   "Bruce Springsteen", "Elton John", "David Bowie", "Bob Dylan",
12   "Tina Turner", "Joan Baez", "Blondie", "Joni Mitchell",
13   "Robert Plant", "Diana Ross", "Lou Reed", "Carly Simon", "Neil Young",
14   "Eric Clapton", "Jimmy Page", "Keith Richards", "Peter Townshend",
15   "Grace Slick", "Jeff Beck", "Dave Davies"
16 };
17
18 int main(void)
19 {
20   int i;
21
22   qsort((char *)stars, NSTARS, sizeof(*stars), strcmp );
23
24   for (i=0; i<NSTARS; i++) {
25     printf ("%s\n", stars[i] );
26   }
27
28   return 0;
29 }
```

Code 17.10 geeft een toepassing van `qsort`. De array `stars` bevat de namen van een aantal grootheden uit de popgeschiedenis. Deze namen staan in een willekeurige volgorde. Met `qsort` worden deze namen alfabetisch gerangschikt, waarna de namen worden afgedrukt. Het resultaat staat in figuur 17.8

De array `stars` heeft een willekeurig aantal elementen. Het aantal elementen `NSTARS` wordt berekend met de definitie van regel 6. Op regel 22 wordt de functie `qsort` aangeroepen met `strcmp` als testfunctie. Dit is niet de juiste manier om `qsort` te gebruiken. Ten eerste geeft dit bij het compileren een waarschuwing, zoals in figuur 17.8 te zien is. Ten tweede is er een beperkt aantal standaardfuncties die als testfunctie bruikbaar zijn. De waarschuwing komt omdat `qsort` een functie verwacht met twee **`const void`** pointers als parameters en de `strcmp` heeft twee **`const char`** pointers. Het is beter om zelf een testfunctie te definiëren en die te ge-




**Figuur 17.8 : Code 17.10 drukt de sterren alfabetisch af.**

bruiken in plaats van `strcmp`. De functie `starcmp` heeft twee **`const void`** parameters `p1` en `p2` en retourneert het resultaat van `strcmp(p1,p2)`.

```
int starcmp(const void *p1, const void *p2)
{
  return(strcmp(p1,p2));
}
```

Een groot voordeel is dat er zo veel meer testvarianten gemaakt kunnen worden. Door `p2` en `p1` te verwisselen worden de popsterren in omgekeerde volgorde afgedrukt.

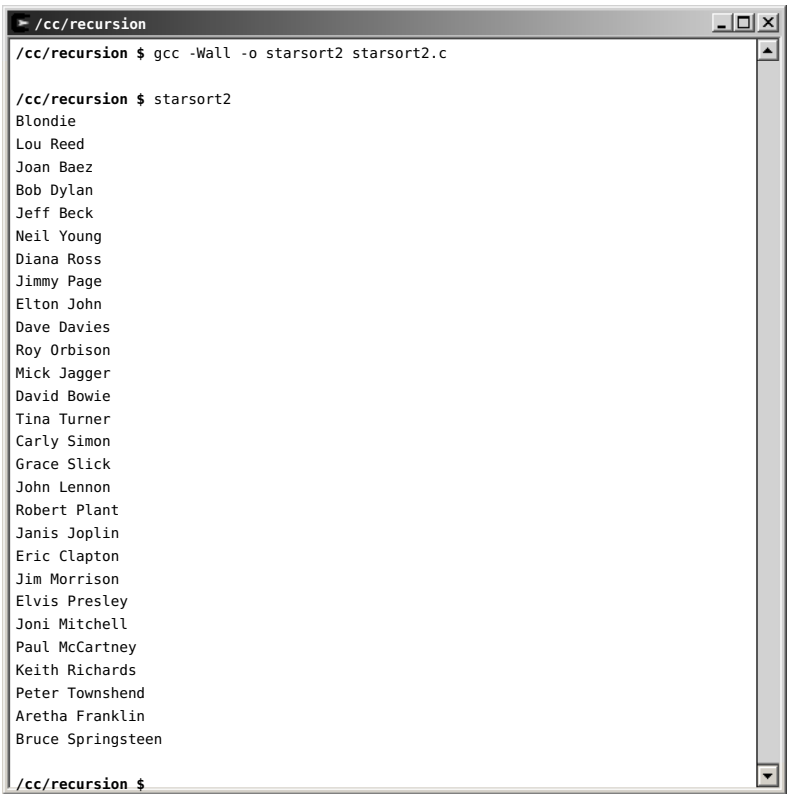
```
int starcmp_reverse(const void *p1, const void *p2)
{
  return(strcmp(p2,p1));
}
```

De functie `starcmp_size` vergelijkt de lengte van de strings waar `p1` en `p2` naar wijzen. Een testfunctie voor `qsort` moet altijd drie waarden retourneren: 1, −1 of 0.

In dit geval van `starcmp_size` wordt aangegeven dat de string van `p1` respectievelijk groter, kleiner of gelijk is aan `p2`.

```
int starcmp_size(const void *p1, const void *p2)
{
  int len1 = strlen(p1);
  int len2 = strlen(p2);

  if (len1 > len2) return 1;

  if (len1 < len2) return -1;

  return 0;
}
```

Als `starcmp_size` in code 17.10 wordt gebruikt in plaats van `strcmp` worden de namen van de sterren gesorteerd op grootte. Figuur 17.9 laat dit zien en toont bovendien aan dat de compiler nu geen waarschuwing geeft.




**Figuur 17.9 : De testfunctie starcmp_size sorteert de namen op grootte.**

### Pointers naar functies

De functie `qsort` maakt gebruik van een pointer naar een functie, zodat een eigen testfunctie aan `qsort` kan worden meegegeven. Dat maakt de functie `qsort` breed inzetbaar.

Functies staan op een willekeurige plaats in het geheugen. Net als bij andere datastructuren kan een pointer naar het begin van deze geheugenruimte wijzen.

**Tabel 17.3 : Voorbeelden van pointer naar functies.**

| declaratie | omschrijving |
|---|---|
| `void (*f) (int, int)` | Dit declareert een pointer `f` naar een functie met twee argumenten van het type **`int`** en een **`void`** retourwaarde. |
| `int (*g) (void)` | Dit is de declaratie van een pointer `g` naar een functie zonder argumenten en met een **`int`** als retourwaarde. |
| `(char *) (*h) (char *))` | Dit is een pointer `h` naar een functie met een **`char`**-pointer als argument en een **`char`**-pointer als retourwaarde. |

Veronderstel dat er een functie `square` is:

```
int square(int x)
{
  return (x*x);
}
```

Dan verwijst de naam `square` — net als dat het geval is bij een array — naar het beginadres van de functie. Een pointer `q`, die naar deze functie kan wijzen, wordt als volgt gedeclareerd:

```
int (*q) (int);
```

De syntax voor de declaratie van een pointer naar een functie is:

*`retourwaarde (*pointernaam)(argumentenlijst)`*

In tabel 17.3 staan een aantal voorbeelden van pointers naar functies. Pointer `q` kan als volgt gebruikt worden:

```
q = square;
printf("%d %d\n", square(13), q(13));  // print 169 169
```

De uitdrukkingen `square(13)` en `q(13)` zijn identiek.

## 17.3 Datastructuren

In paragraaf 15.6 is bij de toepassingen al gezegd dat pointers nuttig zijn bij datastructuren. Een datastructuur is een groep van gegevens, zoals de NAW-gegevens (naam, adres en woonplaats) van een persoon. In C wordt een datastructuur gedefinieerd met **`struct`**:

```
struct naw{
  char naam[128];
  char adres[128];
  char woonplaats[128];
};
```

De structuur `naw` bestaat uit drie velden: naam, adres en woonplaats. Een variabele `x` van dit type `naw` wordt gedeclareerd met:

```
struct naw x;
```

De velden van variabele `x` zijn beschikbaar door achter `x` een punt te zetten met daarachter de betreffende veldnaam.

```
strcpy(x.woonplaats, "Amsterdam");
strcpy(x.adres, "Weesperzijde 190");
printf("%s", x.naam);
```

Meestal wordt **struct** in combinatie gebruikt met **typedef** om een eigen type te maken:

```
typedef struct naw{
  char naam[128];
  char adres[128];
  char woonplaats[128];
} NAW;
```

Hierboven is een type NAW gedefineerd dat uit de structuur naw bestaat. De declaratie van een variabele x luidt in dit geval:

```
  NAW x;
```

Er staat geen **struct** voor NAW bij de declaratie van x.

Code 17.11 bevat een typedefinitie STUD van een datastructuur met drie velden: id, name en mobile om het studienummer, de naam en het mobiele telefoonnummer in op te slaan. De variabele stud_arr is een array van het type STUD en is gevuld met de gegevens van drie studenten. Het programma drukt de gegevens van deze studenten af.

**Code 17.11 : Het afdrukken van gegevens uit een array van een datastructuur.**

```
1  #include <stdio.h>
2  #define MAX_SHORT    16
3  #define MAX_NAME     256
4  #define NUM_STUDS    (sizeof(stud_arr)/sizeof(STUD))
5
6  typedef struct stud{
7    char   id[MAX_SHORT];
8    char   name[MAX_NAME];
9    char   mobile[MAX_SHORT];
10 } STUD;
11
12 STUD stud_arr[]={
13   {"324582", "Winston Churchill",     "0638421956"},
14   {"323732", "Franklin D. Roosevelt", "0665011934"},
15   {"325872", "Joseph Stalin",         "0692344555"}
16 };
17
18 int main(void)
19 {
20   int i;
21
22   for (i=0; i<NUM_STUDS; i++) {
23     printf("%s %s %s\n", stud_arr[i].id,
24                          stud_arr[i].name, stud_arr[i].mobile);
25   }
26
27   return 0;
28 }
```

Bij datastructuren gaat het meestal om een verzameling van dezelfde soort gegevens. In code 17.11 is hiervoor een array gebruikt. Vaak is het moeilijk in te schatten hoeveel gegevens er nodig zijn. Een te groot gekozen array kan geheugenproblemen geven bij het opstarten van de applicatie en een te klein gekozen

Een record is een set met gegevens. Een serie records vormen een lijst. Een record is dus een object van de lijst.

array kan de functionaliteit van het programma beperken. Daarom worden datastructuren vaak dynamisch gebruikt. In code 17.12, 17.13 en 17.14 wordt een lijst van studenten gebruikt. De functie `newStud` creëert een nieuw record voor de gegevens van een student. De naam en het studienummer van de student moeten bekend zijn. Het telefoonnummer kan later toegevoegd worden. De functie `appendStud` voegt steeds dit nieuwe record toe aan de lijst met studenten. De functie `printStuds` drukt de lijst af en tenslotte verwijdert `freeStuds` de lijst.

**Uitleg code 17.13 regel 4-9**

In het headerbestand `header.h` wordt het type `STUD` gedefinieerd, deze is afgeleid van **`struct`** `stud`. Dit type komt overeen met het type **`struct`** `stud` uit code 17.11, alleen is er een vierde veld `next` toegevoegd voor een pointer naar de eigen datastructuur (**`struct`** `stud*`). Dit maakt het mogelijk om een lijst van studenten te maken, zoals in figuur 17.10 is getekend.

**Uitleg code 17.13 regel 11-14**

Het headerbestand `header.h` bevat ook de prototypen van de functies voor de bewerkingen van de studentenlijst.

**Uitleg code 17.14 regel 6-17**

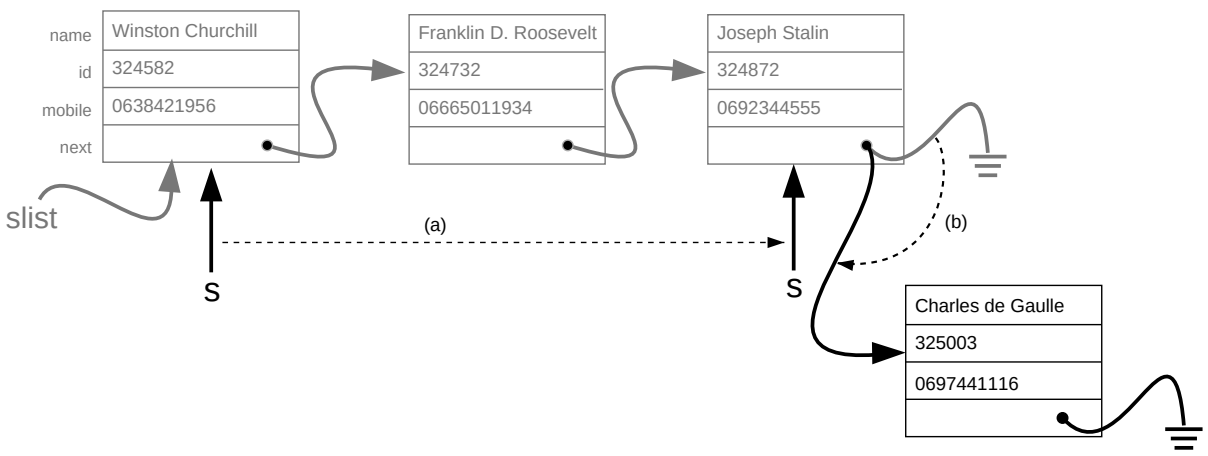
De functie `newStud` alloceert met de functie `malloc` een stuk geheugenruimte voor de gegevens van een student. Met `strcpy` worden het studienummer en de naam gekopieerd naar de betreffende velden. De waarde voor het veld `mobile` is nog niet bekend en wordt gevuld met een lege string. De pointer `next` wijst naar `NULL`.

**Uitleg code 17.14 regel 14 `->` `.`**

Het veld van een **`struct`** wordt bij een variabele gevonden door achter de variabele een punt te zetten met daarachter de naamveld, bijvoorbeeld `x.naam` en `stud_arr[i].name`. Bij een pointer naar een datastructuur wordt een veld gevonden door `->` achter de pointer te plaatsen en geen punt. Als `s` een pointer naar een datastructuur is, dan geeft `s->next` het veld `next` uit deze datastructuur.

**Uitleg code 17.14 regel 19-33**

De functie `appendStud` voegt een record toe aan de lijst. Als deze functie aangeroepen wordt met een lege lijst, wordt de pointer `news` geretourneerd.
Anders wordt het laatste element in de lijst gezocht en wijst de pointer `next` niet meer naar `NULL`, maar naar het record waar `news` naar wijst. In figuur 17.11 is dit grafisch weergeven.




**Figuur 17.11 : Het achteraan toevoegen van een record aan de lijst.** Eerst (a) wordt met een hulppointer `s` het laatste record van de lijst gezocht. Vervolgens (b) wordt er voor gezorgd dat de pointer `next` van het laatste element naar het nieuwe record gaat wijzen.

Code 17.12 : Afdrukken lijst met studenten: main.c.

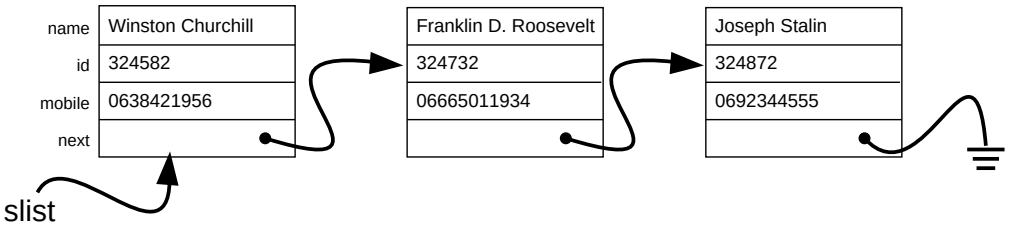
```
#include <string.h>
#include "student.h"

int main(void)
{
  STUD *studentlist=NULL;
  STUD *s;

  s = newStud("324582","Winston Churchill");
  strcpy(s->mobile, "0638421956");
  studentlist = appendStud(studentlist, s);
  s = newStud("323732", "Franklin D. Roosevelt");
  strcpy(s->mobile, "0665011934");
  studentlist = appendStud(studentlist, s);
  s = newStud("325872", "Joseph Stalin");
  strcpy(s->mobile, "0692344555");
  studentlist = appendStud(studentlist, s);

  printStuds(studentlist);

  freeStuds(studentlist);

  return 0;
}
```

Code 17.13 : Afdrukken lijst met studenten: student.h.

```
#define MAX_SHORT    16
#define MAX_NAME     256

typedef struct stud{
  char   id[MAX_SHORT];
  char   name[MAX_NAME];
  char   mobile[MAX_SHORT];
  struct stud *next;
} STUD;

STUD *newStud(char *id, char *name);
STUD *appendStud(STUD *slist, STUD *news);
void  printStuds(STUD *slist);
void  freeStuds(STUD *slist);
```

Code 17.14 : Afdrukken lijst met studenten: student.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "student.h"

STUD *newStud(char *id, char *name)
{
   STUD *s;

   s = (STUD *) malloc(sizeof(STUD));
   strcpy(s->id, id);
   strcpy(s->name, name);
   strcpy(s->mobile, "");
   s->next  = NULL;

   return s;
}

STUD *appendStud(STUD *slist, STUD *news)
{
  STUD *s = slist;

  if ( s == NULL ) {
    return news;
  }

  while ( s->next != NULL ) {
    s = s->next;
  }
  s->next = news;

  return slist;
}

void printStuds(STUD *slist)
{
  while ( slist != NULL ) {
    printf("%-7s %-22s %s\n", slist->id,
            slist->name, slist->mobile);
    slist = slist->next;
  }
}

void freeStuds(STUD *slist)
{
  STUD *s;

  while ( slist != NULL ) {
    s = slist;
    slist = slist->next;
    free(s);
  }
}
```




**Figuur 17.10 :** **De lijst met studentgegevens.** De pointer `slist` wijst naar het eerste record van de lijst. De pointer `next` van deze record wijst naar het tweede record. Dat gaat zo verder tot het laatste record. De pointer `next` van dit record wijst naar `NULL`.

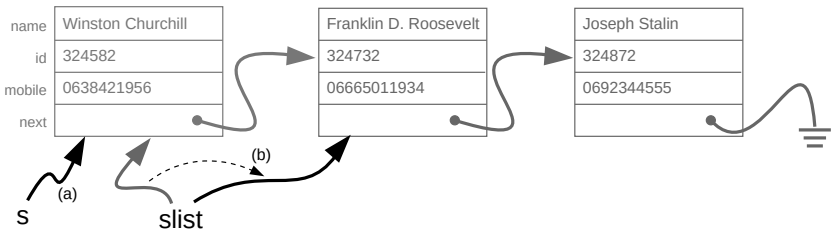
Deze functie gebruikt een hulppointer `s`, omdat de functie het begin van de lijst retourneert. De pointer `slist` wijst naar het begin en pointer `s` wordt gebruikt om de lijst langs te lopen.

**Uitleg code 17.14 regel 19-33**

De functie `printStuds` drukt de verschillende records uit de lijst af. De pointer `slist` wijst aanvankelijk naar het begin de lijst. De **`while`**-lus drukt de inhoud van het record waar `slist` op dat moment naar wijst af en laat daarna de pointer `slist` naar het volgende record uit de lijst wijzen. Dit gaat door totdat `slist` niet meer naar een record wijst. In deze functie wordt geen hulppointer `s` gebruikt. De functie `printStud` hoeft het begin van de lijst niet te onthouden. De pointer is lokaal gedefineerd. De pointer `studentlist` in `main.c` onthoudt waar het begin van de lijst is.

**Uitleg code 17.14 regel 44-52**

De functie `freeStuds` geeft de geheugenruimte die voor de studentenlijst gebruikt is weer vrij. Figuur 17.12 laat de methode zien waarmee steeds het eerste record verwijderd wordt. Deze functie is niet per se nodig omdat bij het afsluiten alle gebruikte geheugenruimte automatisch vrijkomt.




**Figuur 17.12 :** **Het verwijderen van een lijst.** Het voorste record van de lijst wordt steeds vrijgemaakt. Eerst (a) wordt er voor gezorgd dat een hulppointer `s` naar dit record wijst. Daarna (b) wijst `slist` naar het volgende record. Tenslotte wordt het losgemaakte record verwijderd.

# 18

# Analog-to-Digital Converter

**Doelstelling**

In dit hoofdstuk leer je wat analoog-digitaalconversie is en hoe een *Analog-to-Digital Converter* (ADC) werkt. Je leert hoe de ADC van de ATmega32 is opgebouwd en hoe je deze toepast.

**Onderwerpen**

De behandelde onderwerpen zijn:

- Analoog-digitaalconversie en dan met name de ADC, die gebaseerd is op successieve approximatie.
- De opbouw van de ADC bij de ATmega32: de ingangsselectie, het uitlezen van de uitgangsregisters, het instellen van de referentiespanning, de prescaling van het kloksignaal en de *single conversion*-, *automatic trigger*- en *free running*-mode.

Voorbeelden tonen vier verschillende manieren om de ADC te gebruiken:

- Single conversion mode zonder interrupt.
- Single conversion mode met interrupt.
- Automatic trigger mode met timer 0.
- Free running mode.

De microprocessor — het hart van een microcontroller — leest en schrijft alleen digitale signalen en voert daarop digitale rekenkundige en logische bewerkingen uit. In veel gevallen is de omgeving van de microprocessor niet digitaal. Er wordt bijvoorbeeld een druk, een temperatuur of een versnelling gemeten of er wordt een DC-motor bestuurd of een toon gevormd. De wereld van de gebruiker is in ieder geval niet digitaal maar fysisch.

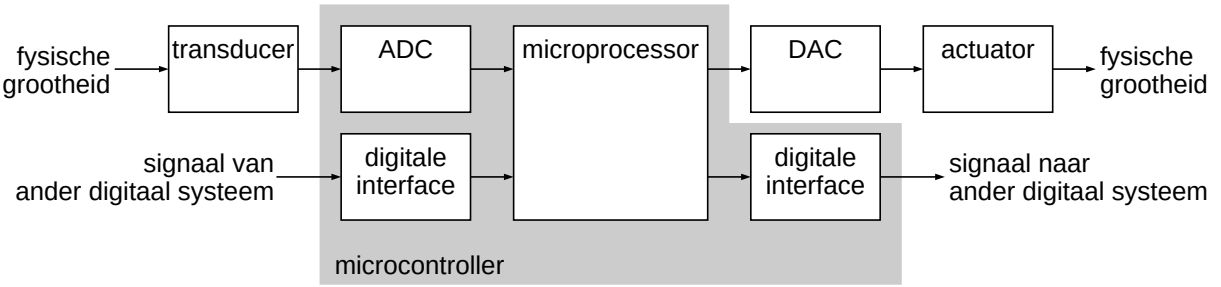
Transducers zetten fysische grootheden om in elektrische signalen.

Actuatoren zetten elektrische signalen om in fysische grootheden.

Soms wordt het begrip transducers ook gebruikt voor wat hier wordt bedoeld met een actuator. Een transducer is dan een omzetter van de ene vorm energie in een andere. Het begrip actuator wordt soms ook breder gebruikt. Het is dan vaak een apparaat dat een aanpassing verricht.

Er zijn transducers en actuatoren nodig om fysische grootheden om te zetten in elektrische analoge signalen en omgekeerd. Hoewel er tegenwoordig slimme transducers zijn die digitale waarden afgeven, leveren veel transducers een analoog signaal af. Voorbeelden zijn een temperatuursensor, een Hall-sensor voor het meten van een magnetisch veld, een rekstrookje voor het meten van druk, verplaatsingen of verdraaiingen en een fotocel voor het meten van de lichtintensiteit. Actuatoren worden vaak met een analoog signaal aangestuurd. Voorbeelden zijn een gelijkspanningsmotor, een wisselstroommotor en een luidspreker.

De microprocessor moet analoge waarden kunnen lezen en apparaten analoog kunnen aansturen. Er zijn componenten nodig die dat doen. De analoog-digitaalomzetter — *ADC, Analog-to-Digital Converter* — maakt een analoog signaal digitaal en de digitaal-analoogomzetter — *DAC, Digital-to-Analog Converter* — maakt digitale signalen analoog. Figuur 18.1 toont de stappen die nodig zijn om een fy-




**Figuur 18.1 : Een digitaal systeem in een fysische omgeving.** Fysische grootheden worden door transducers omgezet in analoge elektrische signalen die met een ADC digitaal worden gemaakt. Met een DAC wordt digitale informatie analoog gemaakt, waar een actuator mee kan worden aangestuurd. Een microcontroller bevat vaak een ADC om analoge signalen te kunnen lezen en een aantal digitale interfaces om met andere digitale systemen te communiceren.

sische grootheid om te zetten in een digitaal signaal en om een digitaal signaal om te zetten in een fysische grootheid.
Microcontrollers hebben meestal geen specifieke analoge uitgang met een DAC. Dat is niet nodig omdat een analoog signaal ook geconstrueerd kan worden uit een PWM-signaal — een pulsbreedte-gemoduleerd signaal — en een eenvoudig RC-netwerk.

Microcontrollers met analoge uitgangen zijn zeldzaam. De ADUC7026 van Analog Device heeft vier uitgangen met een DAC.

## 18.1 Analoog-digitaalconversie

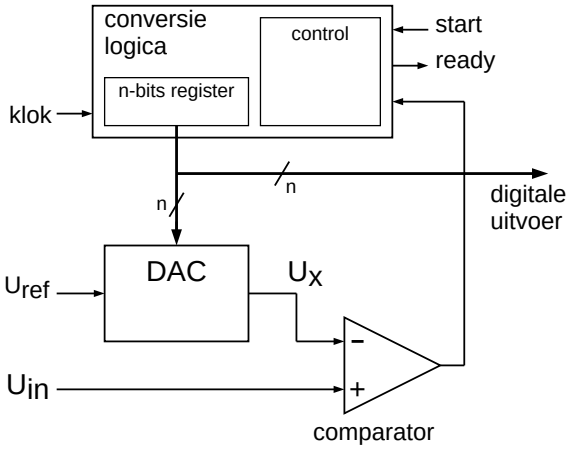
Er bestaan veel verschillende methoden om analoge signalen om te zetten in digitale signalen. Zeker als er een hoge nauwkeurigheid en een grote conversiesnelheid nodig zijn, zijn het complexe schakelingen. De belangrijkste en meest gebruikte omzetters zijn:

- *Ramp converter*
- *Dual slope converter*
- *Flash converter*
- *Successive approximation converter*
- *Sigma delta converter*

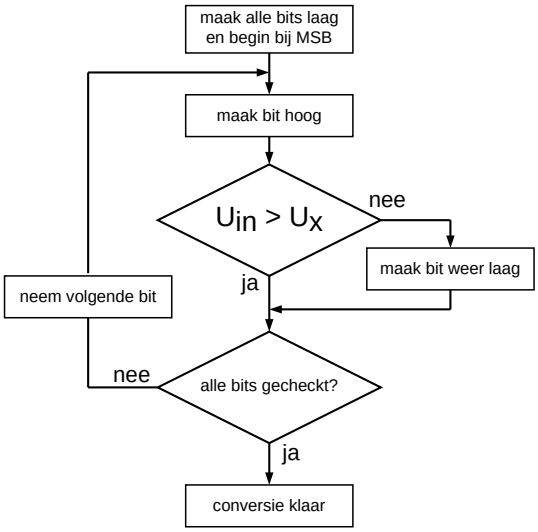
De ATmega32 gebruikt — net als veel andere microcontrollers — een ADC die gebaseerd is op successieve approximatie. Van deze ADC bestaan verschillende uitvoeringen, maar algemeen geldt dat deze convertor in verhouding snel en redelijk goedkoop te produceren is, maar dat de nauwkeurigheid beperkt is.

### ADC gebaseerd op successieve approximatie

Een ADC gebaseerd op successieve approximatie benadert stap voor stap de analoge waarde. Deze ADC is opgebouwd uit een DAC, een analoge comparator en conversie logica, die weer bestaat uit een n-bits dataregister en besturingslogica.
Figuur 18.2 geeft het schema en in figuur 18.3 staat het stroomdiagram met het algoritme voor de stapsgewijze benadering. De DAC zet de waarde uit het dataregister om in een analoge spanning. Deze spanning $U_x$ hangt af van de referentiespanning $U_{ref}$ en ligt tussen 0 en $U_{ref}$. De comparator vergelijkt $U_x$ met de ingangsspanning $U_{in}$ en de ADC past de waarde van het dataregister stapsgewijs aan totdat in dit register de waarde staat die de beste benadering is van de




**Figuur 18.2 :** **De opbouw van een ADC gebaseerd op successieve approximatie.**
De analoge comparator vergelijkt de door de DAC omgezette benadering $U_x$ met het ingangssignaal $U_{in}$.




**Figuur 18.3 :** **Het algoritme van successieve approximatie.**
Alle bits van het dataregister worden een voor een hoog gemaakt Als het ingangssignaal groter is dan de benadering blijft het bit hoog, anders wordt het weer laag gemaakt.

ingangsspanning. De uitgangsspanning $U_x$ van de DAC hangt af van de referentiespanning en van het aantal bits $n$ dat de DAC kan verwerken:

$$U_x = \frac{\text{data}}{2^n - 1} U_{ref} \tag{18.1}$$

De nauwkeurigheid van de ADC is gelijk aan die van de DAC en is gelijk aan:

$$\Delta U_x = \frac{1}{2^n - 1} U_{ref} \tag{18.2}$$

Meer informatie over de DAC en over deze vergelijkingen staat in bijlage C.

Als er met een 8-bits ADC bij een referentie van 5 V een digitale waarde 204 (11001100) gemeten wordt, is dat analoog:

$$\frac{204}{2^8 - 1} 5 = 4\ V$$

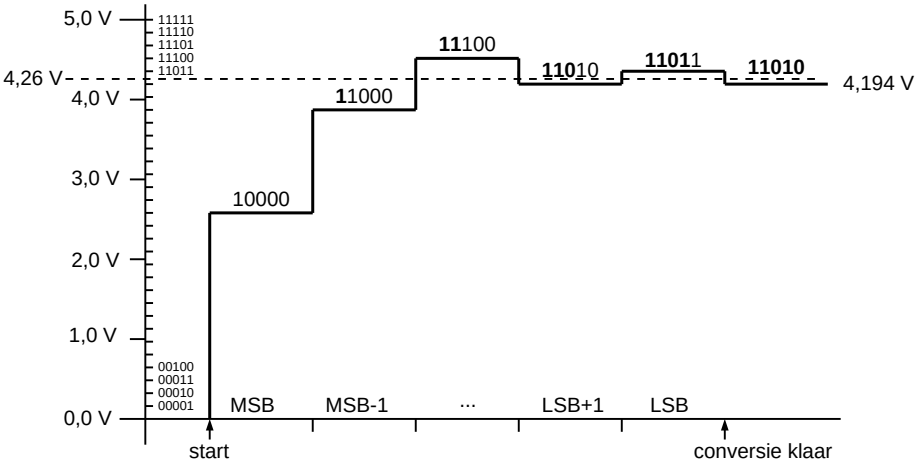
De fout, die er met een 8-bits ADC bij een referentie van 5 V gemaakt wordt, is:

$$\frac{1}{2^8 - 1} 5 \approx 0,0196\ V$$

De fout, die er met een 10-bits ADC bij een referentie van 5 V gemaakt wordt, is:

$$\frac{1}{2^{10} - 1} 5 \approx 0,00489\ V$$

Tijdens de omzetting staat in het dataregister steeds de benadering van het ingangssignaal. De DAC zet deze waarde om in een analoge signaal. De comparator vergelijkt dit signaal met het analoge ingangssignaal, dat geconverteerd moet worden.
De ADC maakt bij de start van de omzetting eerst het meest significante bit (MSB, *most significant bit*) hoog. Als het ingangssignaal groter is dan de benadering is dit bit terecht hoog en blijft het hoog. Anders is het bit ten onrechte hoog en wordt het weer laag gemaakt. Door dit een voor een voor alle bits te doen, krijgt het dataregister een waarde die zo dicht mogelijk bij het analoge ingangssignaal ligt. Figuur 18.4 laat stap voor stap zien hoe een ingangsspanning van 4,26 V door een 5-bits DAC met een referentiespanning van 5 V wordt benaderd door 4,194 V.
In dit voorbeeld wordt eerst het MSB-bit van het dataregister hoog gemaakt. De ingangsspanning is hoger dan de analoge spanning uit de DAC. Het bit is terecht hoog. Hierna wordt het volgende bit hoog gemaakt (`11000`). De ingangsspanning is nog steeds hoger, dus wordt `11100` geprobeerd. Nu is de ingangsspanning lager




**Figuur 18.4** : **De stapsgewijze benadering van een ingangssignaal van 4,26 V met een 5-bits ADC en een referentiespanning van 5 V.** De digitale eindwaarde is `11010` en de DAC geeft als het algoritme doorlopen is 4,194 V af.

en is het bit onterecht hoog. De volgende waarde is `11010`. De ingangsspanning is nu weer hoger en blijft het voorlaatste bit hoog. Tenslotte wordt `11011` gebruikt. Deze waarde is lager en het laatste bit wordt daarom weer gemaakt. Het eindresultaat luidt `11010` en dat is gelijk aan 4,194 V, zoals ook uit vergelijking 18.1 volgt.

De nauwkeurigheid van successieve approximatie is ongeveer een bit. De meetfout is gelijk aan formule 18.2. Naast deze fout kent de ADC een *zero scale error* of *offset error*, *full scale error* of *gain error* en *niet-lineariteiten*. De datasheet van de ATmega32 besteedt hier ruim aandacht aan en geeft als absolute fout twee bits.

De maximale snelheid van de conversie wordt bepaald door de klokfrequentie van de ADC en door het aantal bits dat geconverteerd moet worden. De conversietijd $t_{\text{conv}}$ van de ADC, die gebaseerd is op successieve approximatie, is evenredig met het aantal bits $n$ en de periode $T_{\text{adc\_clock}}$ van het kloksignaal:

$$t_{\text{conv}} = nT_{\text{adc\_clock}} \tag{18.3}$$

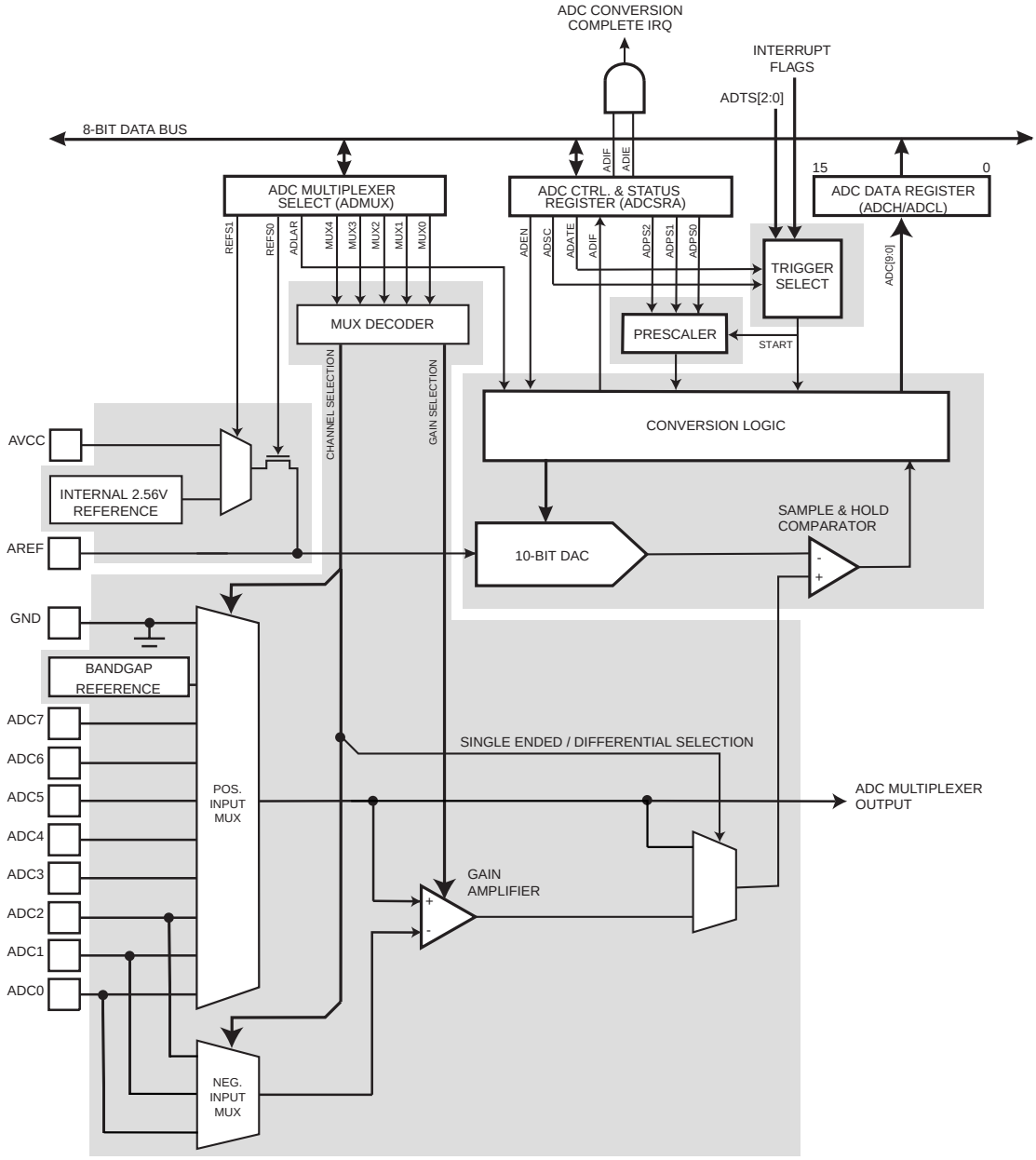
De maximale klokfrequentie van een ADC gebaseerd op successieve approximatie ligt vaak rond de 100 kHz. Een 10-bits ADC heeft daarom voor een conversie typisch minimaal 100 $\mu$s nodig.

## 18.2 De ADC van de ATmega32

Figuur 18.5 toont de opbouw van de ADC uit de datasheet van de ATmega32. Bovenaan staan twee 8-bits registers voor de instellingen van de ADC en een 16-bits register voor de opslag van de berekende waarde. Verder zijn er vijf blokken te onderscheiden: een prescaler voor het kloksignaal, een triggerselectieblok, een referentieblok, een ingangsselectieblok en de feitelijke ADC.

### Ingangsselectie ADC

De ATmega32 heeft één ADC, maar er zijn wel acht ingangen die met deze ADC verbonden kunnen worden. Alle ingangen `ADC0` tot en met `ADC7` van poort A kunnen de ingang van de ADC zijn. Bovendien bevat het ingangsselectieblok een verschilversterker waardoor er ook bepaalde combinaties en versterkingen van twee ingangen mogelijk zijn. Voorbeelden zijn `10×(ADC1-ADC0)`, `200×(ADC3-ADC2)` of `ADC7-ADC1`.

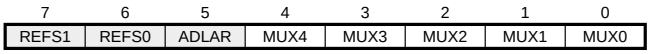



**Figuur 18.5 : De opbouw van de ADC van de ATmega32 uit de datasheet.**
Bovenaan staan het `ADMUX`- en het `ADCSRA`-register en een 16-bits register (`ADCH`/`ADCL`) voor de berekende waarde. Er zijn vijf functionele blokken te onderscheiden. Bovenaan bevindt zich een prescaler en een triggerselectieblok. Links staat een blok met de referentiespanning voor de DAC en onderaan staat het blok dat de ingang selecteert. Aan de rechterkant bevindt zich de feitelijke ADC met de DAC, de comparator en de conversielogica.

De multiplexers in het ingangsselectieblok maken 32 verschillende ingangscombinaties mogelijk. Deze mogelijkheden worden vastgelegd door de vijf bits `MUX0` tot en met `MUX4` uit het `ADMUX`-register uit figuur 18.6. De zogenoemde *single ended input*-selectie staat in tabel 18.1. De bits `MUX4` en `MUX3` zijn dan laag. Deze combinaties staan samen met de andere 24 combinaties in tabel 84 van de datasheet.

**Tabel 18.1 : De selectiebits voor het instellen van de ingang van de ADC.**

| MUX4..MUX0 | ingang ADC |
|---|---|
| 00000 | ADC0 |
| 00001 | ADC1 |
| 00010 | ADC2 |
| 00011 | ADC3 |
| 00100 | ADC4 |
| 00101 | ADC5 |
| 00110 | ADC6 |
| 00111 | ADC7 |
| 01--- | 24 mogelijke ingangs- en |
| 10--- | versterkingscombinaties, |
| 11--- | zie ook tabel 84 van de datasheet |



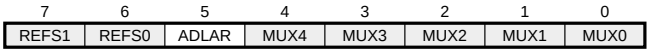

**Figuur 18.6 : De bits MUX0 tot en met MUX4 uit het register ADMUX bepalen de ingang of de ingangscombinatie die de ADC gebruikt.**

## Uitgangsregisters ADC

De ATmega32 is een 8-bits microcontroller. Voor de 10-bits waarde die de ADC berekent, zijn twee registers nodig: een register ADCH voor de meest significante bits en een register ADCL voor de minst significante bits. Deze registers zijn als 16-bits register rechtsboven in figuur 18.5 terug te vinden.

Is het ingangssignaal van de ADC een verschilsignaal, dan is de representatie van het uitgangssignaal two's complement. Dit is het geval als MUX4 en MUX3 uit ADMUX niet laag zijn.

De berekende waarde kan op twee manieren in deze twee registers komen te staan: links of rechts uitgelijnd. Als het ADLAR-bit uit het ADMUX-register hoog is, worden de bits links uitgelijnd en als dit bit laag is, worden de bits rechts uitgelijnd. Figuur 18.8 geeft de beide manieren waarmee de tien bits door de ADC in de registers ADCH en ADCL gezet worden en figuur 18.7 toont het ADLAR-bit uit het ADMUX-register.




**Figuur 18.7 : Het ADLAR-bit uit het ADMUX-register.**


**Figuur 18.8 : De twee manieren waarop de uitkomst in de uitgangsregisters ADCH en ADCL worden geplaatst.** Als het ADLAR-bit hoog is worden de bits links uitgelijnd en anders worden ze rechts uitgelijnd.

Als alle 10-bits gebruikt worden en het ADLAR-bit is laag, dan is de geconverteerde waarde x te bepalen uit ADCH en ADCL met:

```
unsigned int data;

data = ADCL | ((unsigned int)ADCH << 8) ;
```

Om er zeker van te zijn dat ADCL en ADCH bij dezelfde conversie horen, moet ADCL eerst gelezen worden.

En als ADLAR-bit hoog is, wordt de geconverteerde waarde gevonden met:

```
unsigned int data;

data = (ADCL>>6) | ( unsigned int) ADCH<<2);
```

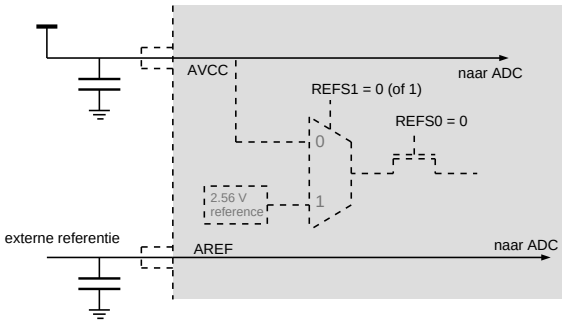
Het lijkt makkelijker om ADLAR laag te laten. Toch zijn er situaties dat dat niet zo is. Wanneer er een nauwkeurigheid nodig is van 8-bits, is het handiger om ADLAR hoog te maken. De geconverteerde waarde is dan gelijk aan de waarde van ADCH.

```
unsigned char data;

data = ADCH;
```

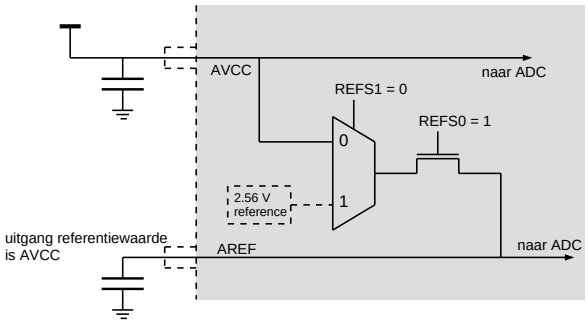
Formule 18.1 geeft het verband tussen de uitgangsspanning $U_x$ van de DAC, het aantal bits $n$ en digitale waarde in de dataregister ADCH en ADCL. Na de conversie is $U_x \approx U_{in}$. De datasheet gebruikt een formule voor $U_{in}$ die iets eenvoudiger is:

$$U_{in} = \frac{\text{data}}{2^n} U_{ref} \tag{18.4}$$

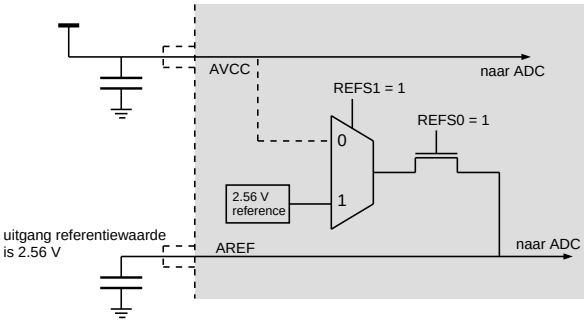
Voor een 10-bit ADC is het verschil tussen formule 18.1 en formule 18.4 niet heel groot. In ieder geval ligt het verschil ruim binnen de absolute nauwkeurigheid van ±2 LSB die de datasheet vermeldt. Dit boek gebruikt bij de voorbeelden formule 18.4.




(a) **Externe referentie met** REFS1=0 **en** REFS0=0.




(b) **Interne referentie AVCC met** REFS1=0 **en** REFS0=1.



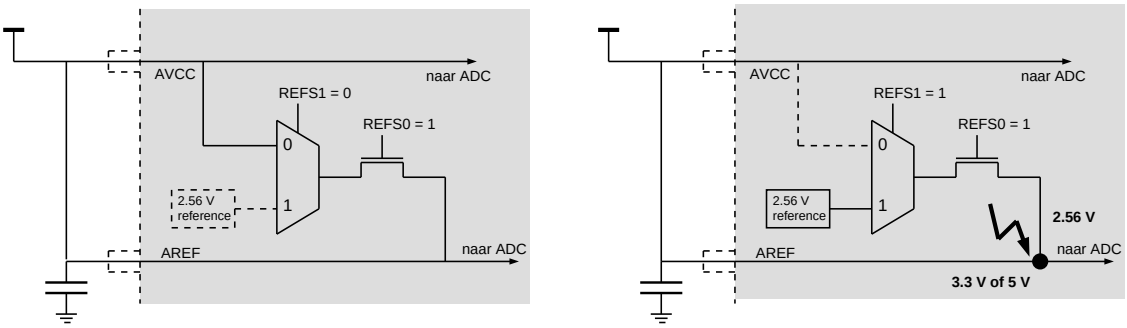

(c) **Interne referentie 2.56 V met** REFS1=1 **en** REFS0=1.

**Figuur 18.9 : De drie alternatieven voor de referentiespanning van de ADC.** Het grijze vlak is het referentieblok van de ADC. Alleen bij een externe referentiespanning, dus bij REFS0=0, is pin AREF een ingang. In alle andere gevallen staat op deze pin de interne referentiespanning.

## Referentiespanning ADC

De Digital-to-Analog Converter van de ADC heeft een referentiespanning nodig. Er zijn drie referenties mogelijk: een interne referentie, die gelijk is aan de analoge voedingsspanning AVCC; een interne referentie van 2,56 V of een externe referentie die aangesloten is aan de pin AREF.

Deze drie mogelijkheden worden geselecteerd met het REFS1-bit en het REFS0-bit uit het ADMUX register. De externe referentie is geselecteerd wanneer bit REFS0 laag is. Figuur 18.9.a laat zien dat de waarde van REFS1 er dan niet toe doet. De referentie is intern als REFS0 hoog is. In figuur 18.9.b is REFS1 laag en is de interne referentie gelijk aan AVCC. In figuur 18.9.c is REFS1 hoog en is de referentie gelijk aan 2,56 V.




**Figuur 18.10 : Mogelijke problemen als de voedingsspanning is aangesloten op AREF en tegelijkertijd de interne referentie (REFS0=1) is geselecteerd.** In de linker figuur is AVCC intern als referentie geselecteerd (REFS1=0). Er is geen probleem omdat de externe spanning ook AVCC is. In de rechter figuur is de interne referentie van 2,56 V geselecteerd (REFS1=1). Er is nu een kortsluiting tussen de extern aangesloten voedingsspanning en deze interne spanning.

De aansluiting AREF is een ingang bij een externe referentie en een hoogohmige uitgang bij een interne referentie. Er kan een kortsluiting optreden als er zowel een signaal is aangesloten op AREF en als tegelijkertijd de interne referentie is geselecteerd (REFS0=1). Zie ook figuur 18.10. Let er dus op dat de externe referentie (REFS0=0) geselecteerd is als er een externe spanning op AREF is aangesloten.

### Prescaling kloksignaal

Een veel gemaakte fout is dat men code overneemt, die bestemd voor een bepaalde kristalfrequentie en deze niet aanpast. De originele code is bijvoorbeeld geschreven voor 8 MHz en heeft een prescaling van 64. Als er dan de interne klok van 1 MHz gebruikt wordt, is de klokfrequentie van de ADC 16 kHz. Dat is veel te laag.

Bij de algemene uitleg van successieve approximatie is gesteld dat de klokfrequentie niet oneindig snel kan zijn. Dat komt omdat de comparator van de ADC altijd een zekere tijd nodig heeft om een stabiel signaal te geven. Daarnaast is het in praktijk wel verstandig de klokfrequentie hoog te maken. De comparator gebruikt een sample-and-hold-schakeling die de ingangswaarde gedurende een beperkte tijd vasthoudt. De ADC van de ATmega32 functioneert optimaal als de klok van de ADC tussen 50 kHz en 200 kHz ligt.

De frequentie van de systeemklok is meestal veel hoger dan 200 kHz. Daarom heeft de ADC een prescaler waarmee een klok afgeleid kan worden die wel voldoet. Tabel 18.2 geeft de zeven waarden waardoor de klok gedeeld kan worden en geeft de frequentiegebieden van de systeemklok waar deze prescaling voor bestemd is. Bij een klokfrequentie van 16 MHz moet de klok door 128 gedeeld worden. De klokfrequentie van de ADC is dan 128 kHz. Voor een systeemklok van 1 MHz is een prescaling van 8 nodig. De klokfrequentie van de ADC is dan eveneens 128 kHz.

De conversietijd van de ADC hangt af van de kloksnelheid. Bij de ATmega32 geldt formule 18.3, maar voor de verdere verwerking van de 10-bits conversie zijn bij een *single ended conversie* drie extra klokslagen nodig:

$$t_{\text{conv}} = 13T_{\text{adc\_clock}} \tag{18.5}$$

**Tabel 18.2 : De bits ADPS2, ADPS1 en ADPS0 uit het ADCSRA-register.** Gegeven zijn de prescaling en de frequentie waarvoor deze prescaling geschikt is.

| ADP2..ADP0 | prescaling | geschikt bij systeemklok (Hz) |
|---|---|---|
| 000 | 2 | < 400 k |
| 001 | 2 | < 400 k |
| 010 | 4 | 200 k – 800 k |
| 011 | 8 | 400 k – 1,6 M |
| 100 | 16 | 800 k – 3,2 M |
| 101 | 32 | 1,6 M – 6,4 M |
| 110 | 64 | 3,2 M – 12,8 M |
| 111 | 128 | > 6,4 M |


**Figuur 18.11 : De bits ADP2, ADP1 en ADP0 van register ADCSRA bepalen de prescaling van de klok van de ADC.**

Voor een klokfrequentie van de ADC van 128 kHz is de periodetijd $T_{adc\_clock}$ van het kloksignaal 7,8 $\mu$s. De totale conversie duurt dan ongeveer 0,1 ms.

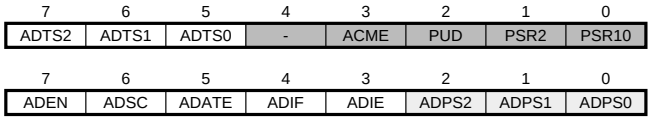
### Single conversion, automatic trigger mode en free running mode

Sommige ATmega's, zoals de ATmega8 en de ATmega128, hebben een ADC zonder triggerblok. Deze hebben geen *automatic trigger*-mode en geen ADATE-bit, maar kennen wel een *free running*-mode met een speciaal ADFR-bit.

De ATmega32 kent drie methoden om de ADC te gebruiken: de *single conversion*-, de *automatic trigger*- en de *free running*-mode. De *free running*-mode is een bijzonder geval van de *automatic trigger*-mode.
De verschillende conversiemethoden worden geselecteerd met het ADATE-bit (*ADc Auto Trigger Enable*) uit het ADCSRA-register en met de drie bits ADTS2, ADTS1 en ADTS0 uit het SFIOR-register (*Special FunctionIO Register*). De letters ADTS staan voor *ADc Trigger Select*. Tabel 18.3 geeft de drie modes met de waarden van het ADATE-bit en de ADTS-bits.

**Tabel 18.3 : De conversiemethoden (modes) van de ADC.**

| conversiemethode | ADATE | ADTS2..0 |
|---|---|---|
| single conversion | 0 | --- |
| free running | 1 | 000 |
| automatic trigger | 1 | niet 000 |




**Figuur 18.12 : De drie ADTS-bits van het SFIOR-register en de vijf bits van het ADCSRA-register regelen de AD-conversie:**
ADEN (*ADc ENable*)
ADSC (*ADc Single Conversion*)
ADATE (*ADc Automatic Trigger Enable*)
ADIF (*ADc Interrupt Flag*)
ADIE (*ADc Interrupt Enable*)

De ADC wordt aangezet door het ADEN-bit (*ADc ENable*) uit het ADCSRA-register hoog te maken. De ADC voert dan nog geen conversie uit. Conversies worden gestart door het ADSC-bit hoog te maken of door een *auto trigger*-signaal.
De ADC kan met en zonder interruptmechanisme gebruikt worden. Als het ADIE-bit (*ADc Interrupt Enable*) hoog is, staat het interruptmechanisme van de ADC aan. Het bit ADIF (*ADc Interrupt Flag*) is hoog als de conversie voltooid is en de uitkomst in de registers staat. Mits de globale interrupt en ADIE aan staan, start

ADIF de bijbehorende interruptfunctie. Het interruptmechanisme maakt ADIF automatisch laag. Figuur 18.12 toont de bits uit ADCSRA en SFIOR die voor het starten en het afhandelen van de AD-conversie relevant zijn.
De *single conversion*-mode wordt gestart door het ADSC-bit uit het ADCSRA hoog te maken. De conversie wordt direct uitgevoerd. Nadat de conversie klaar is, maakt de ADC automatisch het ADSC-bit weer laag. Het algoritme om de conversie te doen, ziet er dan zo uit:

In plaats van te testen of ADSC laag is, kan er ook getest worden op het hoog worden van ADIF. De test in de wachtlus luidt dan: `!(ADCSRA & _BV(ADIF))` Bij deze test is het verwarrend dat er op de interruptvlag ADIF gewacht wordt terwijl er geen interrupt gebruikt wordt.

```
ADCSRA |= _BV(ADSC);                    // start conversion
while (ADCSRA & _BV(ADSC));             // wait until conversion is ready
x = ADCL | ((unsigned int)ADCH << 8) ; // use result
```

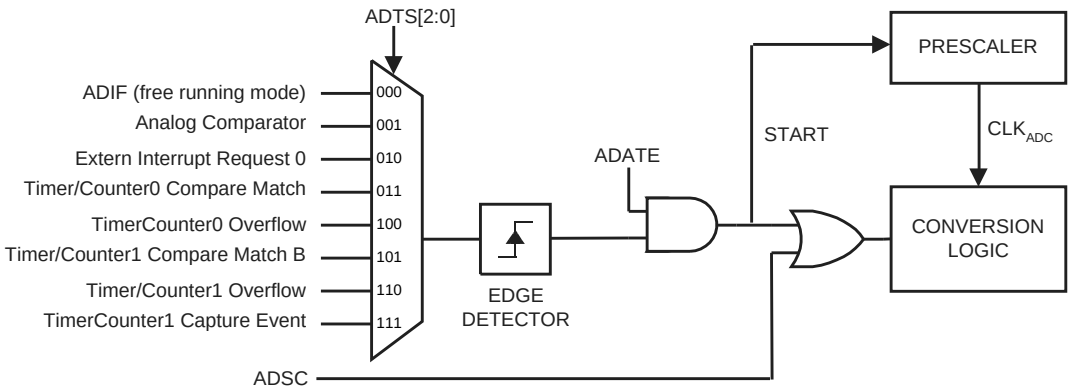
Na een conversie wordt niet alleen het ADSC-bit laag gemaakt, het ADIF-bit wordt ook hoog gemaakt. Dit bit triggert automatisch het interruptmechanisme, mits er een interrupt service routine aanwezig is. Het interruptmechanisme van *interrupt handler* maakt ADIF automatisch laag. De ISR hoeft daarom alleen de uitkomst van de conversie te verwerken en de volgende conversie te starten:

```
ISR(ADC_vect)
{
  x = ADCL | ((unsigned int)ADCH << 8) ;   // use result
  ADCSRA |= _BV(ADSC);                     // start next conversion
}
```

Het hoofdprogramma start de eerste conversie met:

```
ADCSRA |= _BV(ADSC);                    // start conversion
```

In paragraaf 18.3 wordt in code 18.1 wordt de *single conversion* zonder interrupt gebruikt en in paragraaf 18.4 wordt in code 18.2 deze met een interrupt gebruikt.

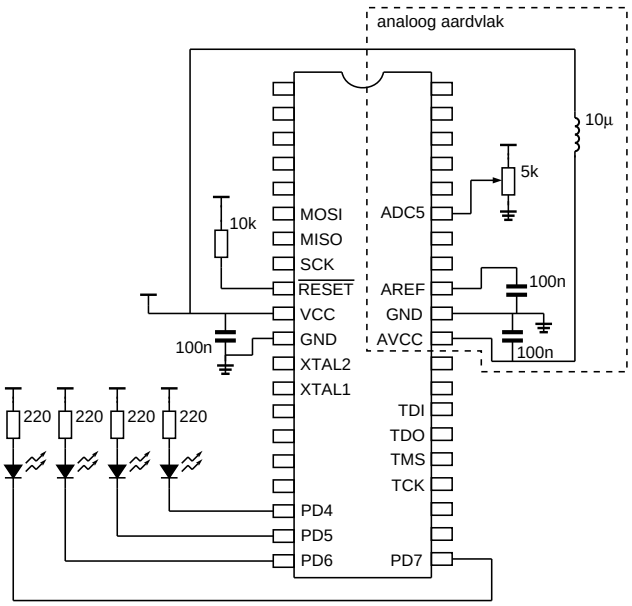



**Figuur 18.13 : Het triggerselectieblok van de ADC van de ATmega32.** Als ADATE hoog is, bepalen de bits ADTS2, ADTS1 ADTS0 het triggersignaal.

In de *automatic trigger*-mode wordt de AD-conversie gestart door een triggersignaal. Dat kan bijvoorbeeld de externe interne interrupt 0 of de overflow interrupt van timer 0 zijn. Figuur 18.5 bevat een triggerselectieblok. De schakeling hiervan staat in figuur 18.13. De *automatic trigger*-mode is vooral handig als er op vaste tijden een AD-conversie gedaan moet worden. Timer 0 genereert bijvoorbeeld elke 500 $\mu$s een overflow, die de ADC triggert. De *automatic trigger*-mode wordt geselecteerd door het ADATE-bit uit het ADCSRA-register hoog te maken. Het ADSC-bit wordt niet gebruikt en moet laag blijven. Het signaal START uit figuur 18.13

start de AD-conversie en reset de teller van de prescaler, waardoor de conversie synchroon loopt met het triggersignaal.

De *free running*-mode is een bijzonder geval van de *automatic trigger*-mode. Deze mode wordt geselecteerd door `ADATE` hoog te maken en `ADTS2`, `ADTS1` en `ADTS0` laag te maken. Deze instelling selecteert de interruptvlag `ADIF` als triggersignaal. Na een conversie wordt `ADIF` hoog en start automatisch een nieuwe conversie. De ADC voert zo continu conversies uit.
Alleen de eerste conversie wordt handmatig gestart door het `ADSC`-bit hoog te maken. Als de eerste conversie klaar is, wordt de interruptvlag `ADIF` hoog en start de volgende conversie.




**Figuur 18.14 : Het schema voor het meten van een analoog ingangssignaal.**
Het analoge signaal is aangesloten op pin 5 van poort A (pin `ADC5`). De referentie is intern aangesloten op VCC. Met de vier leds op bit 4 tot en met 7 van poort D wordt de analoge waarde van het ingangssignaal gevisualiseerd.
Er is een apart grondvlak voor het analoge deel. Tussen de digitale en analoge voeding is een zelfinductie van 10 $\mu$H aangebracht.

## 18.3 Toepassing single conversion mode zonder interrupt

Het basisalgoritme voor de *single conversion* is op pagina 206 besproken. Code 18.1 is hiervan een uitwerking. Het bijbehorende schema staat in figuur 18.14. De analoge ingang is pin 5 van poort A (`ADC5`). Op de uitgangen 4, 5, 6 en 7 van poort D zijn vier leds aangesloten. Het programma laat afhankelijk van de gemeten ingangswaarde een of meer leds branden volgens dit algoritme:

```
als U_in > 4,0 V
  vier leds aan
anders als U_in > 3,0 V
  drie leds aan
anders als U_in > 2,0 V
  twee leds aan
anders
  een led aan
```

Er wordt een interne referentiespanning gebruikt. In figuur 18.14 zit aan de referentiepin AREF een condensator van 100 nF om het signaalgedrag te verbeteren. Bij de PCB-layout voor deze schakeling is het verstandig om een apart aardvlak voor het analoge deel te maken. Tussen de analoge voeding en de digitale voeding is een zelfinductie van 10$\mu$ H aangebracht. Snel wisselende digitale signalen veroorzaken storingen in het analoge deel en beïnvloeden daarmee de nauwkeurigheid van de analoge metingen. De zelfinductie vermindert deze storingen.

Naast het aanbrengen van het aardvlak en de zelfinductie adviseert de datasheet om de analoge lijnen kort te houden en deze zo ver mogelijk van de digitale lijnen te plaatsen. Als van poort A ook pinnen digitaal gebruikt worden, mogen deze pinnen tijdens de AD-conversie niet schakelen. Tenslotte wordt geadviseerd een zogenoemde *ADC noise canceler* te gebruiken.

Code 18.1 beschrijft een *single mode*-conversie zonder interrupt. Regel 9 tot en met 12 bevat de initialisatie van de ADC. Alleen de bits die hoog zijn in het `ADMUX`- en `ADCSRA`-register zijn genoemd.

De microcontroller gebruikt de interne klok van 1 MHz. De klok voor ADC is door acht gedeeld zodat deze tussen 50 en 200 kHz ligt. Niet genoemde bits, zoals het `ADLAR`-bit uit het `ADMUX`-register, zijn laag. Het resultaat wordt dus rechts uitgelijnd. Het commentaar vermeldt deze impliciete instellingen wel. Een andere methode is om alle bits expliciet te benoemen:

```
      ADCSRA  = (1<<ADPS0) |   // prescaling 011 selects clk/8
                (1<<ADPS1) |   //
                (0<<ADPS2) |   //
                (0<<ADIE)  |   // no interrupt
                (0<<ADIF)  |   // interrupt flag (not used)
                (0<<ADATE) |   // single conversion mode
                (0<<ADSC)  |   // not yet started
                (1<<ADEN);     // ADC enabled
```

Het voordeel van deze methode is dat alle instellingen duidelijk zijn. Zo is nu zichtbaar dat `ADATE` laag is en dat de *automatic trigger*-mode uit staat en dat de *single conversion*-mode gebruikt wordt. Als er meer features van de microcontroller geïnitialiseerd worden, is het handig om voor elke feature een aparte initialisatiefunctie te maken. Voor de ADC kan deze er zo uit zien:

```
    void init_adc(void) {
      ADMUX   = (1<<MUX0)  |   // channel select 00101 is ADC5
                (0<<MUX1)  |   //
                (1<<MUX2)  |   //
                (0<<MUX3)  |   //
                (0<<MUX4)  |   //
                (0<<ADLAR) |   // right jusitified
                (1<<REFS0) |   // reference 01 is internal AVCC
                (0<<REFS1);    //
      ADCSRA  = (1<<ADPS0) |   // prescaling 011 selects clk/8
                (1<<ADPS1) |   //
                (0<<ADPS2) |   //
                (0<<ADIE)  |   // no interrupt
                (0<<ADIF)  |   // interrupt flag (not used)
                (0<<ADATE) |   // single conversion mode
                (0<<ADSC ) |   // not yet started
                (1<<ADEN);     // ADC enabled
    }
```

Tussen regel 15 tot en met 17 staat de methode voor de single conversion zonder interrupt van bladzijde 206.

Het **if-else-if**-statement van regel 19 tot en met 27 bevat het algoritme van bladzijde 207. Bij de berekening van de uitgangswaarde is niet vergelijking 18.1 gebruikt, maar vergelijking 18.4 die ook in de datasheet wordt toegepast.

**Code 18.1 : Ingangswaarde meten in single conversion mode zonder interrupt.**

```
1  #include <avr/io.h>
2
3  #define VREF  5
4  unsigned int x;
5
6  int main(void) {
7    DDRD    = 0xF0;                          // bit 4,5,6 and 7 of port D output
8
9    ADMUX   = 0x05|                          // channel ADC5 (is pin PA5), right justified
10             _BV(REFS0);                    // internal VCC selected
11   ADCSRA  = _BV(ADPS1)|_BV(ADPS0)|         // prescaling 8 (F_CPU=1MHz), no interrupt
12             _BV(ADEN);                     // single conversion, enable adc
13
14   while(1) {
15     ADCSRA |= _BV(ADSC);                   // start conversion
16     while (ADCSRA & _BV(ADSC));            // wait until conversion is ready
17     x = ADCL | ((unsigned int)ADCH << 8);  // use result
18
19     if ( x > (4*1024)/VREF ) {             // Vin > 4.0V
20       PORTD = 0xF0;
21     } else if (x > (3*1024)/VREF ) {       // 3.0V < Vin <= 4.0V
22       PORTD = 0x70;
23     } else if (x > (2*1024)/VREF ) {       // 2.0V < Vin <= 3.0V
24       PORTD = 0x30;
25     } else {                               // Vin <= 2.0V
26       PORTD = 0x10;
27     }
28   }
29 }
```

## 18.4 Toepassing single conversion mode met interrupt

Code 18.2 bevat een voorbeeld van een *single conversion* met een interrupt. Bij de initialisatie van de ADC is op regel 19 het interruptmechanisme van ADC en op regel 21 het globale interruptmechanisme aangezet.
Op regel 7 tot en met 11 staat de interrupt service routine. Deze routine hoeft slechts twee dingen te doen: de nieuwe waarde voor x te bepalen en de conversie opnieuw te starten.
De toewijzing op regel 23 start de eerste conversie. Het hoofdprogramma doorloopt continu de **while**-lus en wordt als een conversie klaar is, onderbroken om de ISR uit te voeren.
Het sleutelwoord **volatile** bij de declaratie van x is essentieel. Bij het optimaliseren van de code van het hoofdprogramma laat de compiler de ISR buiten beschouwing. In het hoofdprogramma verandert x niet en wordt verder behandeld als een constante en daardoor weggeoptimaliseerd. Het sleutelwoord **volatile** zegt dat de compiler x als variabele moet behouden.

Vanaf versie 4.13 compileert AVRstudio de code met debug-optie `-Os`. Dit is de sterkste optimalisatie. Voor versie 4.13 werd standaard `-O0` gebruikt. Dit is compileren zonder optimalisatie. Het effect van de optie `-Os` is dat er soms code weggeoptimaliseerd wordt, die absoluut nodig is. Variabelen kunnen met **volatile** beschermd worden.

**Code 18.2 : Ingangswaarde meten in single conversion mode met interrupt.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  #define VREF  5
5  volatile unsigned int x;
6
7  ISR(ADC_vect)
8  {
9    x = ADCL | ((unsigned int)ADCH << 8);    // use result
10   ADCSRA |= _BV(ADSC);                     // restart conversion
11 }
12
13 int main(void) {
14   DDRD    = 0xF0;                          // all bits port D output
15
16   ADMUX   = 0x05|                          // channel ADC5 (is pin PA5), right justified
17             _BV(REFS0);                    // internal VCC selected
18   ADCSRA  = _BV(ADPS1)|_BV(ADPS0)|         // prescaling 8 (F_CPU=1MHz), single conversion
19             _BV(ADEN)|_BV(ADIE);           // enable adc, interrupt
20
21   sei();
22
23   ADCSRA |= _BV(ADSC);                     // start first conversion
24   while(1) {
25     if ( x > (4*1024)/VREF ) {
26       PORTD = 0xF0;
27     } else if (x > (3*1024)/VREF ) {
28       PORTD = 0x70;
29     } else if (x > (2*1024)/VREF ) {
30       PORTD = 0x30;
31     } else {
32       PORTD = 0x10;
33     }
34   }
35 }
```

## 18.5 Toepassing automatic trigger mode met timer 0

Code 18.3 toont een voorbeeld met de *auto trigger*-mode en timer 0. De klok van timer 0 is door acht gedeeld en de teller geeft om de 125 prescaled klokslagen een interrupt overflow. Dit laatste is bereikt door de teller op te hogen met 131 ($= 256 - 125$). Voor een systeemklok van 1 MHz is $T_{\text{cpu}}$ 1 $\mu$s. Met formule 13.2, een prescaling $P$ van 8 en een aantal klokslagen $m$ van 125 is de tijd tussen de interrupts:

$$t \;=\; mPT_{\text{cpu}} \;=\; 125 \times 8 \times 1\mu \;=\; 1\,\text{ms}$$

Bit ADATE van het ADCSRA-register is op regel 28 geselecteerd en zet de ADC in de *auto trigger*-mode. Op regel 29 zijn de drie ADTS-bits ingesteld op 100. De ADC wordt getriggerd door de overflow van timer 0.

De timer genereert elke milliseconde een interrupt, die de ADC automatisch start. Als de ADC klaar is, wordt de interruptroutine van regel 7 gestart en krijgt de globale variabele *x* de nieuw gemeten waarde. De interrupt van de timer start niet alleen de AD-conversie, maar start ook de interruptroutine van regel 12. Deze routine hoogt alleen `TCNT0` met 131 op, zodat er precies 125 prescaled klokslagen worden geteld.

**Code 18.3 : Ingangswaarde meten met automatic trigger en timer 0.**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3
 4  #define VREF  5
 5  volatile unsigned int x;
 6
 7  ISR(ADC_vect)
 8  {
 9    x = ADCL | ((unsigned int)ADCH << 8);     // get result
10  }
11
12  ISR(TIMER0_OVF_vect)
13  {
14    TCNT0  += 131;                           // skip 131 clockticks
15  }
16
17  int main(void) {
18    DDRD    = 0xF0;                          // all bits port D outputs
19
20    TCCR0   = _BV(CS01);                     // prescaling clock/8
21    TCNT0   = 131;                           // skip 131 clockticks
22    TIMSK   = _BV(TOIE0);                    // interrupt timer 0 overflow
23
24    ADMUX   = 0x05|                          // channel ADC5 (is pin PA5), right justified
25              _BV(REFS0);                    // internal VCC selected
26    ADCSRA  = _BV(ADPS1)|_BV(ADPS0)|         // prescaling 8 (F_CPU=1MHz)
27              _BV(ADEN)|_BV(ADIE)|           // enable adc, interrupt
28              _BV(ADATE);                    // auto trigger
29    SFIOR   = _BV(ADTS2);                    // auto trigger timer 0 overflow
30
31    sei();
32
33    while(1) {
34      if ( x > (4*1024)/VREF ) {
35        PORTD = 0xF0;
36      } else if (x > (3*1024)/VREF ) {
37        PORTD = 0x70;
38      } else if (x > (2*1024)/VREF ) {
39        PORTD = 0x30;
40      } else {
41        PORTD = 0x10;
42      }
43    }
44  }
```

## 18.6 Toepassing met free running mode

In code 18.4 is de *free running*-mode gebruikt zonder interrupt. Op regel 12 is het ADATE-bit voor de *automatic trigger*-mode geselecteerd. De drie ADTS-bits van het SFIOR-register zijn op regel 12 expliciet laag gemaakt. De AD-conversie wordt dan getriggerd door het ADIF-bit. Alleen de eerste conversie wordt gestart met de toewijzing op regel 15.
De ADC meet voortdurend de ingangswaarde van poort ADC5. Het hoofdprogramma doorloopt continu de **while**-lus. De variabele x wordt elke keer opnieuw berekend uit de huidige waarde van ADCL en ADCH.

**Code 18.4 : Ingangswaarde meten in free running mode zonder interrupt.**

```
1  #include <avr/io.h>
2
3  #define VCC  5
4  unsigned int x;
5
6  int main(void) {
7    DDRD    = 0xF0;                          // all bits port D outputs
8
9    ADMUX   = 0x05|                          // channel ADC5 (is pin PA5), right justified
10             _BV(REFS0);                    // internal VCC selected
11   ADCSRA  = _BV(ADPS1)|_BV(ADPS0)|         // prescaling 8 (F_CPU=1MHz), no interrupt
12             _BV(ADEN)|_BV(ADATE);          // enable adc, auto trigger (free running)
13   SFIOR  &= ~(_BV(ADTS2)|_BV(ADTS1)|_BV(ADTS0));
14
15   ADCSRA |= _BV(ADSC);                     // first conversion
16   while(1) {
17     x = ADCL | ((unsigned int)ADCH << 8);
18     if ( x > (4*1024)/VCC ) {
19       PORTD = 0xF0;
20     } else if (x > (3*1024)/VCC ) {
21       PORTD = 0x70;
22     } else if (x > (2*1024)/VCC ) {
23       PORTD = 0x30;
24     } else {
25       PORTD = 0x10;
26     }
27   }
28 }
```

# 19

# Liquid Crystal Display

Doelstelling

In dit hoofdstuk leer je hoe een karaktergeoriënteerd LCD is opgebouwd, wat HD44780 compatibel is en hoe je een HD44780 compatibel display met een ATmega32 aanstuurt.

Onderwerpen

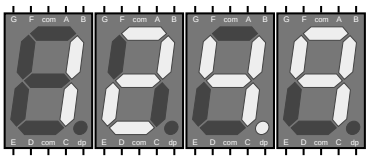
De behandelde onderwerpen zijn:

- De instelling van het contrast van het LCD.
- Het instellen van de achtergrondverlichting.
- De communicatie met het LCD volgens het HD44780-protocol. Daarbij komt aan de orde: de timing bij de HD44780, de *busy flag*, de aansturing van de geheugens van de HD44780, de positionering van tekst op het LCD.
- De LCD-bibliotheek van Peter Fleury.
- De weergave van gehele getallen met behulp van `utoa` en `ultoa`.
- Geformatteerd afdrukken met behulp van `dtostrf` en `sprintf`.

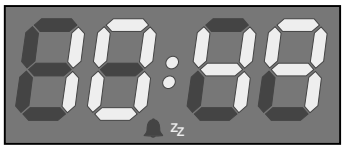
De voorbeelden tonen verschillende mogelijkheden van het LCD en een aantal methoden om het LCD aan te sturen:

- Met 8-bit mode en een *worst case* tijdvertraging.
- Het bewegen van tekst in 8-bit mode.
- Het geformatteerd weergeven van getallen in 4-bit mode.
- Het gebruik van de LCD-bibliotheek van Peter Fleury.

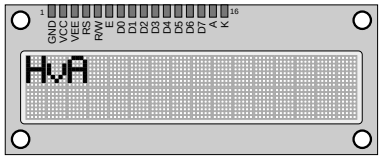
Een nadeel bij het werken met een microcontroller is, dat er standaard geen toetsenbord en beeldscherm bijhoort. Voor het testen is altijd een opstelling nodig met extra elektronica om informatie aan de microcontroller door te geven en om resultaten zichtbaar te maken.
Het weergeven van informatie kan met leds, 7-segmentdisplays, een karaktergeoriënteerd display, een eenvoudige grafische display of met een standaard TFT- of CRT-beeldscherm. Figuur 19.1 toont diverse typen displays.
Leds, ledmatrices en 7-segmentdisplays zijn beperkt wat betreft hun mogelijkheden. Op een 4-digit display kunnen bijvoorbeeld maar vier cijfers worden afgebeeld. Door meerdere ledmatrices of 7-segmentdisplays te combineren, kan meer tekst afgebeeld worden. Lichtkranten bestaan vaak uit vele ledmatrices, maar de aansturing van de verschillende dots is niet eenvoudig.
CRT- en TFT-beeldschermen bevatten heel veel beeldpunten. De aansturing is met een eenvoudige 8-bits microcontroller niet goed mogelijk en over het algemeen niet zinvol.



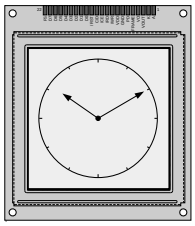
**36-pins 4-digit 7-segmentdisplay**



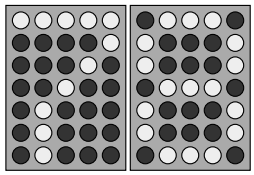
**klantspecifieke LCD**



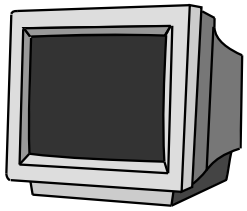

**2 regel × 16 karakters LCD**



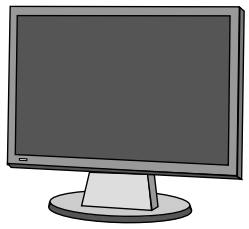
**128×128 pixels grafisch LCD**



**2 karakter 5×7 led matrix**



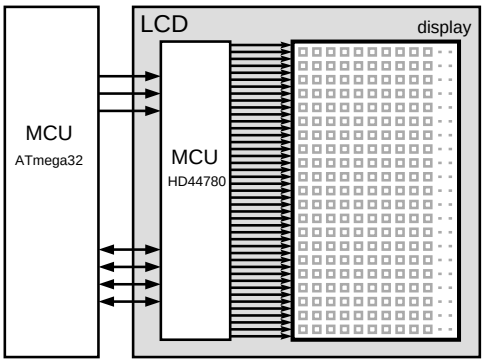
**CRT-beeldscherm (kathodestraalbuis)**



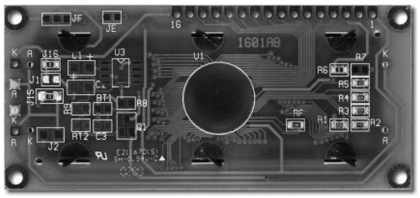
**TFT-beeldscherm**

**Figuur 19.1 : Diverse typen displays.**
Displays zijn te verkrijgen in veel soorten, maten en prijzen. De schaal, waarop deze displays zijn afgebeeld, is willekeurig. Voor microcontrollersystemen zijn vooral de karaktergeoriënteerde en kleinere grafische displays interessant.

Karaktergeoriënteerde displays en de kleinere grafische displays worden veel gebruikt bij microcontrollers. Deze displays bevatten een microcontroller die de pixels van het scherm aansturen, zie figuur 19.2. Deze microcontroller moet met de microcontroller van de applicatie communiceren. Hiervoor zijn een aantal controllijnen en een aantal datalijnen nodig. Er zijn verschillende protocollen beschikbaar. Voor karaktergeoriënteerde displays is dit vaak HD44780 of KS0073. HD44780 is een speciale microcontroller van Hitachi voor karaktergeoriënteerde displays. Bij grafische displays wordt vaak T6963 of KS0108 gebruikt.




**Figuur 19.2 : Communicatie met HD44780.**
De ATmega32 communiceert met de HD44780 die de pixels aanstuurt.



**Figuur 19.3 : Achterkant LCD met HD44780.**
De HD44780 zit onder de zwarte schijf.

De eerste stap als je met een display aan de slag gaat, is dat je moet uitzoeken welke microcontroller het display bevat en dus welk protocol er gebruikt wordt. Er bestaan heel veel verschillende uitvoeringen. Sommige displays zijn compatibel en andere zijn dat niet. Displays met een SED1278 van Epson, KS0066 van Samsung, ST7066 van Sitronix en SPLC780A1 van Sunplus zijn compatibel met de HD44780.

## 19.1 Het karaktergeoriënteerde display op basis van HD44780

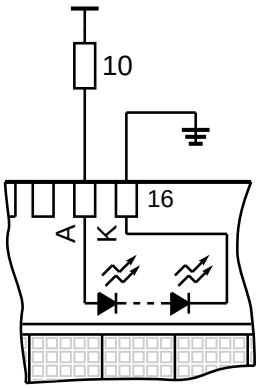
Karaktergeoriënteerde displays op basis van een HD44780 kunnen niet meer dan tachtig karakters aansturen. Deze karakters zijn verdeeld over 1, 2 of 4 regels. Populair zijn displays met 1×16, 1×20, 1×40, 2×16, 2×20, 4×20, 2×40 en 4×40 karakters. Het display met 4 × 40 karakters heeft meer dan tachtig karakters en heeft daarom twee HD44780-microcontrollers: een voor de bovenste twee regels en een voor de onderste twee regels.

Er bestaan displays met twee verschillende karakterformaten, namelijk met karakters van 5 × 8 pixels en van 5 × 10 pixels. De meeste displays hebben karakters met 5 × 8 pixels. De HD44780 wordt bij beide displays gebruikt.

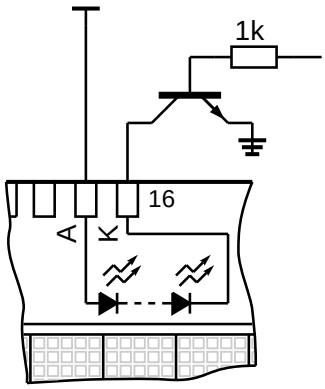
### Achtergrondverlichting (*backlight*)

Een display met een HD44780 heeft veertien of zestien aansluitingen. Een display met achtergrondverlichting (*backlight*) heeft zestien aansluitingen en zonder achtergrondverlichting heeft het veertien aansluitingen. Displays zonder achtergrondverlichting hebben een spiegelende achterkant en reflecteren het omgevingslicht. Het nadeel van deze displays is dat deze niet in een donkere omgeving gebruikt kunnen worden. Displays met achtergrondverlichting zijn vaak helderder en zijn wel in een slecht verlichte omgeving toepasbaar. Natuurlijk verbruiken deze displays wel meer stroom.

Een 4 × 40 display heeft meer dan tachtig karakters en bevat twee HD447780's. Er is een extra aansluiting. Er zijn twee enable-aansluitingen (E1 en E2). De andere control- en datalijnen worden door beide controllers gedeeld.




**Figuur 19.4 : Achtergrondverlichting LCD altijd aan.**




**Figuur 19.5 : Achtergrondverlichting wordt met transistor geschakeld vanuit microcontroller.**

De achtergrondverlichting bestaat uit een of meer in serie geschakelde leds. In figuur 19.4 is de anode (A) verbonden met de voeding en de kathode (K) geaard. Er is een weerstand van 10 Ω gebruikt om de stroom door de leds te beperken. Of deze weerstand nodig is en welke waarde deze weerstand moet hebben, hangt af van het display. Er zijn grote verschillen tussen de spanning $U_{\text{leds}}$ die nodig is en de maximaal toelaatbare stroom $I_{\text{max}}$. Hier is uitgegaan van 4,0 $V$ en 100 mA. De weerstandswaarde $R$ wordt bij een voedingsspanning $U_{\text{dd}}$ van 5,0 V gevonden met:

$$R = \frac{U_{\text{dd}} - U_{\text{leds}}}{I_{\text{max}}} = \frac{5{,}0 - 4{,}0}{0{,}1} = 10\,\Omega$$

De waarde van de weerstand bij de basis hangt sterk af van de keuze van de transistor en van de maximale toelaatbare stroom.
Bestudeer altijd de datasheet voor het elektrisch gedrag van het display.

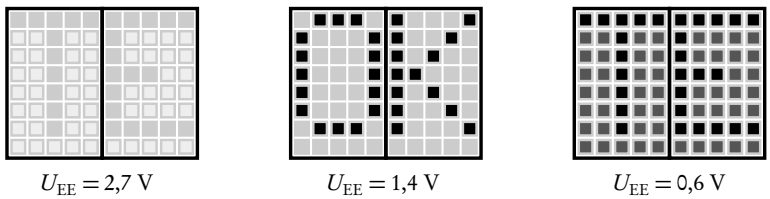
Figuur 19.5 laat zien dat met een transistor de achtergrondverlichting vanuit een microcontroller geschakeld kan worden. Als de basis van de transistor hoog is, geleidt de transistor en is de achtergrondverlichting aan.

### Contrast van het display

Het LCD wordt gevoed via de pinnen VDD en GND. De HD44780S heeft een voedingsspanning van 4,5 - 5,5 V nodig. Voor de HD44780U is dat 2,7 - 5,5 V. De

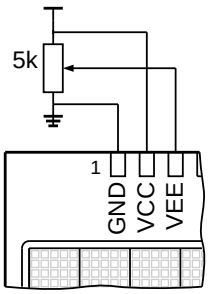
aansluiting VEE regelt het contrast. Bij een lage contrastspanning zijn de pixels, die *aan* zijn, zwart. Bij een te lage spanning zijn de pixels, die *uit* zijn, te donker en is de tekst niet leesbaar. Bij een te hoge spanning zijn de pixels, die *uit* zijn, licht, maar zijn de pixels, die *aan* zijn, ook te licht. De tekst is dan eveneens onleesbaar. Bij een juiste spanning zijn de achtergrondpixels licht en de pixels die *aan* zijn donker. Figuur 19.6 laat dit voor verschillende spanningen zien.

De contrastspanning is voor elke type display anders en hangt sterk af van het omgevingslicht en of de achtergrondverlichting aan staat.




**Figuur 19.6** : **Het effect op het contrast bij een LCD.** Links is de contrastspanning te hoog, in het midden precies goed en rechts te laag.

De potmeter in figuur 19.7 regelt het contrast. De spanning VEE is dan instelbaar tussen de voedingsspanning VCC en GND.




**Figuur 19.7** : **De voeding en de regeling van het contrast van het LCD.** Met de potmeter is het contrast van het LCD instelbaar.

### Communicatie met HD44780

Voor de communicatie zijn elf aansluitingen beschikbaar: drie besturingslijnen en acht datalijnen. Tabel 19.1 geeft een overzicht van alle aansluitingen. Via de datalijnen worden gegevens naar het dataregister of het instructieregister van de HD44780 gestuurd.
Als het signaal `RS` (*Register Select*) laag is, wordt het commando in het instructieregister gezet en als `RS` hoog is, wordt de informatie in het dataregister geplaatst. Er kunnen niet alleen gegevens naar de HD44780 geschreven worden, maar ook gegevens worden uitgelezen. Als signaal `R/W` hoog is, wordt er gelezen (*Read*) en als `R/W` laag is, wordt er geschreven (*Write*). Sommige applicaties lezen nooit informatie uit het LCD en hebben `R/W` vast aan aarde gelegd.

Er zijn acht datalijnen. De informatie kan verstuurd worden in een 8-bit mode of in een 4-bit mode. Deze mode wordt ingesteld bij het initialiseren van het LCD. In de 8-bit mode wordt de informatie in bytes overgestuurd. Signaal `D7` bevat het meest significante bit en `D0` het minst significante bit.
Bij de 4-bit mode zijn minder datalijnen en dus minder aansluitingen van de microcontroller nodig. Deze aansluitingen zijn dan beschikbaar voor andere toepassingen. De aansluitingen `D3`, `D2`, `D1` en `D0` van LCD blijven open. De datalijnen `D7`,

**Tabel 19.1 : Overzicht aansluitingen LCD.**

| Pin | Naam | Functie | Omschrijving |
|---|---|---|---|
| 1 | GND | ground | 0 V |
| 2 | VCC | voedingsspanning | 5 V |
| 3 | VEE | contrast | 0 - 5 V |
| 4 | RS | Register Select | 0 = instructie, 1 = data (tekst) |
| 5 | R/W | Read/Write | 0 = write, 1 = read |
| 6 | E | Enable | bij 1 naar 0 overgang wordt data/instructie overgezet |
| 7 | D0 | Data (LSB) | niet gebruikt in 4-bit mode |
| 8 | D1 | Data | niet gebruikt in 4-bit mode |
| 9 | D2 | Data | niet gebruikt in 4-bit mode |
| 10 | D3 | Data | niet gebruikt in 4-bit mode |
| 11 | D4 | Data | |
| 12 | D5 | Data | |
| 13 | D6 | Data | |
| 14 | D7 | Data (MSB) | |
| 15 | A | Anode | achtergrondverlichting (niet bij alle displays) |
| 16 | A | Kathode | achtergrondverlichting (niet bij alle displays) |

Een *nibble* is een groep van vier bits.

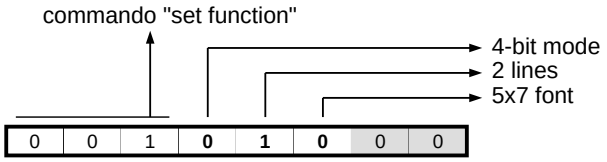
Een *byte* is een groep van acht bits. Een *byte* bestaat uit twee *nibbles*.

D6, D5 en D4 zijn verbonden met de microcontroller. Deze stuurt eerst de meest significante 4-bits van de te verzenden informatie naar het LCD en daarna de minst significante 4-bits. De HD44780 leest eerst het hoogste *nibble*, daarna het laagste *nibble* en maakt hier weer een *byte* van.

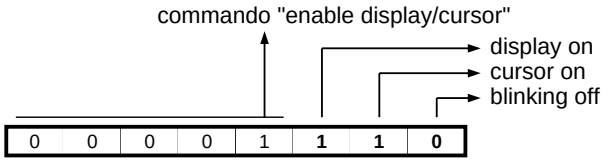
**Tabel 19.2 : Overzicht instructieset HD44780.** Er zijn acht instructies voor het besturen van het display (RS=0 en R/W=0). Er is een instructie voor het versturen van karakters (RS=1 en R/W=0). Er zijn twee instructies die lezen (R/W=1): één instructie om de *busy flag* te lezen en één om het karakter op de huidige positie van het display te lezen. Sommige instructies hebben een of meer extra bits. In de laatste drie kolommen wordt de betekenis van deze bits toegelicht.

| RS | R/W | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Instructie | Bit | Bit is 1 | Bit is 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | clear display | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | - | move cursor to home position | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | ID | S | move cursor | ID | Increment by 1 | Decrement by 1 |
| | | | | | | | | | | | S | display Shift on | display Shift off |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | enable display/cursor | D | Display On | Display off |
| | | | | | | | | | | | C | Cursor on | Cursor off |
| | | | | | | | | | | | B | Blink on | Blink off |
| 0 | 0 | 0 | 0 | 0 | 1 | DC | RL | - | - | shift display/cursor | DC | Display shift | Cursor shift |
| | | | | | | | | | | | RL | shift Right | shift Left |
| 0 | 0 | 0 | 0 | 1 | DL | N | F | - | - | function set | DL | 8-bit mode | 4-bit mode |
| | | | | | | | | | | | N | 2 Lines | 1 Line |
| | | | | | | | | | | | F | 5×10 Font | 5×8 Font |
| 0 | 0 | 0 | 1 | a | a | a | a | a | a | move cursor to CGRAM | a is 6-bits address | | |
| 0 | 0 | 1 | a | a | a | a | a | a | a | move cursor to display | a is 7-bits address | | |
| 1 | 0 | d | d | d | d | d | d | d | d | write character to display | d is 8-bits data | | |
| 0 | 1 | BF | - | - | - | - | - | - | - | read busy flag | BF | Busy Flag is on | Busy Flag is off |
| 1 | 1 | d | d | d | d | d | d | d | d | read character from display | d is 8-bits data | | |

Tabel 19.2 geeft een overzicht van de instructies bij de HD44780. Er zijn acht besturingscommando's voor het display. Bij sommige instructies hebben een aantal bits een bijzondere betekenis. Zo kan met een enkele instructie zowel het display, als de cursor en het knipperen (*blinking*) aan en uit gezet worden.
Figuur 19.8 toont de instructie 0x28 die de 4-bit mode selecteert en die aangeeft dat het display twee regels heeft en dat er een 5 × 8-karakterset is. Figuur 19.9 laat




**Figuur 19.8 : HD44780-instructie 0x28.**




**Figuur 19.9 : HD44780-instructie 0x0E.**

zien dat met de instructie 0x0E het display en de cursor aangezet worden en dat de cursor niet knippert.
Voor de communicatie met de HD44780 is het enable-signaal E essentieel. Er kan informatie van en naar de HD44780 verstuurd worden. Om informatie van de HD44780 te lezen moet R/W hoog zijn en moet E van laag naar hoog gaan. De gegevens komen uit het instructieregister als RS laag is en uit het dataregister als RS hoog is.
Om gegevens naar de HD44780 te schrijven moet R/W laag zijn en moet E van hoog naar laag gaan. Als RS laag is, worden de data in het instructieregister gezet en als RS hoog is, komt het in het dataregister te staan.

### De timing bij de HD44780

De signalen RS en R/W moeten ingesteld zijn voordat E hoog gemaakt wordt. De setuptijd voor deze signalen is $T_{asuE}$. Figuur 19.10 geeft het tijdsdiagram voor het schrijven van informatie. Het signaal R/W moet laag en het signaal RS moet dan hoog zijn bij het versturen van data (tekst) en laag bij het versturen van instructies. De databyte moet gezet zijn voor dat E laag gemaakt wordt. De setuptijd is $T_{dsuE}$. Het enable-signaal E moet minimaal een periode $T_{pwE}$ hoog zijn en tussen twee enable-pulsen moet minimaal $T_{cycE}$ zitten. In tabel 19.3 en 19.4 staan een aantal tijdskarakteristieken uit de datasheet. Voor verschillende displays kunnen deze tijden anders zijn.

**Tabel 19.3 : Timing bij de HD44780 bij 4,5 – 5,5 V.**

| Tijd | Waarde |
|---|---|
| $T_{asuE}$ | 40 ns |
| $T_{dsuE}$ | 80 ns |
| $T_{pwE}$ | 230 ns |
| $T$cycE | 500 ns |

**Tabel 19.4 : Timing bij de HD44780 bij 2,7 – 4,5 V.**

| Tijd | Waarde |
|---|---|
| $T_{asuE}$ | 60 ns |
| $T_{dsuE}$ | 195 ns |
| $T_{pwE}$ | 450 ns |
| $T$cycE | 1000 ns |

Voorbeelden op het internet maken het enable-signaal E vaak één instructie hoog en bij de volgende instructie direct weer laag:

```
PORTD = 'A';           // databyte is 'A'
PORTB = PORTB|0x02;    // bit 1 port B (signal E) high
PORTB = PORTB&0xFD;    // bit 1 port B (signal E) low
```

Het effect is dat E dan twee klokslagen hoog is. Dit is prima voor klokfrequenties lager dan 4 MHz. Bij hogere frequenties is E niet lang genoeg hoog. Voor een frequentie van 10 MHz is de klokperiode 100 ns en is E slechts 200 ns hoog. Dat is minder dan de 230 ns en de 450 ns, die nodig zijn bij respectievelijk een voedingsspanning van 5 V en 2,7 V. Een oplossing voor dit probleem is het toevoegen van een of meer *no operating*-instructies.

```
PORTD = 'A';           // databyte is 'A'
PORTB = PORTB|0x02;    // bit 1 port B (signal E) high
asm volatile ("nop");
asm volatile ("nop");
PORTB = PORTB&0xFD;    // bit 1 port B (signal E) low
```

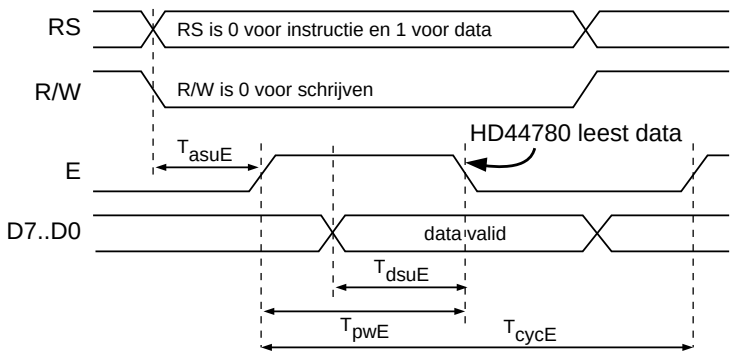
Een andere oplossing is de data pas toe te kennen nadat E hoog gemaakt is. Dit maakt de puls twee klokslagen langer. Mits de setuptijd $T_{dsuE}$ gehaald wordt, kan dit een goede oplossing zijn.

```
PORTB = PORTB|0x02;    // bit 1 port B (signal E) high
PORTD = 'A';           // databyte is 'A'
PORTB = PORTB&0xFD;    // bit 1 port B (signal E) low
```

Een veilige oplossing is om altijd een vertraging $T_{\text{pwE}}$ toe te voegen.

```
PORTB = PORTB|0x02;    // bit 1 port B (signal E) high
PORTD = 'A';           // databyte is 'A'
_delay_us(0.5);        // delay TpwE (worst case 450 ns)
PORTB = PORTB&0xFD;    // bit 1 port B (signal E) low
_delay_us(0.5);        // delay TcycE-TpwE
```

Het signaal `E` is in ieder geval 0,5 $\mu$s plus vier klokslagen hoog en $T_{\text{dsuE}}$ wordt zeker gehaald. De extra vertraging nadat `E` laag is, zorgt ervoor dat twee pulsen minimaal $T_{\text{cycE}}$ na elkaar komen.




**Figuur 19.10 : Het tijdsdiagram voor het schrijven van informatie naar het LCD.** Eerst krijgen de signalen `RS` en `R/W` hun waarde. Voor het schrijven moet `R/W` laag zijn. `RS` moet hoog zijn voor het schrijven van data en laag voor het schrijven van instructies. Nadat `E` laag wordt, leest de HD44780 de `D7..D0`.

**Tabel 19.5 : Tijdvertragingen van de HD44780.**
C staat voor cursor en D voor display.

| Command | Delay |
|---|---|
| clear display | 1,52 ms |
| move C to home | 37 $\mu$s |
| move C | 37 $\mu$s |
| enable D/C | 37 $\mu$s |
| shift D/C | 37 $\mu$s |
| function set | 37 $\mu$s |
| move C CGRAM | 37 $\mu$s |
| move C display | 37 $\mu$s |
| write character | 37 $\mu$s |
| read busy flag | 37 $\mu$s |
| read character | 37 $\mu$s |

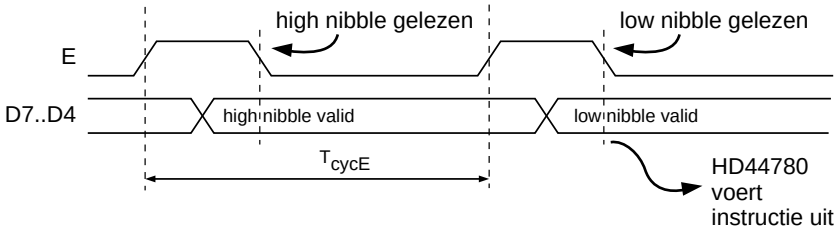
Voor de aansturing van de pixels gebruikt het display een interne oscillator. De oscillatorfrequentie (270 kHz) bepaalt de snelheid waarmee instructies uitgevoerd worden en waarmee tekst op het display gezet wordt. Tabel 19.5 geeft een overzicht van deze vertragingstijden. Displays hebben soms een andere interne oscillator en daarmee ook andere tijdvertragingen. Nadat de databyte verzonden is, moet er minimaal 37 $\mu$s gewacht worden. Met een vertraging van 1,5 ms werkt de code ook bij het leegmaken van de display.

```
PORTB = PORTB|0x02;    // bit 1 port B (signal E) high
PORTD = 'A';           // databyte is 'A'
_delay_us(0.5);        // delay TpwE (worst case 450 ns)
PORTB = PORTB&0xFD;    // bit 1 port B (signal E) low
_delay_us(1520);       // delay display (worst case 1.52 ms)
```

In de 4-bit mode kunnen de hoge en de lage *nibbles* direct na elkaar verstuurd worden. Direct nadat de HD44780 het lage *nibble* gelezen heeft, wordt de instructie uitgevoerd. Figuur 19.11 laat dit zien. De twee pulsen moeten wel minimaal $T_{\text{cycE}}$ (= 1000 ns) uit elkaar liggen.

### De busy flag

De HD44780 maakt bij het uitvoeren van een instructie intern een *busy flag* hoog en maakt deze weer laag als de instructie afgerond is. Sommige applicaties gebruiken de tijdvertragingen uit tabel 19.5 om de signalen op het juiste moment




**Figuur 19.11 : Het schrijven van een instructie in 4-bit mode.** De HD44780 leest de *nibbles* nadat E laag gemaakt is.

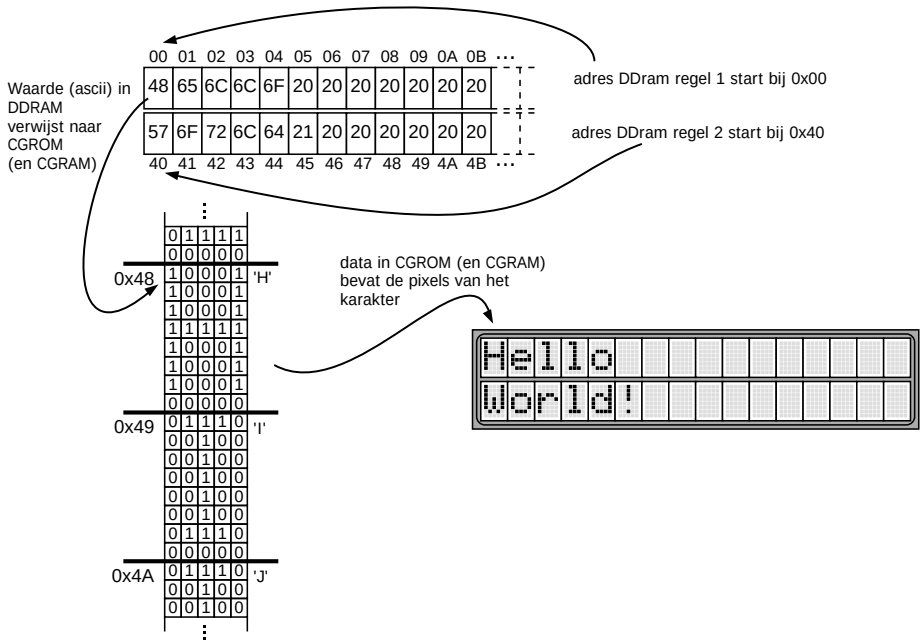
te veranderen. Andere applicaties gebruiken de *busy flag* om te bepalen of de volgende instructie doorgegeven kan worden. Het nadeel van de methode met tijdvertragingen is dat deze tijden afhankelijk zijn van het gebruikte display. Bij de methode met de *busy flag* kan de communicatie vastlopen als om een een of andere reden de *busy flag* niet laag wordt.

## De geheugens van de HD44780 en de aansturing van het display

De HD44780 heeft twee RAM-geheugens: een CGRAM (*Character Generation RAM*) en een DDRAM (*Display Data RAM*). In het CGRAM is plaats voor eigen karakters. Bij een 5 × 8-font is plaats voor acht karakters.

Het maximaal aantal karakters bij een HD44780 display is tachtig.

In het DDRAM is plaats voor tachtig 8-bits adressen. Elk adres verwijst naar een geheugenplaats in het CGRAM of het CGROM (*Character Generation ROM*). Het CGROM bevat de data van 208 voorgedefinieerde karakters van 5 × 8 pixels en 32 voorgedefinieerde karakters van 5 × 10 pixels.




**Figuur 19.12 : Het CGROM en het DDRAM.** Het DDRAM bevat de ASCII-waarden van de af te beelden tekst. Het CGROM (en het CGRAM) bevat de pixelinformatie van de karakterset. Het adres van een karakter komt overeen met de betreffende ASCII-waarde.

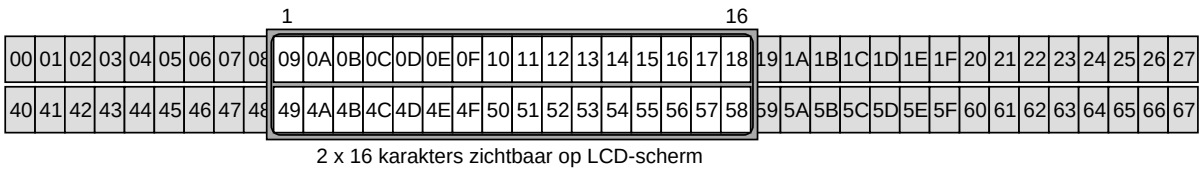
In figuur 19.12 bevat de eerste regel van het LCD de tekst "`Hello`" en de tweede regel de tekst "`World!`". De niet gebruikte posities zijn opgevuld met spaties. Het display heeft twee regels van zestien karakters. Het DDRAM bevat de ASCII-waarden van de af te beelden karakters. Het adres van het DDRAM van de eerste regel start bij `0x00` en dat van de tweede regel start bij `0x40`. De ASCII-waarden in het DDRAM zijn in feite adressen van het CGRAM en het CGROM. Het eerste hokje van het DDRAM bevat de ASCII-waarde `0x48` en is een verwijzing naar adres `0x48` van het CGROM met de pixelbeschrijving van het karakter '`H`'. De HD44780 zorgt er voor dat deze informatie op de juiste plaats op het display komt te staan.

In bijlage J staat meer informatie over ASCII en in tabel J.1 staat een overzicht van alle 7-bits ASCII-waarden.

Het schrijven van tekst naar het display is dus niets anders dan het schrijven van ASCII-waarden naar het DDRAM. Voor eigen karakters (*user defined characters*) moeten zelf ontworpen pixelpatronen in het CGRAM worden gezet. De pixelpatronen in het CGROM kunnen natuurlijk niet worden aangepast. Wel zijn er displays te koop met andere fonts. De HD44780U-A00 komt het meest voor en bevat de standaard ASCII-tekens van `0x20` tot en met `0x7D` en een groot aantal Japanse karakters, enkele Griekse letters (α), speciale Europese karakters (ö) en enkele andere symbolen (√). De karakterset van de HD44780U-A02 komt voor een groot deel overeen met de ASCII-ISO8859. Dat zijn de standaard ASCII-tekens van `0x20` tot en met `0x7D` aangevuld met heel veel Europese karakters (ö, Æ, Å). Een deel van de ISO8859-symbolen is vervangen door Griekse en cyrillische letters.

De datasheet geeft een compleet overzicht van beide karaktersets. Veel programmeurs gebruiken alleen de 7-bits ASCII-waarden, die in beide sets liggen tussen `0x20` tot en met `0x7D`.


**Figuur 19.13 : De adressering van het geheugen en de relatie met karakters die zichtbaar zijn bij een 2x16 display.**

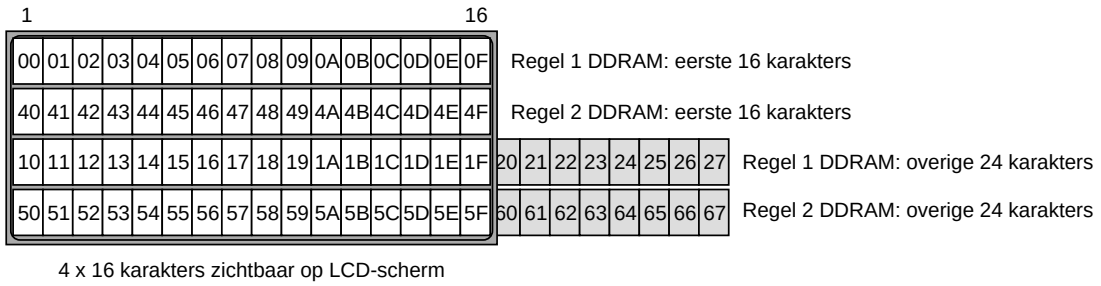



**Figuur 19.14 : De adressering van het geheugen bij een verschuiving en de relatie met karakters die zichtbaar zijn bij een 2x16 display.**

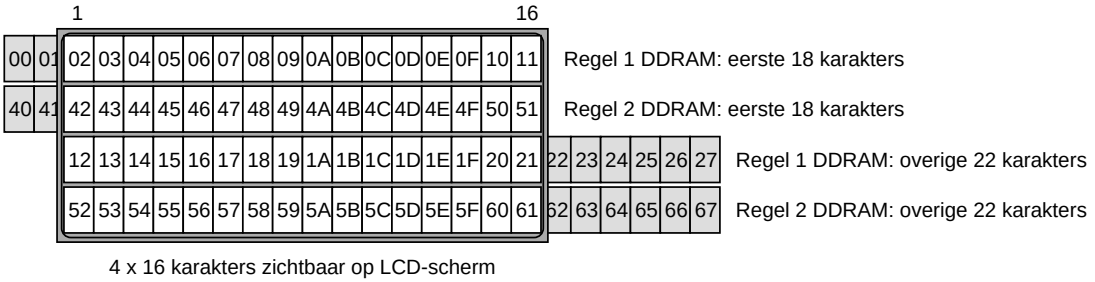
Met de schuiffunctie kan de beginpositie van het LCD-scherm worden gewijzigd. Figuur 19.13 toont het DDRAM en een 2 × 16-display. De eerste regel start bij adres `0x00` en en de tweede regel start bij adres `0x40`. In figuur 19.14 is het LCD negen posities naar rechts geschoven ten opzichte van het DDRAM. De verschuiving geldt altijd voor beide regels.

Figuur 19.15 toont de situatie bij een 4 × 16-display. De regels starten bij de adressen `0x00`, `0x40`, `0x10` en `0x50`. Regel 1 loopt in feite door op regel 3 en regel 2 loopt door op regel 4. In figuur 19.16 is het LCD twee posities verschoven. Regel 1 loopt nu van `0x02` tot `0x11` en regel 3 begint bij adres `0x12`. De karakters die voor het schuiven aan het begin van regel 3 stonden staan na het schuiven aan het einde van regel 1.

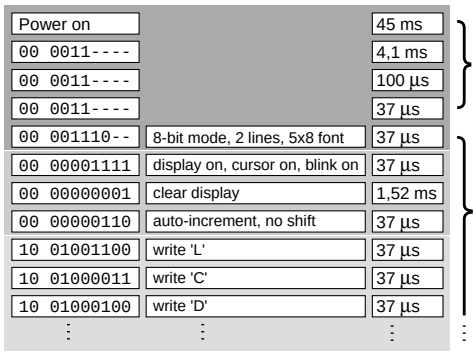
De adressering kan bij verschillende displays anders zijn. Bestudeer de datasheet van de fabrikant dus goed.



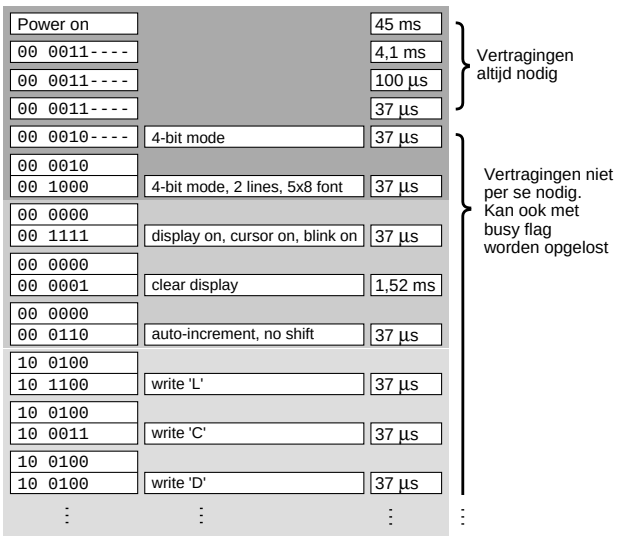

**Figuur 19.15 : De adressering van het geheugen en de relatie met de karakters die zichtbaar zijn bij een 4x16 display.**




**Figuur 19.16 : De adressering van het geheugen bij een verschuiving en de relatie met de karakters die zichtbaar zijn bij een 4x16 display.**




**Figuur 19.17 : Initialisatie bij de 8-bit mode.**

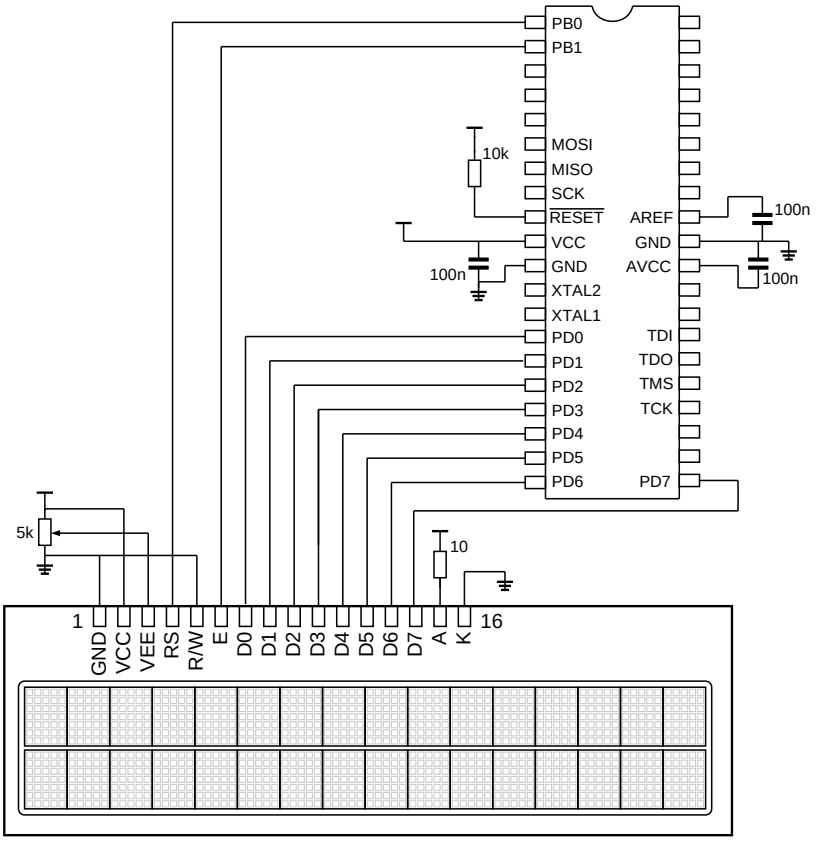



**Figuur 19.18 : Initialisatie bij de 4-bit mode.**

## 19.2 Toepassing LCD in 8-bit mode en met tijdvertraging

Figuur 19.19 geeft een compleet schema voor de 8-bit mode. De datalijnen zijn aangesloten op poort D van de microcontroller. Pin 0 van poort B is aangesloten op de RS-pin en pin 1 van poort B is aangesloten op de E-pin van het LCD. Dit voorbeeld gebruikt een *worst case* tijdvertraging en heeft de leesfunctie niet nodig. De aansluiting R/W van het LCD is verbonden met aarde.

Voordat het LCD gebruikt wordt, moet het display geïnitialiseerd worden. Dat betekent dat er een aantal commando's in een bepaalde volgorde naar het display




**Figuur 19.19** : **Het schema voor de aansturing van het LCD met acht datalijnen.** De status van het LCD wordt niet gelezen. Het `R/W`-signaal is daarom altijd laag.

gestuurd moeten worden. Figuur 19.17 geeft de initialisatie voor de 8-bit mode en figuur 19.18 die voor de 4-bit mode. De acties met de donkergrijze achtergrond zijn verplicht voor de initialisatie. De diverse commando's hebben een eigen verwerkingstijd, zie ook tabel 19.5. Het volgende commando mag pas nadat deze tijd verstreken is gegeven worden, anders wordt het genegeerd. Dit voorbeeld gebruikt een *worst case* tijdvertraging.

De functie `lcdwrite` werkt bij 5 V alleen goed voor frequenties lager dan 5 MHz. Bij hogere frequenties is het enable-signaal `E` niet lang genoeg hoog en is de databyte niet lang genoeg stabiel voordat `E` laag wordt.

De vertraging van 1,52 ms is de grootste vertraging uit tabel 19.5. Het volledig vullen van een display met tachtig karakters duurt minimaal 0,12 s. Veel voorbeelden gebruiken een *worst case* vertraging van 5 ms. Het vullen duurt dan 0,4 s. Een snellere oplossing is 37 $\mu$s te gebruiken en bij *clear display* een vertraging van 1,52 ms toe te voegen.

Het programma staat in code 19.1 en schrijft met `lcdwrite` informatie naar het display. De functie heeft twee ingangsvariabelen. Ingang `d` is het karakter dat afgebeeld wordt of het commando dat gegeven wordt. Ingang `type` geeft aan of er een instructie of dat er een karakter naar het display wordt gestuurd. Deze variabele kan twee waarden hebben, namelijk `PORTB|0x01` en `PORTB&0xFE`. Deze variabele wordt toegekend aan `PORTB` en maakt pin `0`, die aangesloten is op de `RS`-pin van het display, hoog of laag. Vervolgens wordt pin 1 van poort B, het enable-signaal van het display, hoog gemaakt. Hierna wordt de databyte `d` naar poort D geschreven en wordt het enable-signaal weer laag gemaakt. Tenslotte wordt er 1,52 ms gewacht.

Met de functie `LCDwrite` zijn twee macro's gedefinieerd: een macro `lcdcommand`, die een instructie naar het display stuurt, en een macro `lcdputc`, die een karakter schrijft.

Op regel 31 van dit voorbeeld staat een functie `initlcd` die het LCD initialiseert. Deze functie definieert eerst de pinnen `0` en 1 van poort B en poort D als uit-

**Code 19.1 : Aansturing LCD met acht datalijnen en een *worst case* tijdvertraging.**

```
16  #include <avr/io.h>
17  #include <util/delay.h>
18
19  #define lcdcommand(d)  (lcdwrite((d),PORTB&0xFE));  // RS low
20  #define lcdputc(d)     (lcdwrite((d),PORTB|0x01));  // RS high
21
22  void lcdwrite(char d, char type)
23  {
24    PORTB = type;         // RS low or high
25    PORTB = PORTB|0x02;   // make E high
26    PORTD = d;            // assign data
27    PORTB = PORTB&0xFD;   // make E low
28    _delay_us(1520);      // wait 1,52 ms worst case delay
29  }
30
31  void initlcd(void)
32  {
33    DDRD = 0xFF;          // port D: 8-bit data
34    DDRB = DDRB|0x03;     // pin 0 and 1 port B: RS en E
35    _delay_ms(15);
36    lcdcommand(0x38);
37    lcdcommand(0x38);
38    lcdcommand(0x38);
39    lcdcommand(0x38);     // 8-bits, 2 lines, 5x8 font
40    lcdcommand(0x0C);     // display on, cursor off, blink off
41    lcdcommand(0x06);     // move cursor right
42    lcdcommand(0x01);     // clear display
43  }
44
45  int main(void)
46  {
47    initlcd();            // initialization LCD
48
49    lcdputc('L');         // write text
50    lcdputc('C');
51    lcdputc('D');
52
53    while (1) {
54      asm volatile ("nop");
55    }
56  }
```

gang. Vervolgens initialiseert deze functie het LCD in de 8-bit mode volgens het diagram van figuur 19.17.
Het hoofdprogramma roept eerst de initialisatiefunctie `initlcd` aan, stuurt daarna de karakters `'L'`, `'C'` en `'D'` naar het scherm en komt dan in een oneindige wachtlus.

## 19.3 Toepassing met bewegende tekst

Het hoofdprogramma uit code 19.1 zet met `putchar` drie karakters op het scherm. Code 19.2 gebruikt een functie `lcdputs` om een complete string naar het scherm te schrijven.

Bovendien beweegt in dit programma de tekst op het display. Dit wordt bereikt door de beginpositie aan te passen. De variabele `i` varieert van `0` tot 14. De tekst in regel 1 komt op positie `i` te staan en de tekst op regel 2 komt op positie `0x4F-i`. Positie `0x4F` is bij een 2 × 16 display de laatste positie van regel 2. Locatie `0x4F-i` bevindt zich `i` posities van het einde van regel 2.
De tekst in regel 1 wordt rechts aangevuld (`lcdcommand(0x06)`) en de tekst op regel 2 wordt links (`lcdcommand(0x04)`) aangevuld. Het effect is dat de tekst op regel 1 van links naar rechts beweegt en die op regel 2 van rechts naar links beweegt.

**Code 19.2 : Aansturing LCD in 8-bit mode met bewegende tekst.** Regel 1 tot en met 29 is identiek met code 19.1.

```
30  void lcdputs(char *s)
31  {
32    char c;
33
34    while ( (c = *s++) ) {
35      lcdputc(c);
36    }
37  }
38
39  int main(void)
40  {
41    unsigned int i=0;
42
43    initlcd();
44
45    while (1) {
46      lcdcommand(0x01);           // clear display
47      lcdcommand(0x6);            // cursor direction right
48      lcdcommand(1<<7|i);         // move to DDRAM location i
49      lcdputs("LCD");
50      lcdcommand(0x4);            // cursor direction left
51      lcdcommand(1<<7|(0x4F-i));  // move to DDRAM location 0x4F-i
52      lcdputs("dcl");
53      _delay_ms(1000);
54      if (++i==14) i=0;
55    }
56  }
```

De functie `lcdputs` heeft de vorm die meestal bij een `puts` wordt gebruikt.
Dit kan korter:

```
void lcdputs(char *s)
{
  while(*s)
    lcdputc(*s++);
}
```

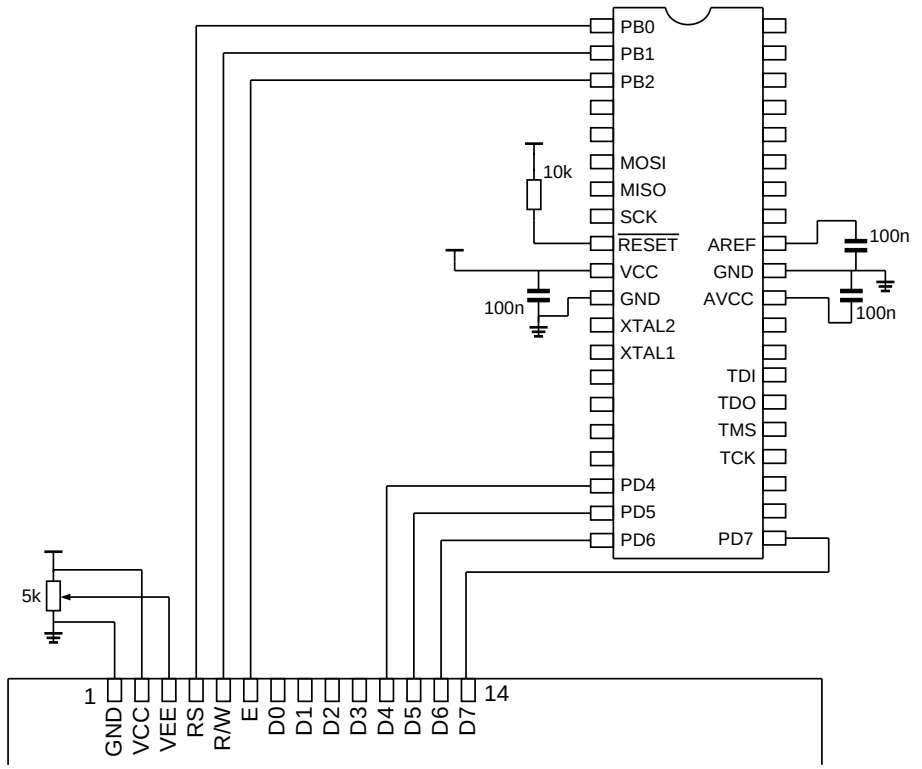
Hapsim is een simulator die gemaakt is door Helmut Wallner en die met AVR-studio gebruikt kan worden. Deze simulator kent een HD44780U compatibel LCD, een set drukknoppen, een set leds, een 4 × 4-toetsenbord en een terminal om de U(S)ART of de TWI ($I^2C$) te simuleren. Figuur 19.20 toont het display van de simulatie van code 19.2 voor de waarden `i=2` en `i=12`. Het LCD van de Hapsim-simulator kent geen tijdvertraging en de *busy flag* is ook niet geïmplementeerd. Een programma dat goed werkt bij de simulatie, hoeft bij een echt display dus niet correct te zijn. Hapsim kan de besturingssignalen `RS`,`R/W` en `E` met elke pin van de microcontroller verbinden. De datalijnen zijn gegroepeerd in *nibbles*. Deze *nibbles* moeten met een hoog of laag *nibble* van een bepaalde poort aan de microcontroller worden aangesloten. Bij de 4-bit mode kunnen de datalijnen `D7`, `D6`, `D5` en `D4` bijvoorbeeld verbonden worden met de pinnen `3`, `2`, `1` en `0` van poort C of met de pinnen `7`, `6`, `5` en `4` van poort A.

De site van Helmut Wallner, de maker van Hapsim, is:
http://www.helmix.at/


**Figuur 19.20 : Twee schermafdrukken van de simulatie met Hapsim van code 19.1.** Links voor `i=2` en rechts voor `i=12`.

## 19.4 Toepassing in de 4-bits mode en met de busy flag

Dit voorbeeld gebruikt het display in 4-bit mode. Het programma leest de *busy flag* en controleert of het volgende commando verstuurd kan worden. Het schema staat in figuur 19.21 en bevat vier datalijnen en drie controllijnen. De besturingssignalen `RS`, `R/W` en `E` zijn verbonden met respectievelijk pin `0`, pin `1` en pin `2` van poort B. De datalijnen `D7`, `D6`, `D5` en `D4` zijn verbonden met de pinnen `7`, `6`, `5` en `4` van poort D. Dit display heeft geen kathode en anode en dus geen achtergrondverlichting.




**Figuur 19.21 : Het schema voor de aansturing van het LCD met vier datalijnen.** Het signaal `R/W` is aangesloten omdat in dit voorbeeld de *busy flag* wordt gebruikt. Het display heeft veertien aansluitpinnen en dus geen achtergrondverlichting.

Code 19.3 zet steeds twee opeenvolgende getallen uit de reeks van Fibonacci op het display. Als de waarde groter wordt dan 16-bits, start de reeks opnieuw met 1 en 1. De reeks van Fibonacci is besproken in paragraaf 14.1. Figuur 19.22 geeft twee schermafdrukken van een simulatie met Hapsim.


**Figuur 19.22 : Twee schermafdrukken van de simulatie van code 19.3.** De linker afdruk geeft de getallen van Fibonacci 34 en 55. De rechter afdruk geeft 28657 en 46368. Deze laatste waarde is het grootste getal van Fibonacci dat in een 16-bits getal past.

**Uitleg code 19.3 regel 6-14**

Met **`#define`**'s worden zes macro's gedefinieerd, die de besturingssignalen `RS`, `R/W` en `E` hoog of laag maken. Verder zijn er twee macro's `lcdcommand` en `lcdputc` die respectievelijk een instructie of een karakter naar het display sturen. Deze macro's gebruiken de functie `lcd4write`.

**Uitleg code 19.3 regel 16-47**

De ingangsparameter `d` van de functie `lcd4write` bevat de databyte die naar het display geschreven wordt en de parameter `rs` geeft aan of het een instructie (`0`) is of dat het een karakter (1) is. Ze zijn allebei van het type `uint8_t`. De functie bestaat uit twee delen: het wachten totdat de *busy flag* laag is en het schrijven van de data.
Omdat eerst de *busy flag* gelezen wordt, maken we de datalijnen ingang (`DDRD &= 0x0F`), `RS` laag en `R/W` hoog. De **`do-while`** leest eerst het hoge *nibble*, daarna het lage *nibble* en blijft dit doen zolang de *busy flag* hoog is.
Voor het schrijven worden de datalijnen opnieuw uitgang (`DDRD |= 0xF0`) gemaakt en `R/W` laag gemaakt. Vervolgens wordt eerst het hoge en daarna het lage *nibble* van `d` verstuurd.

**Uitleg code 19.3 regel 49-54**

Functie `lcd8write` wordt alleen gebruikt bij de initialisatie om instructies naar het display te schrijven. Deze functie heeft een *worst case* vertragingstijd van 1,52 ms Voordat deze functie aangeroepen wordt, moeten het `RS`- en het `R/W`-signaal laag zijn.

**Uitleg code 19.3 regel 56-72**

De functie `initlcd` bestaat uit drie delen. Eerst worden de pinnen van de data- en control lijnen uitgang gemaakt en worden de `RS`-lijn en de `R/W`-lijn laag gemaakt.
Vervolgens worden met de functie `write8lcd` in de 8-bit mode een aantal instructies naar het LCD gestuurd. Alleen het hoge *nibble* is daarbij relevant.
Nadat het display met het algoritme van figuur 19.18 in de 4-bit mode is gezet, verstuurt de functie met `lcdcommand` de overige instellingen. Deze macro doet dat in de 4-bit mode met de functie `lcd4write`.

**Uitleg code 19.3 regel 74-81**

De functie `lcdputs` komt overeen met `lcdputs` uit code 19.2 en drukt de string, waar pointer `s` naar wijst, af. De end-of-string (`'\0'`) wordt niet afgedrukt.

**Uitleg code 19.3 regel 83-110**
`utoa()`
`UINT_MAX`

Voor het berekenen van de getallen van Fibonacci gebruikt het hoofdprogramma drie 16-bits *unsigned* integers (`uint16_t`): `f2` bevat de laatst en `f1` de voorlaatst berekende waarde. De variabele `h` is een hulpvariabele. De reden dat `uint16_t` gebruikt is en niet **`unsigned int`** is dat nu duidelijk is dat het altijd om een 16-bits getal gaat. De variabele `f1` en `f2` hebben als startwaarde 1. Nadat het LCD geïnitialiseerd is, komt het programma in een oneindige lus. Deze maakt het scherm leeg en schrijft vervolgens `f1` op de eerste regel en `f2` op de tweede regel op het display. Daartoe zet het de *unsigned* integers met de functie `utoa` om naar een alfanumerieke string en stuurt deze string naar het display met `lcdputs`.

**Code 19.3 : Weergeven van getallen van Fibonacci op LCD met 4-bit mode.**

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <limits.h>

#define RShigh()  PORTB = PORTB|0x01
#define RSlow()   PORTB = PORTB&0xFE
#define RWhigh()  PORTB = PORTB|0x02
#define RWlow()   PORTB = PORTB&0xFD
#define Ehigh()   PORTB = PORTB|0x04
#define Elow()    PORTB = PORTB&0xFB

#define lcdcommand(d)  (lcd4write((d),0))
#define lcdputc(d)     (lcd4write((d),1))

void lcd4write(uint8_t d, uint8_t rs)
{
  uint8_t x;

  // read and test busy flag
  DDRD &= 0x0F;
  RSlow();
  RWhigh();
  do {
    Ehigh();            _delay_us(0.23);
    x = (PIND&0xF0);
    Elow();             _delay_us(0.5);
    Ehigh();            _delay_us(0.23);
    x |= (PIND&0xF0) >> 4;
    Elow();
  } while ( x & 0x80 );

  // write data
  DDRD |= 0xF0;
  if (rs) {
    RShigh();
  } else {
    RSlow();
  }
  RWlow();
  Ehigh() ;
  PORTD = d & 0xF0;     _delay_us(0.23);
  Elow();               _delay_us(0.5);
  Ehigh() ;
  PORTD = d << 4;       _delay_us(0.23);
  Elow();
}

void lcd8write(uint8_t d)
{
  Ehigh() ;
  PORTD = d & 0xF0;     _delay_us(0.23);
  Elow() ;              _delay_us(1520);
}

void initlcd(void)
{
  DDRD |= 0xF0;      // PD7..PD4 output
  DDRB |= 0x07;      // PB2..PB0 output
  RSlow();
  RWlow();

  _delay_ms(15);
  lcd8write(0x30);
  lcd8write(0x30);
  lcd8write(0x30);
  lcd8write(0x20);   // set 4-bit
  lcdcommand(0x28);  // 4-bit, 2 lines, 5x8
  lcdcommand(0x0C);  // display on, cursor off
  lcdcommand(0x06);  // write right
  lcdcommand(0x01);  // clear display
}

void lcdputs(char *s)
{
  char c;

  while ( (c = *s++) ) {
    lcdputc(c);
  }
}

int main(void)
{
  uint16_t f1 = 1;
  uint16_t f2 = 1;
  uint16_t h;
  char buffer[6];

  initlcd();

  while (1) {
    lcdcommand(0x01);       // Clear display
    lcdcommand(1<<7|0x0);   // Start at line 1
    utoa(f1, buffer, 10);
    lcdputs(buffer);
    lcdcommand(1<<7|0x40);  // Start at line 2
    utoa(f2, buffer, 10);
    lcdputs(buffer);
    _delay_ms(1000);
    if (f2 > (UINT_MAX/2)) {
      f1 = 1;
      f2 = 1;
    } else {
      h  = f2;
      f2 = f1 + f2;
      f1 = h;
    }
  }
}
```

Nadat er een seconde gewacht is, worden nieuwe getallen van Fibonacci uitgerekend en begint het proces van voren af aan. De getallen van Fibonacci worden 1 gemaakt als `f2` groter is dan de helft van het bereik (`UINT_MAX/2`) van de 16-bits *unsigned* integers.

## 19.5 Toepassing met de bibliotheek van Peter Fleury

De site van Peter Fleury is: http://jump.to/fleury/ Deze site bevat naast de LCD-bibliotheek ook een UART- en een $I^2C$-bibliotheek.

Het voorgaande voorbeeld toont aan dat het schrijven naar een karaktergeoriënteerd LCD behoorlijk complex is. Daarom is het verstandig gebruik te maken van bestaande bibliotheken.
Peter Fleury heeft een HD44780 compatibele LCD-bibliotheek gemaakt voor de AVR GNU-compiler. Deze bibliotheek bestaat uit twee bestanden `lcd.c` en `lcd.h`. Het c-bestand bevat een groot aantal functies. Tabel 19.6 geeft de functies die toegankelijk zijn voor de gebruiker. Het *header*-bestand `lcd.h` bevat een aantal definities die de gebruiker moet aanpassen voor zijn situatie. Deze definities staan in tabel 19.7 met de standaardwaarde en de waarde, die gebruikt is in het voorbeeld van code 19.4.

Overigens is de bibliotheek van Fleury geen echte bibliotheek, want dan zou het als een gecompileerd bestand, bijvoorbeeld als `liblcd.a`, aan `avr/lib` en `lcd.h` aan `avr/include` toegevoegd zijn. Het bestand `lcd.h` moet door de gebruiker worden aangepast, `lcd.c` gebruikt `lcd.h` en daarom moet `lcd.c` zelf gecompileerd worden.

De bestanden `lcd.c` en `lcd.h` kunnen het beste aan de werkdirectory worden toegevoegd. Aan `lcd.c` hoeft — of beter moet — niets worden gewijzigd. Het *header*-bestand moet wel worden aangepast. De bibliotheek heeft de kristalfrequentie `XTAL` nodig voor verschillende tijdvertragingen. De eerste groep definities legt de eigenschappen van het type display vast. De tweede groep definieert de aansluitingen tussen de microcontroller en het LCD.

**Tabel 19.6 : Overzicht functies LCD-bibliotheek Peter Fleury.**

| Functie | Omschrijving |
|---|---|
| `void lcd_init (uint8_t dispAttr)` | Initialisatie display en selectie type cursor |
| `void lcd_clrscr (void)` | Display leeg maken en cursor naar beginstand |
| `void lcd_home (void)` | Zet cursor bij beginstand |
| `void lcd_gotoxy (uint8_t x, uint8_t y)` | Zet cursor op specifieke positie |
| `void lcd_putc (char c)` | Schrijf karakter op huidige positie cursor |
| `void lcd_puts (const char *s)` | Schrijf string op huidige positie cursor |
| `void lcd_puts_p (const char *progmem_s)` | Toon string uit programmageheugen op huidige positie |
| `void lcd_command (uint8_t cmd)` | Stuur instructie naar het LCD |
| `void lcd_data (uint8_t data)` | Identiek aan `lcd_putc` zonder interpretatie `'\n'` |
| `lcd_puts_P(__s)` | Macro voor `lcd_puts_p(PSTR(__s))` |

De LCD-bibliotheek van Peter Fleury kent twee aansluitschema's: één voor de 4-bit *io* mode met vier datalijnen en één voor een 8-bit *memory mapped* mode. Deze bibliotheek kent de 8-bit *io* mode met acht datalijnen niet. De 8-bit *memory mapped* mode kan alleen toegepast worden bij microcontrollers, die een directe interface hebben naar een SRAM, zoals bijvoorbeeld de AT90S8515, de ATmega64, de ATmega128 en de ATmega103. Deze interface gebruikt de adres- en databusaansluitingen en de signalen `RD` en `WR` van de microcontroller. De ATmega32 heeft geen interface naar een SRAM en dus kan de 8-bits *memory mapped* mode niet worden toegepast.

Een schema voor de 8-bit *memory mapped* mode staat op de site van Peter Fleury: http://jump.to/fleury/

In code 19.4 staat een programma dat exact hetzelfde doet als code 19.3 maar nu met behulp van de bibliotheek van Peter Fleury. Tabel 19.7 geeft naast de

**Tabel 19.7 : Overzicht definities LCD-bibliotheek Peter Fleury.** De laatste kolom geeft de waarden die in het voorbeeld van deze paragraaf gebruikt zijn. Als er niets staat, is de definitie niet relevant en wordt de standaardwaarde gebruikt.

| Functie | Omschrijving | Standaardwaarde | Voorbeeld |
|---|---|---|---|
| XTAL | Kristalfrequentie | 4000000 | F_CPU |
| LCD_CONTROLLER_KS0073 | KS0073 (1) of HD44780U (0) | 0 | 0 |
| LCD_LINES | Aantal regels | 2 | 2 |
| LCD_DISP_LENGTH | Regellengte | 16 | 16 |
| LCD_LINE_LENGTH | Regellengte DDRAM | 0x40 | 0x40 |
| LCD_START_LINE1 | Start adres regel 1 | 0x00 | 0x00 |
| LCD_START_LINE2 | Start adres regel 2 | 0x40 | 0x40 |
| LCD_START_LINE3 | Start adres regel 3 | 0x14 | |
| LCD_START_LINE4 | Start adres regel 4 | 0x54 | |
| LCD_WRAP_LINES | Doorgaan op volgende regel | 0 | 0 |
| LCD_IO_MODE | 4-bit (1) of memory mapped (0) | 1 | 1 |
| LCD_PORT | Poort voor LCD | PORTA | |
| LCD_DATA0_PORT | Poort voor data bit 0 (D4) | LCD_PORT | PORTD |
| LCD_DATA1_PORT | Poort voor data bit 1 (D5) | LCD_PORT | PORTD |
| LCD_DATA2_PORT | Poort voor data bit 2 (D6) | LCD_PORT | PORTD |
| LCD_DATA3_PORT | Poort voor data bit 3 (D7) | LCD_PORT | PORTD |
| LCD_DATA0_PIN | Pin voor data bit 0 (D4) | 0 | 4 |
| LCD_DATA1_PIN | Pin voor data bit 0 (D5) | 1 | 5 |
| LCD_DATA2_PIN | Pin voor data bit 0 (D6) | 2 | 6 |
| LCD_DATA3_PIN | Pin voor data bit 0 (D7) | 3 | 7 |
| LCD_RS_PORT | Poort voor signaal RS | LCD_PORT | PORTB |
| LCD_RS_PIN | Pin voor signaal RS | 4 | 0 |
| LCD_RW_PORT | Poort voor signaal RW | LCD_PORT | PORTB |
| LCD_RW_PIN | Pin voor signaal RW | 5 | 1 |
| LCD_E_PORT | Poort voor signaal E | LCD_PORT | PORTB |
| LCD_E_PIN | Pin voor signaal E | 6 | 2 |

F_CPU wordt onder andere door delay.h gebruikt. AVRstudio definieert deze constante automatisch als bij de configuratie opties de frequentie wordt opgegeven. In de Makefile is F_CPU dan een voorgedefineerde macro.

standaardwaarden voor de definities ook de waarden zoals deze in dit voorbeeld in lcd.h zijn aangepast. De kristalfrequentie XTAL mag ook de voorgedefinieerde macro F_CPU zijn. De LCD_IO_MODE is de 4-bit io mode. Dan kan het LCD op elke pin van de microcontroller worden aangesloten. Het schema van dit voorbeeld staat in figuur 19.21. De controllijnen en de datalijnen zijn met twee verschillende poorten van de microcontrollers verbonden. Daarom wordt de macro LCD_PORT niet gebruikt. Voor de dataverbinding is per bit aangegeven dat deze aangesloten op poort D. De besturingssignalen zijn alle drie expliciet verbonden met poort B. Het hoofdprogramma van code 19.4 komt bijna helemaal overeen met het hoofdprogramma uit code 19.3. De functies lcd_init, lcd_clrscr, lcd_home, lcd_gotoxy en lcd_puts komen alleen nu uit de bibliotheek van Peter Fleury.

## 19.6 Geformatteerd afdrukken op een LCD

Met FDEV_SETUP_STREAM() kan printf zo worden gedefinieerd, dat het naar een LCD of TWI- of UART-verbinding schrijft. In paragraaf 20.10 wordt dit voor de UART besproken.

Een gewoon C-programma gebruikt printf met de *format specifiers* %e, %f en %g om drijvende komma getallen en gebroken getallen af te drukken. Omdat de microcontroller beperkte geheugen- en rekenfaciliteiten heeft, is het geformatteerd afdrukken en lezen niet of anders geïmplementeerd. Lezen en afdrukken naar bestand is niet mogelijk. Er is geen stdin en stdout, dus de functie printf is niet bruikbaar.

**Code 19.4** : **Voorbeeld met de LCD-bibliotheek van Peter Fleury.** Deze code gebruikt de 4-bit mode en het schema uit figuur 19.21 en is functioneel gelijk aan code 19.3. In `lcd.h` zijn de definities uit de laatste kolom van tabel 19.7 gebruikt.

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3  #include <stdlib.h>
4  #include <limits.h>
5  #include "lcd.h"
6
7  int main(void)
8  {
9    uint16_t f1 = 1;
10   uint16_t f2 = 1;
11   uint16_t h;
12   char buffer[16];
13
14   lcd_init(LCD_DISP_ON);
15
16   while (1) {
17     lcd_clrscr();
18     utoa(f1, buffer, 10);
19     lcd_puts(buffer);
20     lcd_gotoxy(0,1);
21     utoa(f2, buffer, 10);
22     lcd_puts(buffer);
23     _delay_ms(400);
24     if (f2 > (UINT_MAX/2)) {
25       f1 = 1;
26       f2 = 1;
27     } else {
28       h  = f2;
29       f2 = f1 + f2;
30       f1 = h;
31     }
32   }
33 }
```

Peter Fleury geeft aan dat de bestanden `lcd.c` en `lcd.h` verspreid en aangepast mogen worden, mits er voldaan wordt aan de *GNU General Public License*.
Op zich is dat een heel goed initiatief, maar het gevolg is dat er op het internet veel verschillende versies van deze bibliotheken te vinden zijn. Dit boek gebruikt `lcd.c` versie `1.14.2.1` van 29 januari 2006 en `lcd.h` versie `1.13.2.2` van 30 januari 2006.

De functie `sprintf` is wel beschikbaar. Deze functie schrijft de uitvoer geformatteerd naar een stringbuffer. Vervolgens kan de inhoud van de buffer naar het LCD worden geschreven. Het nadeel is dat deze functie veel resources gebruikt. In plaats daarvan heeft de *avr-libc* bibliotheek een aantal conversiefuncties voor verschillende dataformaten. Tabel 19.8 geeft een overzicht. De functies `dtostrf` en `dtostre` gebruiken de math-bibliotheek (*libm*).

In code 19.3 en 19.4 is de functie `utoa` gebruikt om de **`unsigned int`**'s `f1` en `f2` om te zetten naar een string. Deze conversie kan ook met de functie `sprintf` worden gedaan. Regel 18 moet dan vervangen worden door:

```
19    sprintf(buffer, "%u", f1);
```

en regel 21 door:

```
22    sprintf(buffer, "%u", f2);
```

Het nadeel van deze methode is dat het programma meer resources gebruikt. Met `sprintf` is het programmageheugen 2288 bytes groot in plaats van 906 bytes,

**Tabel 19.8 : De extra functies in de stdlib-bibliotheek bij de avr-libc.**

| C-functie | omschrijving |
|---|---|
| `char *itoa(int val, char *s, radix)` | zet (**signed**) **int** om naar alfanumerieke string |
| `char *ltoa(long val, char *s, radix)` | zet (**signed**) **long** om naar alfanumerieke string |
| `char *utoa(unsigned int val, char *s, radix)` | zet **unsigned int** om naar alfanumerieke string |
| `char *ultoa(unsigned int val, char *s, radix)` | zet **unsigned long** om naar alfanumerieke string |
| `long random(void)` | geeft een willekeurige waarde tussen `0` en `RANDOM_MAX` |
| `void srandom(unsigned long seed)` | geeft een nieuwe startwaarde aan `random()` |
| `long random_r(unsigned long *ctx)` | variant van `random()` |
| `char *dtostre(double val, unsigned char prec, unsigned char flags)` | zet **double** om naar string (`[-]d.ddde±dd`) |
| `char *dtostrf(double val, signed char width, double val, unsigned char prec, char *s)` | zet **double** om naar string (`[-]d.ddd`) |

bovendien zijn er ook vier databytes nodig. De conversie duurt ook langer. Er zijn voor de conversie bij `sprintf` ongeveer 600 klokslagen en bij `utoa` ruim 250 klokslagen.

## 19.7 Het weergeven van gebroken getallen op een LCD

De functies `dtostrf` en `dtostre` hebben de math-bibliotheek nodig. Het linken van de code moet met de optie `-lm` gedaan worden.

Door code 19.5 na regel 22 aan code 19.4 toe te voegen, drukt het programma ook de Gulden Snede ($\Phi = f_2/f_1$) af. Het quotiënt `f2/f1` is een gebroken getal en wordt met de functie `dtostrf` omgezet in een alfanumerieke string. De eerste parameter van `dtostrf` is het getal (`f2/f1`) dat omgezet wordt. De tweede en de derde parameter zijn het totaal aantal karakters van de alfanumeriek string en het aantal cijfers achter de komma (punt). De vierde parameter is het adres van de stringbuffer. De berekende waarde wordt op positie zeven van de eerste regel geschreven. Het resultaat staat in figuur 19.23.

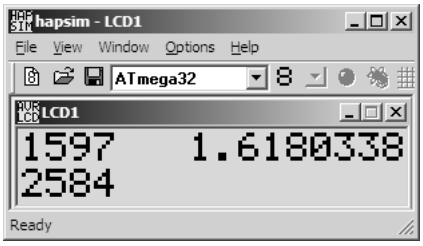
**Code 19.5 : Toevoeging voor afdrukken Gulden Snede met `dtostrf`.**

```
23      lcd_gotoxy(7,0);
24      dtostrf ((double) f2/f1, 9, 7, buffer);
25      lcd_puts(buffer);
```

**Code 19.6 : Toevoeging voor afdrukken Gulden Snede met `sprintf`.**

```
23      lcd_gotoxy(7,0);
24      sprintf(buffer,"%9.7f", (double) f2/f1);
25      lcd_puts(buffer);
```

Code 19.6 bevat een toevoeging met exact dezelfde functionaliteit als code 19.5, maar in dit geval wordt de conversie met `sprintf` gedaan.




**Figuur 19.23 : De LCD-simulatie met rechtsboven de Gulden Snede.**

De *avr-gcc*-compiler kent drie implementaties voor de `printf`-functies. De standaard implementatie bevat alle functionaliteiten, behalve de functies die te maken

hebben met gebroken getallen. De *format specifiers* `%e`, `%f` en `%g` zijn niet geïmplementeerd en leveren altijd een vraagteken (`?`) op. Er is een minimale versie met alleen basale integer- en stringconversies. De meeste opties zijn niet beschikbaar. Van de *flags* is bijvoorbeeld alleen `#` voorhanden. Voor het linken van de minimale versie zijn deze opties nodig:

```
-Wl,-u,vfprintf -lprintf_min
```

De uitgebreide versie met de volledige functionaliteit wordt meegelinkt bij deze compileropties:

```
-Wl,-u,vfprintf -lprintf_flt
```

Bij de compilatie moet `stdio.h` worden ingesloten, anders geeft de compiler een waarschuwing dat `sprintf` niet bekend is. De genoemde opties zijn alleen nodig bij het linken.

Code 19.7 bevat een alternatief voor code 19.5 en code 19.6. De conversie gebeurt hier met behulp van `utoa` en `ultoa`. Deze methode gebruikt een **`double`** voor het berekenen van de cijfers achter de komma. Dit is nodig om Φ met zeven cijfers achter de komma (punt) te kunnen berekenen. Zonder deze *typecasting* kan de Gulden Snede slechts met vijf decimalen worden berekend.

**Code 19.7 : Alternatieve methode voor afdrukken Gulden Snede.**

```
23      h=f2/f1;
24      utoa(h, buffer,10);
25      lcd_puts(buffer);
26      lcd_putc('.');
27      ultoa ( ((double) f2/f1 - h)*1000000, buffer, 10);
28      lcd_puts(buffer);
```

Tabel 19.9 vergelijkt de drie methoden met elkaar. De programma's zijn steeds gecompileerd met de optimalisatie-optie `-Os`. Voor elke methode staat in de tabel de grootte van het programmageheugen en het datageheugen, het aantal klokslagen dat voor de conversie nodig is en de uitvoer van het programma. Een Φ betekent dat de Gulden Snede correct wordt weergegeven en een vraagteken (`?`) betekent dat er een vraagteken op het display verschijnt. De drie programma's zijn op drie manieren gecompileerd: zonder extra opties, met `-lm` en met `-Wl,-u,vfprintf -lprintf_flt -lm`.

De waarden uit tabel 19.9 zijn gevonden met AVRstudio versie 4.15.589 en WinAVR versie 20080610. Bovendien verschilt de geteste codes in geringe mate met de code uit paragraaf 19.6 en paragraaf 19.7.

**Tabel 19.9 : Vergelijking van de drie methoden om gebroken getal af te drukken.**

| methode | aspect | extra optie | | |
|---|---|---|---|---|
| | | geen | `-lm` | `-Wl,-u,vfprintf -lprintf_flt -lm` |
| `dtostrf` | program (data) | 4268 (264) | 2980 (0) | 5266 (0) |
| | klokslagen | 3680 | 1927 | 1927 |
| | uitvoer | Φ | Φ | Φ |
| `sprintf` | program (data) | 4156 (270) | 2978 (6) | 4558 (6) |
| | klokslagen | 3680 | 1756 | 2903 |
| | uitvoer | ? | ? | Φ |
| alternatief | program (data) | 4864 (264) | 2182 (0) | 5150 (0) |
| | klokslagen | 3680 | 2449 | 2449 |
| | uitvoer | Φ | Φ | Φ |

Bij alle drie de methoden zorgt de optie `-lm` ervoor dat de grootte van het programma kleiner wordt en dat de conversie sneller gaat. De functie `sprintf` geeft een vraagteken zonder de opties `-Wl,-u,vfprintf -lprintf_flt`. De *reference manual* stelt terecht dat bij de functie `dtostrf` de optie `-lm` nodig is. Sterker, geef altijd de optie `-lm` mee als het type **`double`** gebruikt wordt.

Methode `dtostrf` met optie `-lm` is duidelijk kleiner en sneller dan de methode `sprintf` met optie `-Wl,-u,vfprintf -lprintf_flt -lm`. Daarom is het beter om `dtostrf` te gebruiken.
De alternatieve methode van code 19.7 is kleiner maar langzamer dan de methode met `dtostrf`. Het lijkt daarom weinig zinvol om eigen oplossingen te bedenken. Wel kan het zinvol zijn om methoden toe te passen zonder **`double`**'s. In figuur 19.24 staan twee codefragmenten: één met een **`double`** en één met een **`unsigned int`**.

```
1    double d = 12.34;
2    dtostrf (d, 5, 2, buffer);
3    lcd_puts(buffer);
```

```
1    unsigned int u = 1234;
2    utoa(u/100, buffer, 10);
3    lcd_puts(buffer);
4    lcd_putc('.');
5    utoa(u%100, buffer, 10);
6    lcd_puts(buffer);
```

**Figuur 19.24 : Twee oplossingen voor afdrukken van `12.34` op LCD.** Links staat een fragment dat het getal als een **`double`** definieert en rechts een fragment met het getal als **`unsigned int`**. In beide situaties komt er `12.34` op het LCD te staan.

De oplossing met de **`double`** is 1372 bytes groter en heeft voor de conversie 1254 klokslagen nodig in plaats van 669. Veel programmeurs vermijden daarom bij het programmeren van eenvoudige 8-bits en 16-bits microcontrollers het gebruik van gebroken getallen.

# 20

# UART

**Doelstelling**

Dit hoofdstuk behandelt de USART van de ATmega32. Uitgelegd wordt wat een UART en een USART is en waarvoor je deze kunt gebruiken. Je leert hoe je het juiste protocol en de juiste *baud rate* instelt. Je leert hoe je gegevens verstuurt en ontvangt met en zonder gebruikmaking van een circulaire buffer.

**Onderwerpen**

De behandelde onderwerpen zijn:

- Het verschil tussen synchrone en asynchrone communicatie.
- De opbouw van UART en het instellen van de *baud rate*.
- De instelling van het RS232-protocol.
- De communicatie met het RS232-protocol tussen een ATmega32 en een pc.
- Het ontvangen en verzenden van data.
- Circulaire buffers voor de buffering van gegevens.
- De UART-bibliotheek van Peter Fleury.
- Geformatteerd versturen en ontvangen van data via de UART met `printf` en `scanf`.

De communicatie met de UART wordt gedemonstreerd met deze voorbeelden:

- Het versturen van karakters zonder interrupt.
- Het ontvangen en versturen van gegevens met een interrupt.
- Het ontvangen en versturen van tekst met een circulaire buffer.
- Het ontvangen en versturen van tekst met de UART-bibliotheek van Peter Fleury.
- Het ontvangen en versturen van getallen.
- Het maken van een stream met `FDEV_SETUP_STREAM` voor `printf`.
- Het ontvangen en versturen met `scanf` en `printf`.

De ATmega32 heeft drie mogelijkheden voor seriële communicatie, namelijk: een USART, een SPI en een TWI. De SPI (*Serial Peripheral Interface*) en de TWI *Two-Wire serial Interface* zijn bedoeld voor communicatie tussen geïntegreerde schakelingen op een PCB. Deze interfaces worden bijvoorbeeld gebruikt voor de verbinding met een DAC, een serieel LCD of een andere microcontroller. De USART (*Universal Synchronous/Asynchronous Receiver Transmitter*) is bedoeld voor de communicatie tussen apparaten, dus tussen PCB's onderling. De microcontroller kan via de USART met de RS232-poort van een pc communiceren.

USB (*Universal Serial Bus*), Firewire en Ethernet zijn voorbeelden van snelle seriële verbindingen die bij een pc worden gebruikt.

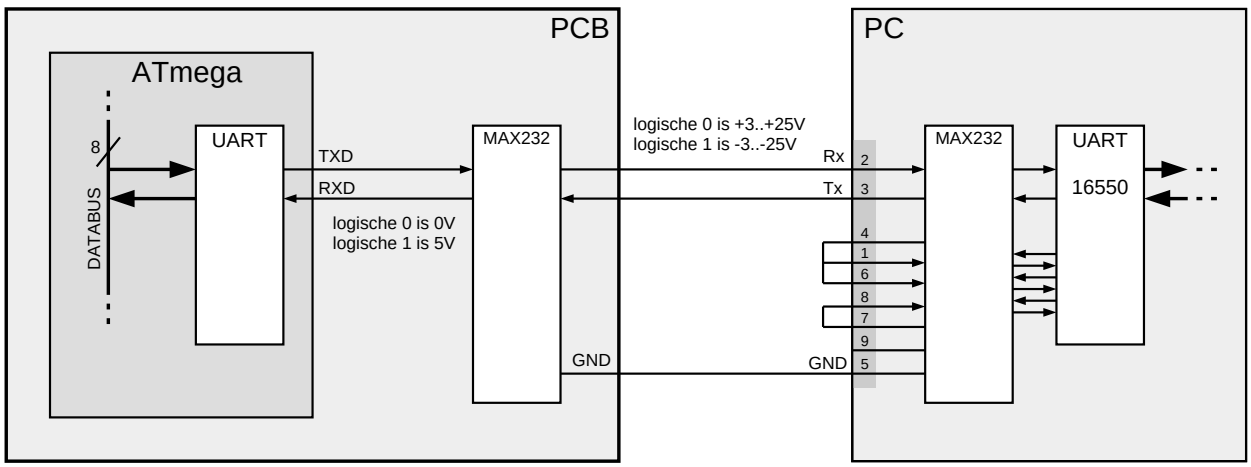
Het feit dat de ATmega32 drie manieren kent om serieel te communiceren, geeft aan dat deze methode van communiceren belangrijk is. Bij parallelle communicatie worden een aantal bits gelijktijdig verzonden. Bij seriële communicatie

worden de bits na elkaar verstuurd. Hoewel dat laatste onhandig lijkt, is serieel efficiënter dan parallel. Ten eerste is maar een enkele verbinding nodig in plaats van bijvoorbeeld acht parallelle lijnen. Ten tweede is het parallel versturen van gegevens bij grote snelheden lastig omdat er looptijdverschillen tussen de datalijnen ontstaan.

Om goed te kunnen communiceren moeten de zender (*transmitter*) en de ontvanger (*receiver*) hetzelfde protocol gebruiken en moet de ontvanger weten wanneer een bericht begint. Dit kan synchroon (*synchronous*) en asynchroon (*asynchronous*) gedaan worden. Bij synchrone communicatie is er een gemeenschappelijk synchronisatiesignaal (klok). Bij asynchrone communicatie is er geen synchronisatiesignaal en wordt er gesynchroniseerd op het datasignaal.

Dit hoofdstuk gebruikt meestal de term UART. Alleen als er ook synchrone aspecten aan de orde zijn, wordt er over een USART gesproken.

Hoewel de USART van de ATmega32 ook synchroon gebruikt kan worden, wordt in de meeste applicaties de USART van de ATmega32 asynchroon toegepast. Het kloksignaal XCK voor de synchronisatie wordt dan niet gebruikt. Pin 0 van poort B is dan beschikbaar voor andere doeleinden. Feitelijk is de USART dan een UART (*Universal Asynchronous Transmitter Receiver*) met een uitgang (TXD) en een ingang (RXD).




**Figuur 20.1 :** **De verbinding tussen een pc en een ATmega32.** De TXD en de RXD van UART van de ATmega32 zijn verbonden via een nulmodemverbinding met de Rx en de Tx van de pc. De handshake-signalen van de pc zijn direct teruggekoppeld naar de pc. Tussen de ATmega32 en de pc zit een MAX232 die de signalen naar de juiste signaalniveaus converteert.

De MAX232 is de meest populaire *RS232 level converter*. De fabrikant Maxim levert een groot aantal varianten, onder andere de MAX233 die geen externe condensatoren nodig heeft. Vele andere fabrikanten, zoals TI en Intersil, leveren ook *RS232 level converters*.

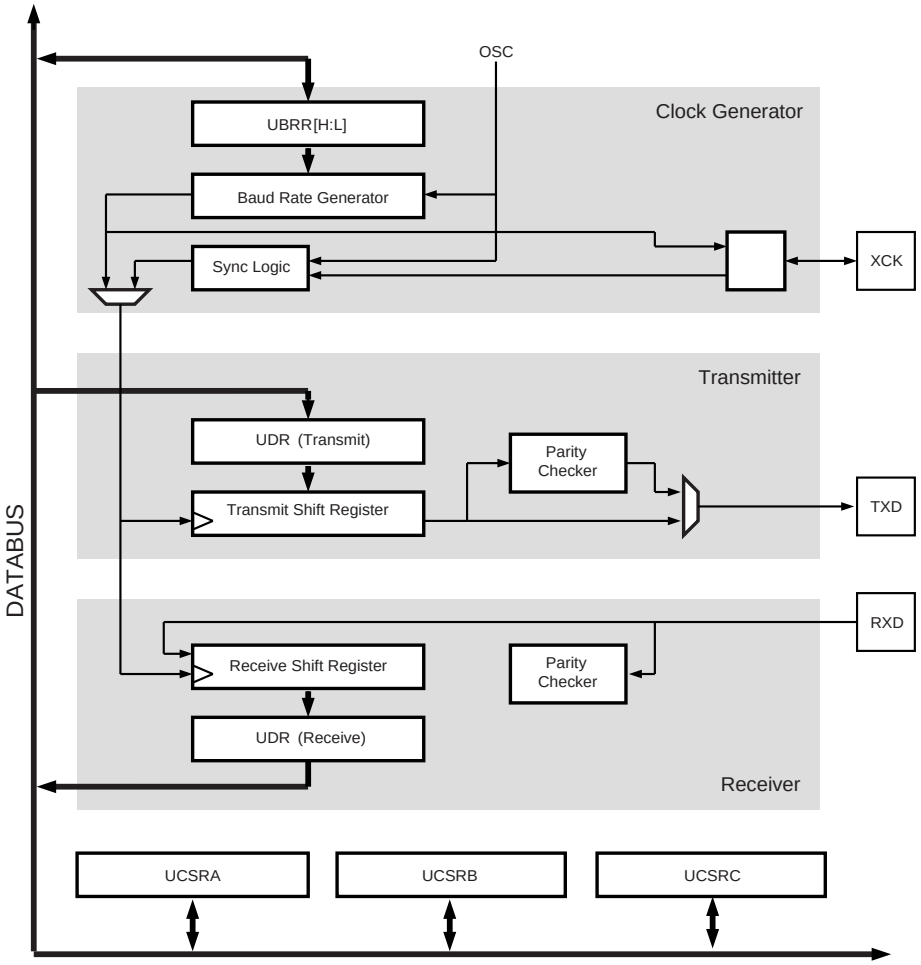
Naast de Rx en Tx kent de RS232-poort ook een aantal signalen voor *handshaking*. Deze aansluitingen ontbreken bij de UART van de ATmega32. In figuur 20.1 zijn de handshake-signalen teruggekoppeld naar de pc.

In figuur 20.1 is de UART van de ATmega32 verbonden met de seriële RS232-poort van een pc. De TXD en de RXD zijn via een MAX232 met de Rx en de Tx van de RS232-poort te verbonden. De signalen mogen niet direct op elkaar worden aangesloten. De signaalniveaus van de RS232-poort van de pc voldoen aan de RS232-norm. Dit betekent dat de logische 0 tussen de +3 en +25 V en de logische 1 tussen de −3 en −25 V ligt. De MAX232 converteert de signalen naar het juiste signaalniveau.

Figuur 20.1 maakt zichtbaar dat de pc intern ook een MAX232 en een UART heeft. In bijlage A staat meer informatie over RS232 en er staan een aantal voorbeelden — in de taal C — van pc-applicaties die informatie van en naar de RS232-poort sturen.

Het UBRR-register is 16-bits breed en bestaat uit een hoog (UBRRH) en laag byte (UBRRL). Het adres van UBRRH is identiek aan het adres van UCSRC. De ATmega32 gebruikt UBRRH als bit 7 laag is en UCSRC als bit 7 hoog is.

Het UDR (transmit)-register en het UDR (receive)-register hebben beide hetzelfde adres. Als er naar UDR wordt geschreven, wordt automatisch UDR (transmit) gebruikt en als er uit UDR wordt gelezen, gebeurt dit ook automatisch uit UDR (receive).




**Figuur 20.2 : De opbouw van de USART bij de ATmega32.** Het bovenste deel genereert het kloksignaal voor de schuifregisters. Dit signaal wordt bepaald uit UBRR en de systeemklok of — in het geval dat de USART synchroon in slave mode wordt gebruikt — het externe signaal XCK.
In het midden staat de zender. De te verzenden waarde wordt in UDR (transmit) gezet en door de transmitter via het schuifregister naar buiten geschoven.
Daaronder staat de ontvanger, die serieel de bits naar binnen schuift en daarna het gelezen byte in UDR (receive) plaatst.
Helemaal onderaan staan de drie control- en statusregisters.

## 20.1 Opbouw USART en instellen baud rate

Een UART bestaat uit een serieel-parallel-omzetter en een parallel-serieel-omzetter. De parallel-serieel-omzetter haalt bytes van de databus en stuurt deze serieel weg en de serieel-parallel-omzetter maakt van de serieel ingelezen bits weer bytes en zet deze bytes op de databus.
Figuur 20.2 laat zien dat de USART van de ATmega32 uit drie delen bestaat: een blok dat een kloksignaal met de juiste *baud rate* genereert, de zender (*transmitter*) en de ontvanger (*receiver*). De ontvanger en de zender hebben beide een schuifregister om de bits met het juiste tempo naar binnen of naar buiten te schuiven. Het tempo wordt bepaald door de waarde in het UBRR-register (*USART Baud Ra-*

*te Register*) en door de frequentie $f_{cpu}$ van de systeemklok. De frequentie $f_{baud}$ van het signaal, waarmee de schuifregisters — in de normale asynchrone mode — geklokt worden, is:

$$f_{baud} = \frac{f_{cpu}}{16\,(\texttt{UBRR}+1)} \tag{20.1}$$

Behalve de normale asynchrone mode heeft de USART ook een *double speed* asynchrone mode en een synchrone mode. Formule 20.1 en 20.2 zijn voor deze modes gelijk, maar met respectievelijk een factor 8 en 2 in plaats van 16.
De *double speed* mode wordt geselecteerd door het U2X-bit uit het UCSRA-register hoog te maken.
De synchrone mode wordt ingesteld met het UMSEL-bit uit het UCSRC-register.

Meestal is er een gewenste *baud rate* gegeven en moet daaruit de waarde, die in het UBRR-register moet komen te staan, worden berekend:

$$\texttt{UBRR} = \frac{f_{cpu}}{16\,f_{baud}} - 1 \tag{20.2}$$

Veel combinaties van $f_{cpu}$ en $f_{baud}$ leveren een gebroken getal. Het afronden kan tot gevolg hebben dat de bits niet met het juiste tempo geschoven worden en de communicatie niet goed functioneert. De fout $\Delta_{ubrr}$ in procenten is:

$$\Delta_{UBRR} = (\frac{f_{baud,benadering}}{f_{baud}} - 1) \times 100\% \tag{20.3}$$

Deze fout mag volgens de datasheet niet groter zijn dan 0,5%. De tabellen 68 tot en met 71 uit de datasheet van de ATmega32 geven voor een groot aantal gebruikelijke frequenties de waarde van UBRR met de bijbehorende fout $\Delta_{UBRR}$ voor alle gangbare *baud rates*.
Belangrijk is dat men zich realiseert dat niet elke *baud rate* geschikt is bij elke systeemklok. Standaard is de Window's HyperTerminal ingesteld op 9600 baud. Deze snelheid is niet geschikt bij de standaard 1 MHz klok van de ATmega32. De beste benadering voor UBRR is 6. Dit geeft een snelheid 8929 baud in plaats van 9600 baud. Deze afwijking van 7% is veel te groot en de communicatie zal niet lukken.

De datasheet is met een fout van 0,5% nogal streng. Theoretisch is de maximale fout een halve bit. Bij tien bits (start-, stop- en 8 databits) is dat 5%. Anderen noemen vaak 1% of 2% als maximale tolerantie voor beide zijde van de communicatie.

Het is verstandig een oscillator te kiezen met een 'mooie' of magische frequentie. Dat zijn frequenties die goed passen bij de *baud rate*-reeks. In tabel 20.1 staat een lijst met magische oscillatorfrequenties.
De voorbeelden in dit hoofdstuk gebruiken een klokfrequentie van 8 MHz en een communicatiesnelheid van 38400 baud. De berekende waarde voor UBRR is dan 12,02 en afgerond is dat 12. De afrondingsfout is kleiner dan 0,2%.
Het is beter om de berekening in de code op te nemen. De macro BAUD bevat de *baud rate*. De macro UBRR_VALUE bepaalt met formule 20.2 de waarde van UBRR uit BAUD en de klokfrequentie F_CPU.

**Tabel 20.1 : Magische oscillatorfrequenties.**

| $f_{cpu}$ in MHz |
|---|
| 1,8432 |
| 3,6864 |
| 7,3728 |
| 11,0592 |
| 14,7456 |

```
#define BAUD          38400
#define UBRR_VALUE    ( (F_CPU/(16UL*BAUD)) - 1 )
```

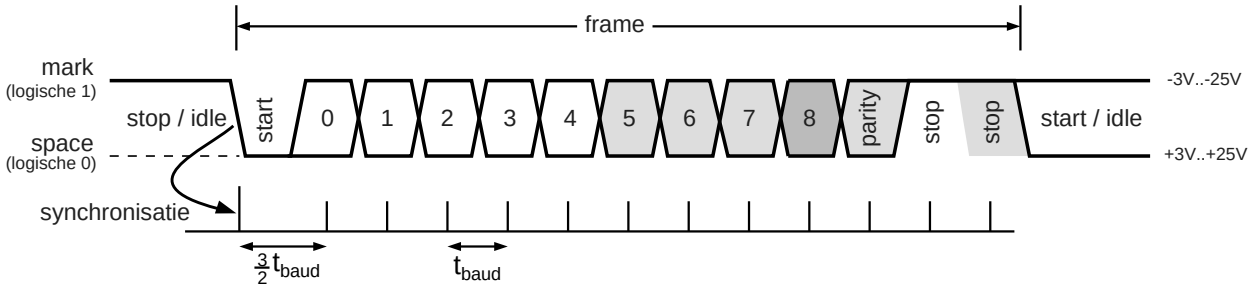
Bovenstaande berekening rondt altijd naar beneden af. De versie hieronder rondt — als de rest groter is dan een half — de waarde naar boven af en anders naar beneden af.

```
#define BAUD          38400
#define UBRR_VALUE    ( ((F_CPU) + 8UL*(BAUD)) / (16UL*(BAUD)) - 1UL )
```

## 20.2 Instelling protocol

Het protocol voor de asynchrone communicatie bestaat uit een startbit, één of meer stopbits en een aantal databits. Figuur 20.3 toont een frame van het protocol. De bits zijn met frequentie $f_{baud}$ verstuurd. De tijd tussen twee bits is de

baudperiode $t_{\text{baud}} = 1/f_{\text{baud}}$. Bij het lezen start de synchronisatie op de neergaande flank van het startbit. De kans op fouten is het kleinst als de ontvanger de bits halverwege detecteert. Na $\frac{3}{2}t_{\text{baud}}$ komt het midden van het eerste bit.




**Figuur 20.3 : De synchronisatie van het frame bij asynchrone communicatie vindt plaats op de neergaande flank van het startbit.** De grijs gekleurde bits zijn optioneel. De ontvanger detecteert de flank van het startbit. Na anderhalve baudperiode is het midden van het eerste bit. Tussen de bits zit steeds een baudperiode.

Het frame bevat vijf tot negen databits. Het negende bit is niet gebruikelijk en wordt verder niet besproken. Het pariteitsbit (*parity bit*) is optioneel en kan even of oneven zijn. De UART van de ATmega32 kent één of twee stopbits. De meeste toepassingen gebruiken een protocol met acht databits, geen pariteitsbit en een stopbit: het zogenoemde 8N1-protocol.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| URSEL | UMSEL | UPM1 | UPM0 | USBS | UCSZ1 | UCSZ0 | UPOL |

**Figuur 20.4 : Het `UCSRC`-register met de bits voor het instellen van het protocol.** Bit `URSEL` moet altijd 1 zijn. De `UPM1` en `UPM0` stellen het pariteitsbit in en bit `USBS` stelt het stopbit in. Samen met het `UCSZ2`-bit uit `UCSRB` regelen `UCSZ1` en `UCSZ0` het aantal databits.

Bit `URSEL` (bit 7) moet hoog zijn om de andere bits van `UCSRC` te kunnen schrijven. De registers `UCSRC` en `UBRR` hebben hetzelfde adres. Als `URSEL` (bit 7) laag is komen de gegevens in `UBRR` terecht.

Figuur 20.4 toont het `UCSRC`-register (*USART Control Status Register C*). Vijf bits uit dit register en het `UCSZ2`-bit uit `UCSRB` bepalen de eigenschappen van het protocol. Tabel 20.2, 20.3 en 20.4 geven de mogelijke waarden van deze bits voor de soort pariteit, het aantal stopbits en het aantal databits. Als bit `UMSEL` hoog is wordt de synchrone mode geselecteerd en als het laag is de asynchrone mode.

**Tabel 20.2 : De instelling van het pariteitsbit.**

| pariteit | UPM1..UPM0 |
|---|---|
| geen | 00 |
| even | 10 |
| oneven | 11 |

**Tabel 20.3 : De instelling van het aantal stopbits.**

| aantal | USBS |
|---|---|
| 1 | 0 |
| 2 | 1 |

**Tabel 20.4 : De instelling van het aantal databits.**

| aantal | UCSZ2 | UCSZ1..UCSZ0 |
|---|---|---|
| 5 | 0 | 00 |
| 6 | 0 | 01 |
| 7 | 0 | 10 |
| 8 | 0 | 11 |
| 9 | 1 | 11 |

## 20.3 Ontvangen en verzenden van data

Figuur 20.2 laat zien dat de zender (*transmitter*) en de ontvanger (*receiver*) allebei een dataregister en een schuifregister hebben. De dataregisters zijn twee fysiek verschillende registers. Het adres van deze registers is hetzelfde en daarom is de naam gelijk. Beide worden aangeduid met `UDR` (*USART Data I/O Register*).

Te verzenden gegevens worden in de UDR van de verzender geplaatst. De zender voegt de start-, stop-, en eventueel het pariteitsbit toe en schuift deze gegevens met de ingestelde *baud rate* naar buiten.
De ontvanger schuift de bits naar binnen in het schuifregister van de ontvanger. Als alle informatie — inclusief een eventueel pariteitsbit — ontvangen is, plaats de ontvanger de gegevens uit het schuifregister in het dataregister van de ontvanger. Het ontvangen en verzenden van gegevens is niets anders dan het lezen en schrijven van gegevens naar het UDR-register. Het UDR (receive)-register kan alleen gelezen worden als alle informatie ontvangen is. Er kan alleen naar het UDR (transmit)-register geschreven worden als de eerder verzonden gegevens verwerkt zijn.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **RXC** | **TXC** | **UDRE** | FE | DOR | PE | U2X | MPCM |

**Figuur 20.5 : Het UCSRA-register met de statusbits voor de UART.** Bit RXC geeft aan dat er gegevens ontvangen zijn. Bit TXC geeft aan dat de gegevens verstuurd zijn. Bit UDRE is hoog als het UDR-register leeg is.

Het UCSRA-register (*USART Control Status Register A*) — zie figuur 20.5 — heeft drie bits die de status van de UART aangeven: RXC, TXC en UDRE.
Het RXC-bit (*Receive Complete*) is hoog als er gegevens zijn ontvangen. Onderstaande code blijft de **while**-lus doorlopen totdat RXC hoog is. Daarna wordt de inhoud van UDR aan data toegekend.

```
while (!(UCSRA & (1<<RXC))) {};    // wait until data is received
data = UDR;                        // read data
```

Het TXC-bit (*Transmit Complete*) wordt hoog als de gegevens verstuurd zijn. In onderstaande code krijgt UDR de waarde data en daarna wordt er gewacht totdat TXC hoog is en de gegevens verstuurd zijn:

```
UDR = data;                        // write data
while (!(UCSRA & (1<<TXC))) {};    // wait until data is send
```

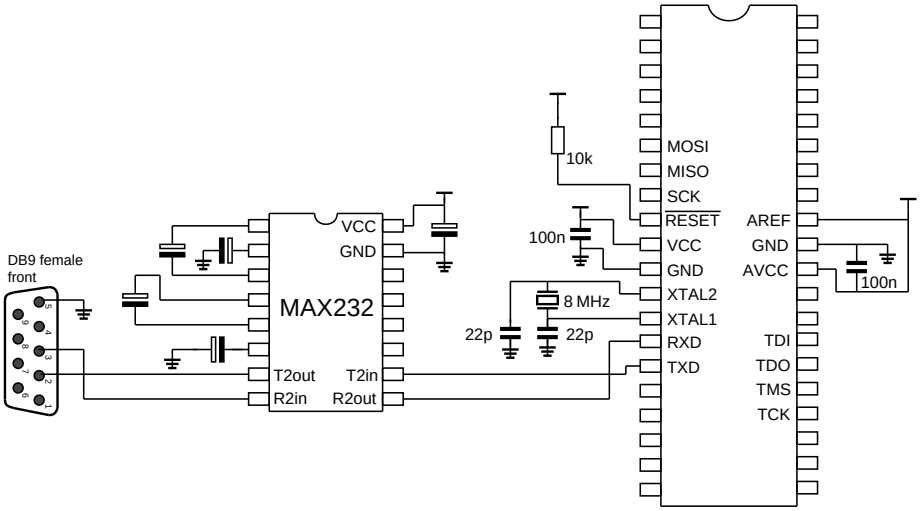
Deze oplossing is niet optimaal. Bij elk verzonden byte wordt er gewacht totdat alles verzonden is. Deze wachttijd kan aan meer zinvolle zaken besteed worden. Een betere oplossing is om het UDRE-bit te gebruiken. (*USART Data Register Empty*) uit UCSRA.

```
while (!(UCSRA&(1<<UDRE))){};      // wait until UDR is empty
UDR = data;                        // write data
```

Nu wordt eerst gewacht totdat UDR leeg is. Nadat data aan UDR is toegekend, gaat het programma gewoon verder. Onderwijl verstuurt de zender de databyte die in UDR is geplaatst.

## 20.4 Het versturen van karakters via de UART

In figuur 20.6 staat het schema voor het verzenden en het ontvangen van informatie via de UART. De MAX232 zorgt voor de inversie en de *level shifting*. De signalen links van de MAX232 voldoen aan de RS232-norm en rechts liggen de waarden tussen GND en VCC. De MAX232 heeft elektrolytische condensatoren van 1 $\mu$F nodig. Er bestaan veel varianten van de MAX232. Zo is de MAX233 van Maxim een IC met dezelfde functionaliteit, alleen heeft deze component geen condensatoren nodig.
In figuur 20.6 zijn pin 2 en pin 3 van de DB9-connector via de MAX232 verbonden met respectievelijk de TXD en de RXD van de ATmega32. Dit voorbeeld




**Figuur 20.6 :** **Het schema voor een seriële verbinding met de UART.**
Pin 2 en pin 3 van DB9-connector zijn verbonden via de MAX232 met respectievelijk de `TXD` en de `RXD` van de ATmega32.

verstuurt alleen data; de `RXD` wordt dus niet gebruikt. Het versturen van gegevens is eenvoudiger dan het ontvangen. Bovendien is het testen makkelijker. De ATmega32 heeft een extern kristal van 8 MHz.

Het protocol van 8 databits, geen (*None*) pariteitsbit en 1 stopbit wordt ook aangeduid als 8N1.

De functie `uart_init` uit code 20.1 initialiseert de UART. De waarde die in het `UBRR`-register moet komen wordt als parameter aan `uart_init` meegegeven. De UART wordt ingesteld op acht databits, een stopbit en geen pariteitsbit. De zender wordt geactiveerd door `TXEN`-bit uit het `UCSRB`-register hoog te maken. Het `UCSRB`-register (*USART Control Status Register B*) staat in figuur 20.7 en bevat een aantal besturingsbits voor de UART.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **RXCIE** | **TXCIE** | **UDRIE** | **RXEN** | **TXEN** | UCSZ2 | RXB8 | TXB8 |

**Figuur 20.7 :** **Het `UCSRB`-register met de besturingsbits voor de UART.** De bits `RXEN` en `TXEN` zetten de zender en de ontvanger aan. De bits `RXCIE`, `TXCIE` en `UDRIE` zetten de interrupts aan voor respectievelijk het ontvangen, het verzenden en het leeg zijn van het `UDR`-register.

Een (hyper-)terminal is een communicatieprogramma dat verbinding maakt met andere computers via modems, RS232 of Ethernet.

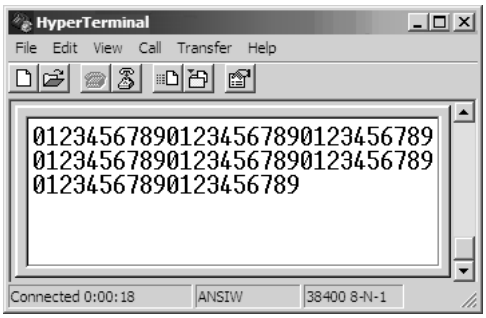
Tot Windows Vista zat HyperTerminal standaard bij Windows. HyperTerminal is gratis te downloaden op http://www.hilgraeve.com.

Naast HyperTerminal bestaan er vele andere terminals zoals Putty en SuperTerm (http://www.pteq.com).

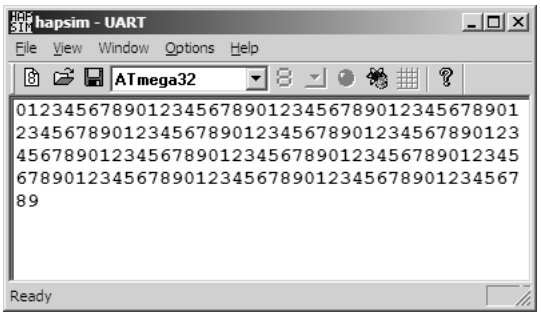
De waarde van het `UBRR`-register wordt berekend uit de klokfrequentie `F_CPU` en de *baud rate* `BAUD`. In dit voorbeeld — met 8 MHz en 38400 baud — is dat 12.

Het hoofdprogramma verstuurt elke 500 ms de ASCII-waarde van een cijfer uit de reeks `0` tot en met `9` naar buiten. De **`for`**-lus wacht totdat `UDR` leeg is en kent daarna aan `UDR` de ASCII-waarde van de lusvariabele toe. Deze waarde is gelijk aan de lusvariabele verhoogd met de ASCII-waarde van `'0'`.

In figuur 20.8 staat de schermafdruk van een hyperterminal, die aangesloten is op de schakeling van figuur 20.6 en waarbij de microcontroller geprogrammeerd is met het programma van code 20.1. Voor een goede communicatie moet de terminal ingesteld zijn op 38400 baud en op het 8N1-protocol. De simulator Hapsim heeft ook een terminal. Alleen hoeft er geen *baud rate* ingesteld te worden en is de terminal alleen geschikt voor het 8N1-protocol. Figuur 20.9 toont een schermafdruk van de Hapsim-terminal met de simulatie van code 20.1.




Figuur 20.8 : De uitvoer van het programma van code 20.1 in een hyperterminal.




Figuur 20.9 : De uitvoer van de simulatie van code 20.1 in de terminal van Hapsim.

## 20.5 Het ontvangen, converteren en versturen van karakters

Code 20.2 ontvangt een karakter via de RXD-ingang van de microcontroller. Het converteert dit karakter van hoofdletter naar kleine letter of omgekeerd en stuurt het geconverteerde karakter via de TXD-uitgang terug. De RXD van figuur 20.6 wordt nu wel gebruikt.
De functie `uart_init` is identiek aan die uit code 20.1. Alleen regel 12 is aangepast; de zender en de ontvanger staan nu allebei aan. De functie `change_case` converteert een hoofdletter naar onderkast en een kleine letter naar bovenkast. De andere karakters blijven ongewijzigd.

Code 20.1 : Het versturen van gegevens met de UART.

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  #define BAUD         38400   // F_CPU is 8000000 Hz
5  #define UBRR_VALUE   ( ((F_CPU) + 8UL*(BAUD)) / (16UL*(BAUD)) - 1UL )
6
7  void uart_init(unsigned int ubrr)
8  {
9    UBRRL = ubrr;
10   UBRRH = (ubrr >> 8);
11   UCSRC = _BV(URSEL)|_BV(UCSZ1)|_BV(UCSZ0);
12   UCSRB = _BV(TXEN);
13 }
14
15 int main(void)
16 {
17   int i;
18
19   init_uart(UBRR_VALUE);
20
21   while (1) {
22     for (i=0; i<=9; i++) {
23       while ( !(UCSRA & _BV(UDRE)) ) {};
24       UDR = i + '0';
25     }
26     _delay_ms(500);
27   }
28 }
```

In de oneindige lus van regel 29 wacht het hoofdprogramma op gegevens. Als er een karakter ontvangen is, wordt deze in `data` opgeslagen. Daarna wacht het programma op regel 31 totdat de `UDR` van de zender leeg is en stuurt het karakter geconverteerd terug.

**Code 20.2 : Het ontvangen en versturen van gegevens met de UART.**

```
1  #include <avr/io.h>
2  #include <ctype.h>
3
4  #define BAUD          38400   // F_CPU is 8000000 Hz
5  #define UBRR_VALUE    ( ((F_CPU) + 8UL*(BAUD)) / (16UL*(BAUD)) - 1UL )
6
7  void uart_init(unsigned int ubrr)
8  {
9    UBRRL = ubrr;
10   UBRRH = (ubrr >> 8);
11   UCSRC = _BV(URSEL)|_BV(UCSZ1)|_BV(UCSZ0);
12   UCSRB = _BV(RXEN)|_BV(TXEN);
13 }
14
15 char change_case(char c)
16 {
17   if ( isupper(c) )  return tolower(c);
18   if ( islower(c) )  return toupper(c);
19   return c;
20 }
21
22 int main(void)
23 {
24   char data;
25
26   uart_init(UBRR_VALUE);
27
28   while(1) {
29     while ( !(UCSRA & _BV(RXC)) ) {};
30     data = UDR;
31     while ( !(UCSRA & _BV(UDRE)) ) {};
32     UDR = change_case(data);
33   }
34 }
```

**Tabel 20.5 : Tijd nodig voor versturen van een byte.**

| $f_{\text{baud}}$ | $t_{\text{byte}}$ |
|---|---|
| 2400 | 4,2 ms |
| 4800 | 2,1 ms |
| 9600 | 1,1 ms |
| 14400 | 694 $\mu$s |
| 19200 | 521 $\mu$s |
| 28800 | 347 $\mu$s |
| 38400 | 260 $\mu$s |
| 57600 | 174 $\mu$s |
| 76800 | 130 $\mu$s |
| 115200 | 87 $\mu$s |

## 20.6 Toepassing met gebruik van een interrupt

Het kost tijd om een karakter te versturen of te ontvangen met een UART. Voor het 8N1-protocol moeten er tien bits verstuurd worden. De tijd $t_{\text{byte}}$ die nodig is om een byte te versturen of te ontvangen is dan:

$$t_{\text{byte}} = 10 * \frac{1}{f_{\text{baud}}}$$

In tabel 20.5 is deze tijd uitgerekend voor de diverse snelheden. Vooral bij lage snelheden kost het versturen of ontvangen van een karakter relatief veel tijd. In praktische situaties is het beter om met interrupts te werken. Als het `RXCIE`-bit

(*RX Complete Interrupt Enable*) uit UCSRB hoog is, is er steeds een interrupt als er een databyte ontvangen is door de UART.

**Code 20.3 : Het ontvangen en versturen van gegevens met een interrupt.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <util/delay.h>
4
5  #define BAUD         38400   // F_CPU is 8000000 Hz
6  #define UBRR_VALUE   ( ((F_CPU) + 8UL*(BAUD)) / (16UL*(BAUD)) - 1UL )
7
8  char change_case(char c);
9
10 void uart_init(unsigned int ubrr)
11 {
12   UBRRL = ubrr;
13   UBRRH = (ubrr >> 8);
14   UCSRC = _BV(URSEL)|_BV(UCSZ1)|_BV(UCSZ0);
15   UCSRB = _BV(RXEN)|_BV(TXEN)|_BV(RXCIE);
16 }
17
18 ISR(USART_RXC_vect)
19 {
20   char data;
21
22   data = UDR;
23   UDR = change_case(data);
24 }
25
26 int main(void)
27 {
28   DDRA = 0x01;            // Pin 0 port A output
29   uart_init(UBRR_VALUE);
30   sei();
31
32   while(1) {
33     PORTA ^= 0x01;        // Toggle Pin 0 port A
34     _delay_ms(500);
35   }
36 }
```

Code 20.3 gebruikt deze interrupt, maar is functioneel identiek aan code 20.2. Alleen wordt nu simultaan met het lezen en schrijven van de karakters ook pin 0 van poort A voortdurend hoog en laag gemaakt. De gebruikte schakeling is identiek aan figuur 20.6. Aan pin 0 van poort A kan een led worden aangesloten. Deze led zal dan met een frequentie van 1 Hz knipperen.

De functie change_case staat in een apart bestand. Op regel 8 staat het prototype van de functie. Het RXCIE-bit van het UCSRB-register is op regel 15 hoog gemaakt. Er wordt een interrupt gegeven als er gegevens ontvangen zijn. De interruptroutine die dan uitgevoerd wordt, staat op regel 18. Deze routine zet de waarde van UDR in data en zet de geconverteerde waarde weer in UDR om te verzenden. Er wordt niet getest op het leeg zijn van de UDR van de zender. Dat is niet nodig omdat er steeds maar één karakter gelezen wordt en het lezen en het versturen van beide

even lang duurt. Als er een karakter ontvangen wordt, is het daarvoor verzonden karakter ook helemaal verstuurd.
Het hoofdprogramma heeft een oneindige lus waarin pin 0 van poort A om de 500 ms afwisselend hoog en laag gemaakt wordt. Naast het activeren van het interruptbit RXCIE is op regel 30 ook het globale interruptmechanisme aangezet.
De Interrupt Service Routine leest voortdurend de gegevens uit het UDR (receive)-register en schrijft deze — geconverteerd – naar het UDR (transmit)-register. Meestal wordt er net als in code 20.3 een hulpvariabele gebruikt, hoewel dit ook korter kan:

```
ISR(USART_RXC_vect)
{
  UDR = change_case(UDR);
}
```

Bovenstaande werkt ook prima. De gegenereerde hexcode is exact hetzelfde. Waarschijnlijk gebruikt men dit niet graag, omdat het er vreemd uit ziet. De interruptroutine voor het terugkaatsen van gegevens zonder de conversie is:

```
ISR(USART_RXC_vect)
{
  UDR = UDR;
}
```

De linker UDR is de UDR (transmit) van de zender en de rechter UDR is de UDR (receive) van de ontvanger.

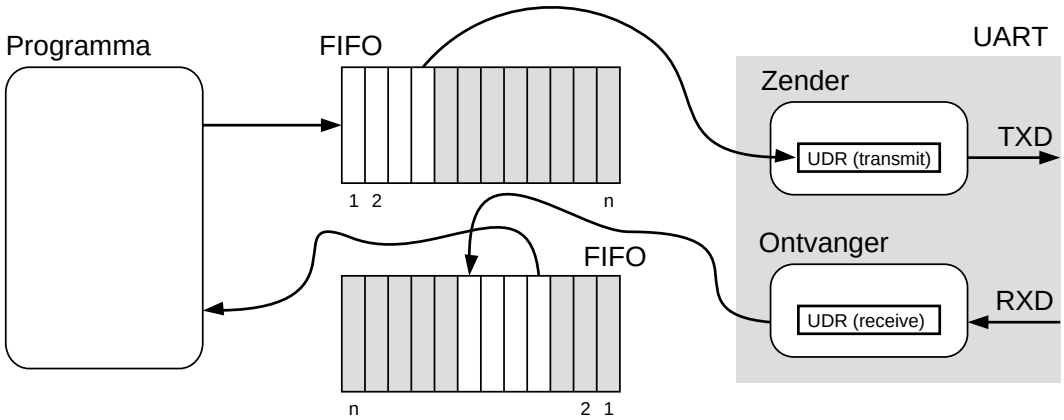
## 20.7 Het gebruik van een circulaire buffer

Analoog aan het ontvangen van gegevens met een interruptroutine, kan er ook een interruptroutine gemaakt worden voor het versturen van gegevens. De ontvangroutine zet nu de gelezen informatie in een globale variabele data en de verzendroutine plaatst data in UDR.

```
ISR(USART_RXC_vect)
{
  data = UDR;
}
```

```
ISR(USART_UDRE_vect)
{
  UDR = data;
}
```

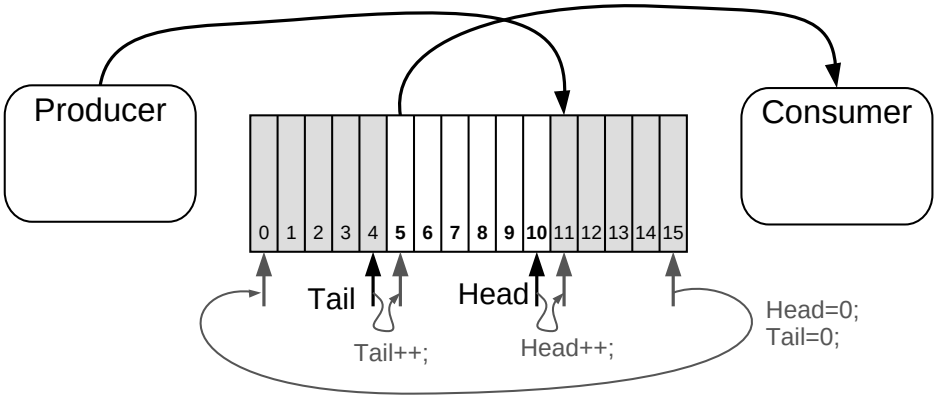
De verzendroutine wordt getriggerd als het UDR (transmit) register leeg is. Dat kan alleen als er geldige gegevens zijn om te versturen. Het zou wel heel toevallig zijn, als op het moment dat UDR leeg is, de variabele data ook weer nieuwe te versturen gegevens bevat. Bij het lezen van gegevens treedt hetzelfde probleem op. Als de vorige waarde van data nog niet verwerkt is, wordt deze overschreven als er weer een nieuwe waarde ontvangen is.
Een oplossing is om een buffer voor het lezen en het schrijven te gebruiken. In figuur 20.10 plaatst de ontvanger de gegevens in een buffer. Het programma leest de gegevens — op een moment dat het programma dat schikt — uit de buffer. Het programma zet de te verzenden gegevens ook in een buffer. De verzender verstuurt de gegevens, die klaar staan in de buffer.
In figuur 20.10 is als buffer een fifo (*first in - first out*) gebruikt. Bij het verzenden plaatst het programma de te verzenden byte aan het begin van de fifo. De zender van de UART verstuurt de byte die aan het einde van de fifo staat. Bij het toevoegen van bytes schuiven alle gegevens in de buffer één positie op. Dat is niet erg praktisch; zeker bij grote buffers kost het veel performance.




**Figuur 20.10 : Communicatie met de UART met behulp van twee fifo-buffers.** Het programma zet de te versturen bytes aan het begin van de fifo. De zender van de UART neemt de byte die aan het eind van de fifo staat. De ontvanger van de UART plaatst de te versturen databyte vooraan in de rij. Het programma pakt de bytes achteraan weg.

Bij het ontvangen is een andere methode gebruikt. De ontvanger zet de bytes vooraan in de buffer. Het programma haalt de bytes achteraan weg. De bytes in de buffer blijven op hun plaats. De pointers die naar het begin en het eind van de nog te verwerken gegevens wijzen, schuiven wel naar het eind op. Deze vorm van buffering werkt alleen goed als de buffer oneindig lang is. Bij een microcontroller is de hoeveelheid RAM en daarmee de maximale bufferlengte beperkt.

Dit communicatieprobleem van een of meer processen die data afgeven en een of meer processen die data opnemen, staat bekend als het *producer-consumer problem* of als het *bounded-buffer problem*. De oplossing is om een circulaire buffer of ringbuffer te gebruiken. Mits er gemiddeld evenveel gegevens uitgaan als ingaan en de buffer voldoende groot is, is de circulaire buffer te beschouwen als een oneindig grote buffer.




**Figuur 20.11 : Een circulaire buffer tussen een producent en een consument van gegevens.** De producent hoogt eerst `Head` op en voegt dan de nieuwe databyte toe. De consument hoogt eerst `Tail` op en leest dan de databyte uit de buffer. `Head` en `Tail` worden weer nul gemaakt als de maximale waarde bereikt is.

Figuur 20.11 toont een proces dat gegevens produceert, een circulaire buffer en een proces dat de gegevens consumeert. De kop van de rij is index `Head` en het eind van de rij is index `Tail`. De producent hoogt eerst positie `Head` met één op

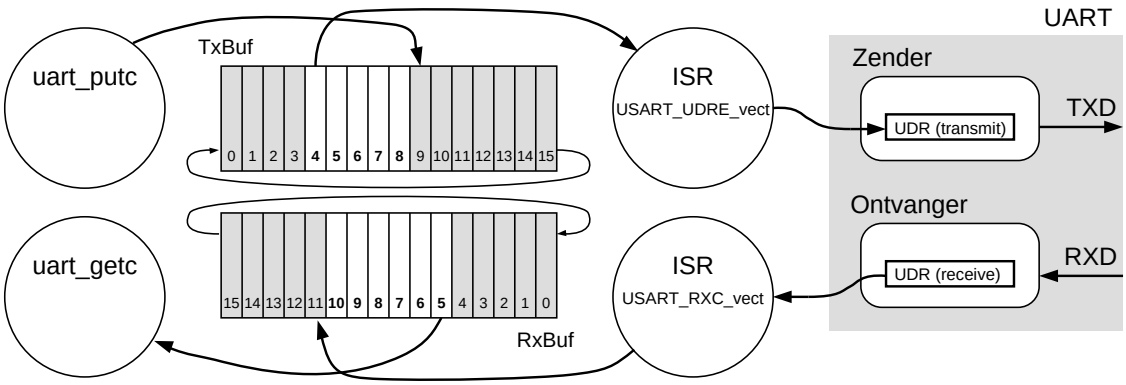
en zet dan de gegevens op deze nieuwe positie. De consument hoogt eerst positie `Tail` met één op en leest dan de gegevens van de nieuwe positie.
Als het einde van de buffer bereikt is, worden `Head` en `Tail` niet opgehoogd, maar juist nul gemaakt. Meestal is de lengte van de buffer een macht van twee. In dat geval is het ophogen en weer nul maken van de indices met een enkele uitdrukking te beschrijven:

```
Tail =  (Tail + 1) & BUFFER_MASK;
Head =  (Head + 1) & BUFFER_MASK;
```

De constante `BUFFER_MASK` hangt af van de grootte $n$ van de buffer en is gelijk aan $n-1$. Voor een bufferlengte 16 is `BUFFER_MASK` gelijk aan `0x0F`. De maskering maakt `Tail` en `Head` automatisch weer nul als het einde van de buffer is bereikt.
Bij deze aanpak wijst `Tail` naar de eerste positie achter de rij gegevens. Het testen op het vol of leeg zijn van de buffer is relatief eenvoudig. De buffer is leeg als `Tail` en `Head` gelijk zijn. De buffer is vol als `Tail` en de volgende waarde van `Head` gelijk zijn. Het maximale aantal elementen, dat de buffer kan bevatten, is daarom $n-1$.

## 20.8 Circulaire buffers bij de communicatie met een UART

Figuur 20.12 geeft de opbouw van de software voor de communicatie met een UART met behulp van twee circulaire buffers. Er is een leesfunctie `uart_getc`, er is een schrijffunctie `uart_putc` en er zijn twee interruptfuncties.




**Figuur 20.12 : De toepassing van circulaire buffers bij de communicatie met een UART.**
De ontvanger triggert `ISR(USART_RXC_vect)`. Deze routine zet de ontvangen byte in een buffer `RX_buf`. De functie `uart_getc` leest de bytes uit de buffer.
De functie `uart_putc` zet de te versturen gegevens in de buffer `TX_buf`. Als de zender klaar is, triggert deze `ISR(USART_UDRE_vect)`. Deze routine kopieert de te versturen byte naar de zender.

Het programma staat in code 20.4. De lengte van de buffers `RxBuf` en `TxBuf` is 16. Beide buffers hebben een index voor de kop en de staart, namelijk: `RxHead`, `TxHead`, `RxTail` en `TxTail`. De initialisatiefunctie `uart_init` activeert alleen de interruptvlag `RXCIE` en zet de vier indices op nul. De interruptvlag `UDRIE` wordt door de functie `uart_putc` aangezet.

De interruptfunctie `ISR(USART_RXC_vect)` wordt actief als de ontvanger een byte heeft ontvangen. Deze routine schuift de index `RxHead` een positie op en zet de ontvangen byte op die positie neer. De functie `uart_getc` schuift de index `RxTail` een positie op en retourneert de byte die op locatie `RxTail` staat. De functie

**Code 20.4 : Het ontvangen en versturen van gegevens met een circulaire buffer.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  #define BAUD        38400   // F_CPU is 8000000 Hz
5  #define UBRR_VALUE  ( ((F_CPU) + 8UL*(BAUD)) / \
6                        (16UL*(BAUD)) - 1UL )
7  #define RX_BUFFER_SIZE 16
8  #define TX_BUFFER_SIZE 16
9  #define RX_BUFFER_MASK ( RX_BUFFER_SIZE - 1 )
10 #define TX_BUFFER_MASK ( TX_BUFFER_SIZE - 1 )
11
12 static volatile unsigned char RxBuf[RX_BUFFER_SIZE];
13 static volatile unsigned char RxHead;
14 static volatile unsigned char RxTail;
15 static volatile unsigned char TxBuf[TX_BUFFER_SIZE];
16 static volatile unsigned char TxHead;
17 static volatile unsigned char TxTail;
18
19 void uart_init(unsigned int ubrr)
20 {
21   UBRRL  = ubrr;
22   UBRRH  = (ubrr >> 8);
23   UCSRC  = _BV(URSEL)|_BV(UCSZ1)|_BV(UCSZ0);
24   UCSRB  = _BV(RXEN)|_BV(TXEN)|_BV(RXCIE);
25   RxTail = 0;
26   RxHead = 0;
27   TxTail = 0;
28   TxHead = 0;
29 }
30
31 ISR(USART_RXC_vect)
32 {
33   RxHead = (RxHead + 1) & RX_BUFFER_MASK;
34   RxBuf[RxHead] = UDR;
35 }
36
37 unsigned char uart_getc(void)
38 {
39   while (RxHead == RxTail) ;
40   RxTail = (RxTail+1) & RX_BUFFER_MASK;
41
42   return RxBuf[RxTail];
43 }
44
```

```
45 ISR(USART_UDRE_vect)
46 {
47   if ( TxHead != TxTail ) {
48     TxTail = (TxTail+1) & TX_BUFFER_MASK;
49     UDR = TxBuf[TxTail];
50   } else {
51     UCSRB &= ~_BV(UDRIE);
52   }
53 }
54
55 void uart_putc(unsigned char data)
56 {
57   unsigned char tmp;
58
59   tmp = (TxHead+1) & TX_BUFFER_MASK;
60   while (tmp == TxTail) ;
61   TxBuf[tmp] = data;
62   TxHead = tmp;
63
64   UCSRB |= (1<<UDRIE);
65 }
66
67 void uart_puts(char *s)
68 {
69   char c;
70
71   while ( (c = *s++) ) {
72     uart_putc(c);
73   }
74 }
75
76 int main(void)
77 {
78   int c;
79
80   uart_init(UBRR_VALUE);
81
82   sei();
83
84   while(1) {
85     c = uart_getc();
86     uart_puts("Character: '");
87     uart_putc(c);
88     uart_puts("'\n");
89   }
90 }
```

`uart_putc` schuift de index `RxHead` een positie op, kopieert de te schrijven byte naar die nieuwe positie en zet de interruptvlag `UDRIE` aan. De interruptfunctie `ISR(USART_UDRE_vect)` wordt actief als het $\texttt{UDR}_{\text{transmit}}$ register leeg is. Deze routine schuift de index `TxTail` een positie op en kopieert de byte die op deze positie staat naar $\texttt{UDR}_{\text{transmit}}$.

Bij een circulaire buffer mag een producent geen data toevoegen als de buffer vol is en mag een consument geen data lezen als de buffer leeg is. De oplossing van code 20.4 test de meeste van deze situaties.

De interruptfunctie `ISR(USART_RXC_vect)` test niet op het vol zijn van de buffer. De buffer `RxBuf` moet voldoende groot zijn, anders gaan eerder ontvangen gegevens verloren. De functie `uart_getc` wacht op regel 39 tot er gegevens in de buffer staan. In plaats van te wachten is het soms beter alleen te testen op een lege buffer en bij een lege buffer een foutmelding terug te geven. Het hoofdprogramma kan dan gewoon verder gaan. De functie `uart_putc` wacht op regel 60 totdat er ruimte in buffer `TxBuf` is om gegevens weg te schrijven. De interruptfunctie `ISR(USART_UDRE_vect)` test of buffer `TxBuf` leeg is. Als deze leeg is, wordt de interruptvlag `UDRIE` uitgezet. Dit voorkomt dat de zender data gaat versturen als de buffer leeg is. Als `uart_putc` nieuwe data in de buffer zet, wordt `UDRIE` weer aangezet.

Het voorbeeld met de circulaire buffer is gebaseerd op *application note* AVR306 van Atmel. De UART-bibliotheek van Peter Fleury, die in paragraaf 20.9 wordt toegepast, is ook gebaseerd op dit document. De namen van variabelen, definities en functies in code 20.4 zijn aangepast aan de namen die Peter Fleury gebruikt.

Na de initialisatie van de UART en het aanzetten van de globale interrupt komt het hoofdprogramma in een oneindige lus. Deze lus leest een karakter en drukt daarna met `uart_puts` een tekst af met het gelezen karakter. De functie `uart_puts` maakt gebruik van `uart_putc` om een string te versturen. Als er geen karakter is ontvangen, blijft het programma — oftewel de functie `uart_getc` — wachten totdat er een karakter is ontvangen.

**Code 20.5 : Ontvangen en versturen van karakters met de UART-bibliotheek van Peter Fleury.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include "uart.h"
4
5  #define BAUD   38400   // F_CPU is 8000000 Hz
6
7  int main(void)
8  {
9    int c;
10
11   uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
12
13   sei();
14
15   while(1) {
16     if ( (c = uart_getc()) == UART_NO_DATA) {
17       continue;
18     }
19
20     uart_puts("Character: '");
21     uart_putc(c);
22     uart_puts("'\n");
23   }
24 }
```

## 20.9 De UART-bibliotheek van Peter Fleury

De site van Peter Fleury is: http://jump.to/fleury/ Deze site bevat naast de UART-bibliotheek ook een LCD- en een $I^2C$-bibliotheek.

Er bestaan verschillende UART-bibliotheken voor de `avr-gcc`-compiler. Deze zijn net als het vorig voorbeeld gebaseerd op de *application note* AVR306 van Atmel en passen ook twee circulaire buffers toe. Code 20.5 gebruikt de bibliotheek van Peter Fleury en is functioneel gelijk aan code 20.4.

Op regel 3 is een headerbestand `uart.h` ingevoegd. Daarin is onder andere de macro `UART_BAUD_SELECT` gedefinieerd die uit `F_CPU` en `BAUD` de waarde van het `UBRR`-register berekend. Deze macro wordt gebruikt op regel 11 om de juiste waarde voor `UBRR` mee te geven aan de functie `uart_init`.

De aanroep van `uart_getc` op regel 16 is anders dan die in code 20.4. De functie `uart_getc` van code 20.4 wacht op regel 39 totdat er een databyte is ontvangen. De `uart_getc` van Fleury test met een **`if`**-statement of er data in de buffer staan en geeft de foutmelding `UART_NO_DATA` terug als de buffer leeg is. De `uart_getc` van Fleury wacht niet. In dit geval is dat niet gewenst en is dit ondervangen door te controleren of er een databyte is gelezen. Alleen als er een databyte is gelezen worden de `put`-commando's uitgevoerd.

Procyon AVRlib van Pascal Stang is een uitgebreide bibliotheek voor `avr-gcc` en bevat ook verschillende UART-drivers, zie: http://hubbard.engr.scu.edu/embedded/avr/avrlib/

De UART-bibliotheek bestaat uit twee bestanden `uart.c` en `uart.h`. Plaats deze bestanden direct in de werkdirectory. Aan `uart.c` hoeft — of beter moet — niets worden gewijzigd. De bufferlengten zijn gedefinieerd in `uart.h`. Deze kunnen aangepast worden in `uart.h` of door ze mee te geven als preprocessoroptie (`-DUART_TX_BUFFER_SIZE=128`) aan de compiler. Bij een wijziging van `uart.h` moet `uart.c` opnieuw gecompileerd worden.

Net als bij de LCD-bibliotheek van Peter Fleury bestaat de UART-bibliotheek uit een c-bestand en een h-bestand. Het is dus geen echte bibliotheek.
Peter Fleury geeft aan dat zijn bestanden mogen verspreid en aangepast, mits voldaan wordt aan de *GNU General Public License*.
Op zich is dit goed, maar het gevolg is dat er op het internet heel veel verschillende versies van deze bibliotheken te vinden zijn.
Dit boek gebruikt `uart.c` versie `1.6.2.1` en `uart.h` versie `1.8.2.1`. Beide dateren van 1 juli 2007.

**Code 20.6 : Het versturen van getallen met de bibliotheek van Peter Fleury.**

```
 1  #include <avr/interrupt.h>
 2  #include <stdlib.h>
 3  #include "uart.h"
 4
 5  #define BAUD   38400   // F_CPU is 8000000 Hz
 6
 7  int main(void)
 8  {
 9    int  c;
10    char buffer[3];
11
12    uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
13
14    sei();
15
16    while(1) {
17      if ( (c = uart_getc()) == UART_NO_DATA) continue;
18      uart_puts("Character: '");
19      uart_putc(c);
20      uart_puts("' Hex: 0x");
21      itoa(c, buffer, 16);
22      uart_puts(buffer);
23      uart_putc('\n');
24    }
25  }
```

### Het versturen van getallen via de UART

Code 20.6 is identiek aan code 20.5, maar stuurt voor alle gelezen karakters ook de hexadecimale ASCII-waarde als string terug. Met `uart_getc` wordt een karakter `c` gelezen. De functie `itoa` converteert de waarde `c` naar een alfanumerieke string `buffer`. Het getal 16 geeft aan dat de representatie hexadecimaal is. De functie `uart_puts` verstuurt de string `buffer` via de UART naar buiten.

## 20.10 Het creëren van een stream voor printf en scanf

De `printf`- en `scanf`-functies uit de `avr-libc`-bibliotheek zijn zonder speciale acties niet zinvol. De microcontroller heeft immers geen toetsenbord en beeldscherm. Er is geen standaard in- en uitvoer aanwezig. De functies `printf` en `scanf` weten niet waar de informatie naar toe moet worden geschreven of waar deze moet worden gelezen.
Wel is het mogelijk om een eigen `printf` of `scanf` te maken, die een alternatieve invoer of uitvoer gebruikt. De functie `FDEV_SETUP_STREAM` uit `stdio.h` creëert een `FILE`-structuur. Voor de uitvoer heeft `FDEV_SETUP_STREAM` een `fputc` nodig om gegevens te versturen en voor de invoer een `fgetc` om gegevens te ontvangen.

**Code 20.7 : Het gebruik van `FDEV_SETUP_STREAM` om een eigen `printf` te maken.**

```
1  #include <avr/interrupt.h>
2  #include <stdio.h>
3  #include "uart.h"
4
5  #define BAUD   38400   // F_CPU is 8000000 Hz
6
7  int uart_fputc(char c, FILE *stream);
8
9  FILE uart_stdout = FDEV_SETUP_STREAM(uart_fputc,  NULL, _FDEV_SETUP_WRITE);
10
11 int uart_fputc(char c, FILE *stream)
12 {
13   uart_putc(c);
14   return 0;
15 }
16
17 int main(void)
18 {
19   int  c;
20
21   uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
22   stdout = &uart_stdout;
23
24   sei();
25
26   while(1) {
27     if ( (c = uart_getc()) == UART_NO_DATA) continue;
28     printf("Character: '%c' Hex: %#x\n", c, c);
29   }
30 }
```

Code 20.7 gebruikt een eigen `printf`. Op de regels 11 tot en met 15 staat een functie `uart_fputc` die met behulp van de functie `uart_putc` uit de bibliotheek van Fleury een karakter naar de UART schrijft. Het prototype van `uart_fputc` staat op regel 7 en heeft net als de standaard `fputc` twee ingangsvariabelen van het type **`char`** en `FILE *` en een retourtype **`int`**.

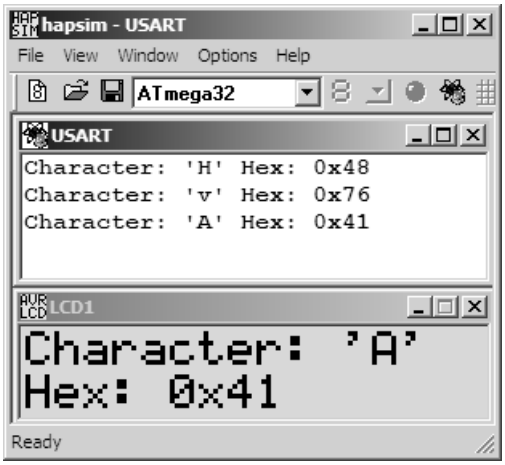
De functie `FDEV_SETUP_STREAM` creëert op regel 9 een `FILE`-structuur `uart_stdout`. De toekenning van regel 22 zorgt er voor dat standaard filepointer `stdout` naar deze `FILE`-structuur wijst. De `printf` van regel 28 stuurt een geformatteerde string naar de UART.

De oneindige lus van het hoofdprogramma leest een karakter van de UART en drukt dit karakter en de hexadecimale waarde van dit karakter af.

**Code 20.8 : Het gebruik van `FDEV_SETUP_STREAM` om een eigen `printf` en `scanf` te maken.**

```
1  #include <avr/interrupt.h>
2  #include <stdio.h>
3  #include "uart.h"
4
5  #define BAUD   38400   // F_CPU is 8000000 Hz
6
7  int uart_fputc(char c, FILE *stream);
8  int uart_fgetc(FILE * stream);
9
10 FILE uart_stdinout = FDEV_SETUP_STREAM(uart_fputc, uart_fgetc, _FDEV_SETUP_RW);
11
12 int uart_fputc(char c, FILE *stream)
13 {
14   if (c == '\n') uart_putc('\r');
15   uart_putc(c);
16   return 0;
17 }
18
19 int uart_fgetc(FILE * stream)
20 {
21   int c;
22
23   while ( (c = uart_getc()) == UART_NO_DATA) ;
24   return c;
25 }
26
27 int main(void)
28 {
29   char  c;
30
31   uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
32   stdout = stdin = &uart_stdinout;
33
34   sei();
35
36   while(1) {
37     scanf("%c", &c);
38     printf("Character: '%c' Hex: %#x\n", c, c);
39   }
40 }
```

Code 20.8 definieert een `FILE`-structuur `uart_stdinout` met een zend- en een ontvangfunctie. De zendfunctie `uart_fputc` is identiek aan die uit code 20.7, alleen is er een **`if`** toegevoegd die bij een *end-of-line* (`'\n'`) een extra *carriage return* (`'\r'`) verstuurd. Nieuwe regels beginnen dan bij de Window's HyperTerminal ook op het begin van de regel.
De ontvangfunctie `uart_fgetc` gebruikt `uart_getc` uit de bibliotheek van Fleury en blijft wachten totdat er een karakter gelezen is. De `FILE`-structuur `uart_stdinout` wordt op regel 32 gekoppeld aan zowel `stdout` als aan `stdin`. De oneindige lus van het hoofdprogramma bevat een `scanf` die een karakter leest en een `printf` die het karakter en de hexadecimale waarde van het karakter afdrukt.
Het is mogelijk om meerdere *streams* te gebruiken. Het programma van code 20.9 leest karakters van de UART en schrijft deze net als in code 20.8 geformatteerd terug, maar stuurt dezelfde informatie ook naar een LCD. Op regel 12 en 13 zijn twee `FILE`-structuren gedefinieerd. In het hoofdprogramma worden deze twee structuren op regel 41, 42 en 44 gekoppeld aan de drie filepointers `uart_in`, `uart_out` en `lcd_out`.
Op regel 49 wordt met `fscanf` een karakter van de UART gelezen. Dit karakter wordt op regel 50 met `fprintf` geformatteerd naar de UART en op regel 52 naar de LCD geschreven. In het eerste geval is de filepointer van `fprintf` `uart_out` en in het tweede geval is dat `lcd_out`. Een simulatieresultaat van het programma staat in figuur 20.13.




**Figuur 20.13 : De simulatie van code 20.9.** Bovenaan staat de uitvoer van de UART en onderaan de uitvoer op het LCD.

De *format string* van de `fprintf` voor het LCD heeft achter de eerste `%c` een extra *end-of-line*. Het eerste gedeelte van de af te drukken string komt op de eerste regel van het display en het tweede deel op de tweede regel. Op regel 51 wordt het LCD leeggemaakt. Dit commando is bij de invoer van afdrukbare karakters niet nodig omdat het karakter en de hexadecimale waarde altijd even breed is. Alleen als er een niet afdrukbaar karakter wordt ingevoerd (bijvoorbeeld `^J`, `^M`) komt er ook tekst op andere posities terecht.
Of een `printf` voor een LCD hier nuttig is, is de vraag. De meeste tekst blijft hetzelfde. Het is in principe voldoende om alleen het karakter en de hexadecimale waarde te herschrijven. Dat kan met `lcd_gotoxy` en `lcd_putc` gedaan worden.
Het schrijven naar de UART kan ook met een `sprintf` en een `uart_puts` gedaan worden op de manier zoals dat in paragraaf 19.7 voor een LCD is gedaan.

Code 20.9 : Een eigen **printf** en **scanf** voor de UART en een eigen **printf** voor het LCD.

```
1  #include <avr/interrupt.h>
2  #include <stdio.h>
3  #include "uart.h"
4  #include "lcd.h"
5
6  #define BAUD   38400   // F_CPU is 8000000 Hz
7
8  int uart_fputc(char c, FILE *stream);
9  int uart_fgetc(FILE * stream);
10 int lcd_fputc(char c, FILE *stream);
11
12 FILE uart_stdinout = FDEV_SETUP_STREAM(uart_fputc, uart_fgetc, _FDEV_SETUP_RW);
13 FILE lcd_stdout    = FDEV_SETUP_STREAM(lcd_fputc, NULL, _FDEV_SETUP_WRITE);
14
15 int uart_fputc(char c, FILE *stream)
16 {
17   uart_putc(c);
18   return 0;
19 }
20
21 int uart_fgetc(FILE * stream)
22 {
23   int c;
24
25   while ( (c = uart_getc()) == UART_NO_DATA) ;
26   return c;
27 }
28
29 int lcd_fputc(char c, FILE *stream)
30 {
31   lcd_putc(c);
32   return 0;
33 }
34
35 int main(void)
36 {
37   char  c;
38   FILE *uart_in, *uart_out, *lcd_out;
39
40   uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
41   uart_in  = &uart_stdinout;
42   uart_out = &uart_stdinout;
43   lcd_init(LCD_DISP_ON);
44   lcd_out = &lcd_stdout;
45
46   sei();
47
48   while(1) {
49     fscanf(uart_in, "%c", &c);
50     fprintf(uart_out, "Character: '%c' Hex: %#x\n", c, c);
51     lcd_clrscr();
52     fprintf(lcd_out, "Character: '%c'\nHex: %#x\n", c, c);
53   }
54 }
```

# 21

# EEPROM en seriële communicatie

Doelstelling

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.

Onderwerpen

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 niet-vluchtige 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)`<br>Deze functie leest een byte van het EEPROM-adres `addr`. |
| `eeprom_read_word` | `uint8_t eeprom_read_word (const uint16_t *addr)`<br>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)`<br>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)`<br>Deze functie schrijft een byte `value` naar het EEPROM-adres `addr`. |
| `eeprom_write_word` | `void eeprom_write_word (uint16_t *addr, uint16_t value)`<br>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)`<br>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.

**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 }
```

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.

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.

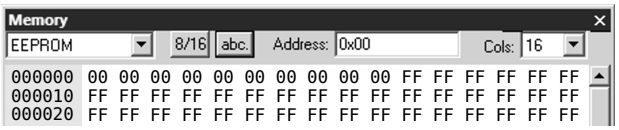
In code 21.4 is op regel 5 een array v met dit attribuut gedeclareerd. Deze code is functioneel identiek met code 21.1. Alleen bepaalt de compiler nu de plaats waar v in het EEPROM komt te staan. De adreslocatie die op regel 14 en 20 aan de functies eeprom_write_byte en eeprom_read_byte wordt meegegeven,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

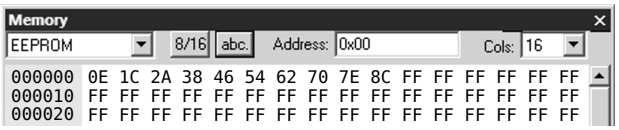
het EEPROM geprogrammeerd kan worden. Omdat bij de declaratie van variabele v geen beginwaarden staan, worden de tien bytes ingevuld met nullen. Het eep-bestand gebruikt hetzelfde Intel Hex-formaat als het hex-bestand en bevat deze tekst:

```
:0A000000000000000000000000F6
: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.




**Figuur 21.1 : Het EEPROM na het laden.**




**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.

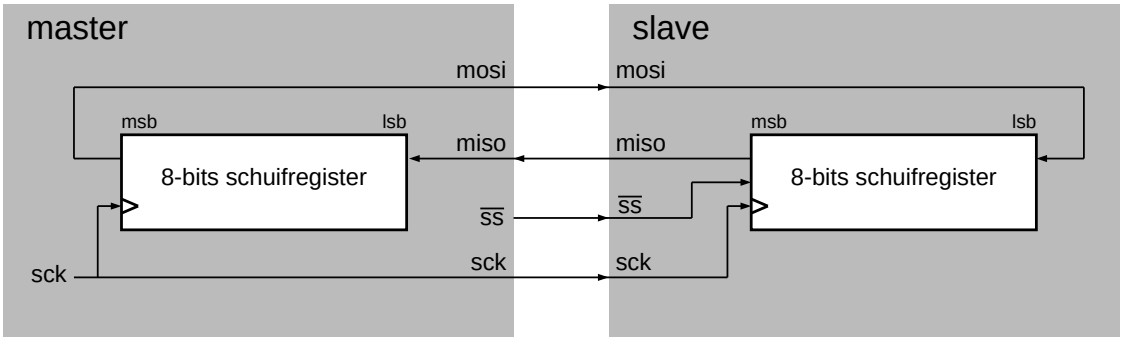
**Code 21.6 : Het initialiseren van en het lezen uit het flashgeheugen.**

```
 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 $I^2C$-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.



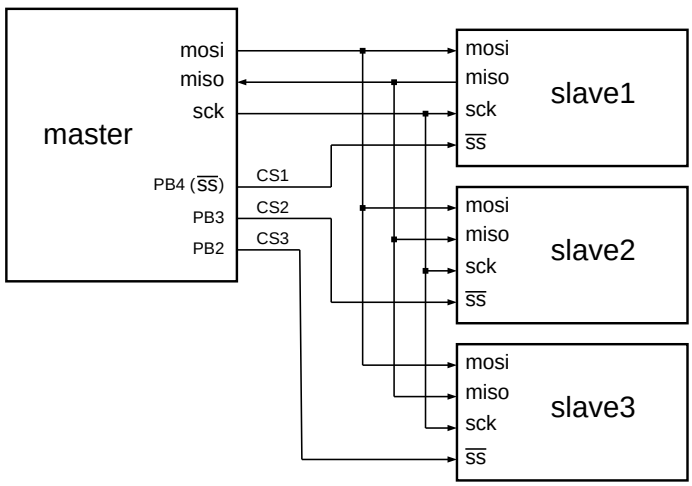

**Figuur 21.3 :** **De SPI met de master en een slave.** De `MOSI` van de master is aangesloten aan de `MOSI` van de slave. De `MISO` van de slave is verbonden met de `MISO` van de master. De schuifregisters van de master en de slave vormen samen een roterend schuifregister. De master genereert de klok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

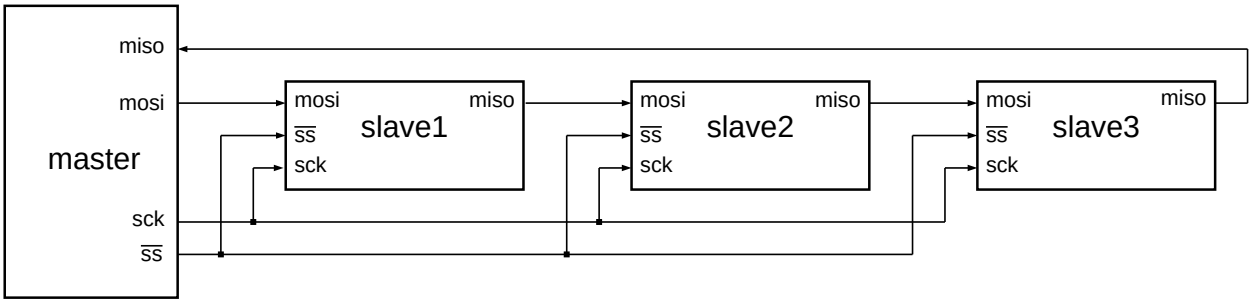
De `MISO`, *Master in Slave Out* van de master is verbonden met de `MISO` van de slave en de `MOSI`, *Master Out Slave In* van de master is verbonden met de `MOSI` van de slave. Hierdoor ontstaat een groot, roterend schuifregister. Na acht klokslagen staat de waarde van de master in het schuifregister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

## 21.3 I²C

Spreek I²C 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.

I²C 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. I²C 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 I²C-interface en gaf de mogelijkheid aan andere chipfabrikanten om — in licentie — deze interface toe te passen.

Binnen het I²C-protocol heeft elke type component een unieke identificatiecode. Philips stelt deze codes in licentie ter beschikking. Als de microcontroller master

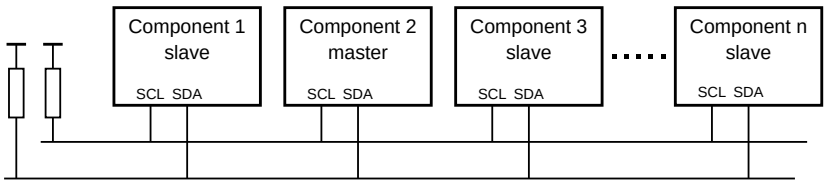
is, heeft de I²C-interface geen eigen identificatiecode nodig. Als de microcontroller slave is, heeft het een slave-adres nodig. Microcontrollerfabrikanten laten dit aan de ontwerper over. Deze kan dan zelf een code kiezen. De ATmega32 heeft wel een I²C-interface, maar heeft deze niet aangemeld bij Philips. 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*

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.

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.

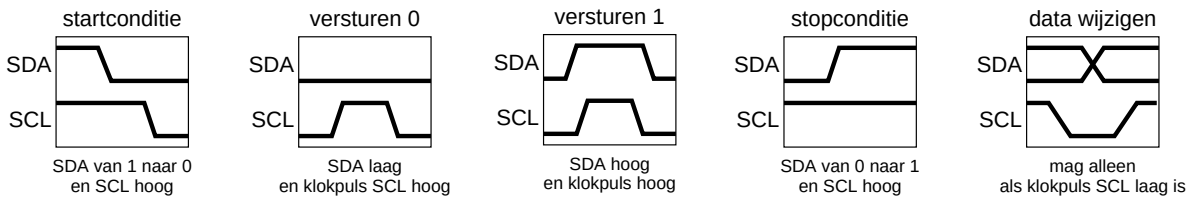
Componenten, die met een I²C-interface verkrijgbaar zijn, zijn bijvoorbeeld: SDRAM, EEPROM, flash, port expander, DAC, ADC, *real time clock*, LCD, temperatuursensor en vele andere sensoren.




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.



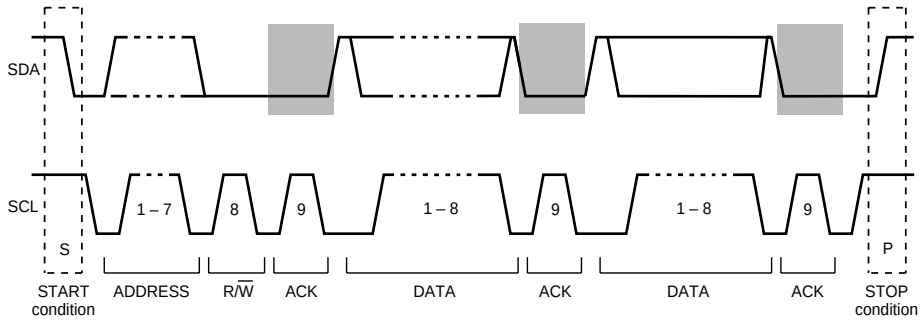

**Figuur 21.7 : De betekenis van de verschillende signaalcondities met de `SDA`- en de `SCL`-lijn.**

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.

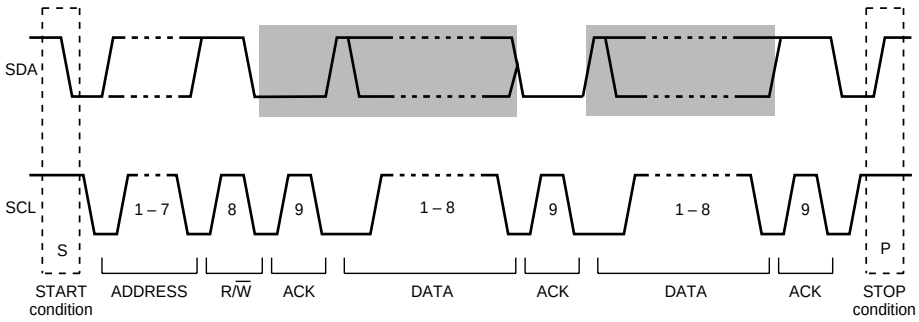
Figuur 21.9 toont de signaalwaarden van het protocol van figuur 21.8. Het kloksignaal wordt door de master gegenereerd. De slave zet de bevestigingsbits op de datalijn. De rest van het datasignaal maakt de master.

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` ($\overline{\texttt{ACK}}$). 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` ($\overline{\texttt{ACK}}$) 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$^2$C-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.

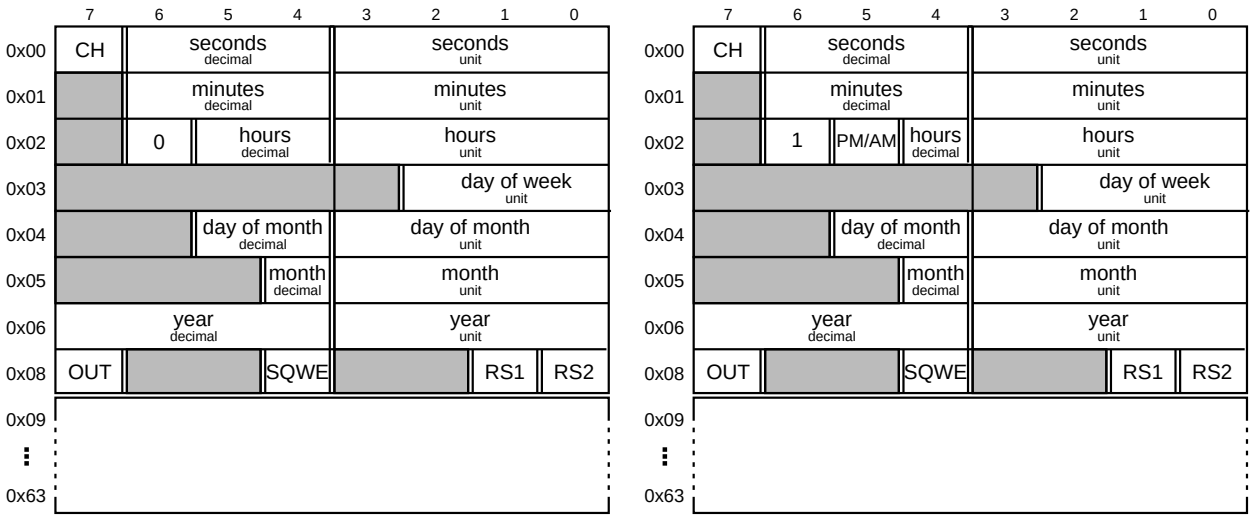
Code 21.12 : De I²C-functies uit `i2c.c` voor het versturen en ontvangen.

```
#include "i2c.h"

void i2c_init(void)
{
  TWCR = 0x00;
  TWAR = 0x00;                          // slave address
  TWSR = 0x00;                          // no prescaling
  TWBR = ((uint8_t)(F_CPU/F_SCL) - 16)/2;
  if (TWBR < 10 ) {                     // TWBR must be 10 or more
    TWBR = 10;
  }
}

uint8_t i2c_start(void)
{
  TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWSTA);
  while ( !(TWCR & _BV(TWINT)) );

  if ( TW_STATUS != TW_START ) return 1;
  return 0;
}

uint8_t i2c_restart(void)
{
  TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWSTA);
  while ( !(TWCR & _BV(TWINT)) );

  if ( TW_STATUS != TW_REP_START ) return 1;
  return 0;
}

void i2c_stop(void)
{
  TWCR =  _BV(TWINT)|_BV(TWEN)|_BV(TWSTO);
  while ( (TWCR & _BV(TWSTO)) );
}

uint8_t i2c_write(uint8_t data)
{
  TWDR = data;
  TWCR = _BV(TWINT)|_BV(TWEN);
  while ( !(TWCR & _BV(TWINT)) );

  if( TW_STATUS != TW_MT_DATA_ACK) return 1;
  return 0;
}

uint8_t i2c_read(uint8_t ack)
{
  TWCR = _BV(TWINT)|_BV(TWEN)|( ( ack == I2C_ACK) ? _BV(TWEA) : 0 );
  while ( !(TWCR & _BV(TWINT)) );

  return TWDR;
}
```

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/`$\overline{\texttt{W}}$-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$^2$C-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$^2$C 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 12-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.**

```
#define RTC_SLAVE_ADDRESS   0xD0

#define RTC_SECOND          0x00
#define RTC_MINUTE          0x01
#define RTC_HOUR            0x02
#define RTC_DAY             0x03
#define RTC_DATE            0x04
#define RTC_MONTH           0x05
#define RTC_YEAR            0x06
#define RTC_CONTROL         0x07

void  rtc_set_time(void);
void  rtc_set_date(void);
void  rtc_get_time(void);
void  rtc_get_date(void);
char *rtc_time_to_string(char *s);
void  string_to_rtc_time(char *s);

struct rtc_time {
  uint8_t second;
  uint8_t minute;
  uint8_t hour;
};

struct rtc_date {
  uint8_t day;
  uint8_t month;
  uint8_t year;
};
```

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

```
#include "i2c.h"
#include "rtc.h"

struct rtc_time time;
struct rtc_date date;

void rtc_set_time(void)
{
  i2c_start();
  i2c_write(RTC_SLAVE_ADDRESS|I2C_WRITE);
  i2c_write(RTC_SECOND);
  i2c_write(time.second);
  i2c_write(time.minute);
  i2c_write(time.hour);
  i2c_stop();
}

void rtc_get_date(void)
{
  i2c_start();
  i2c_write(RTC_SLAVE_ADDRESS|I2C_WRITE);
  i2c_write(RTC_DATE);
  i2c_restart();
  i2c_write(RTC_SLAVE_ADDRESS|I2C_READ);
  date.day   = i2c_read(I2C_ACK);
  date.month = i2c_read(I2C_ACK);
  date.year  = i2c_read(I2C_NACK);
  i2c_stop();
}
```

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:

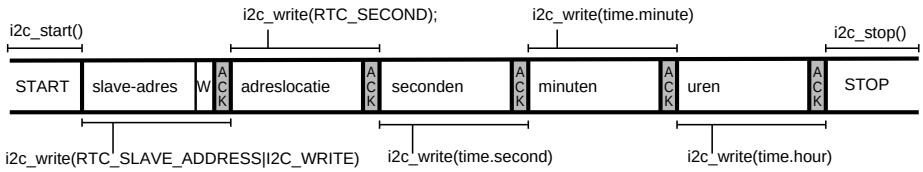
```
uint8_t ds1307[7];
```

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`.

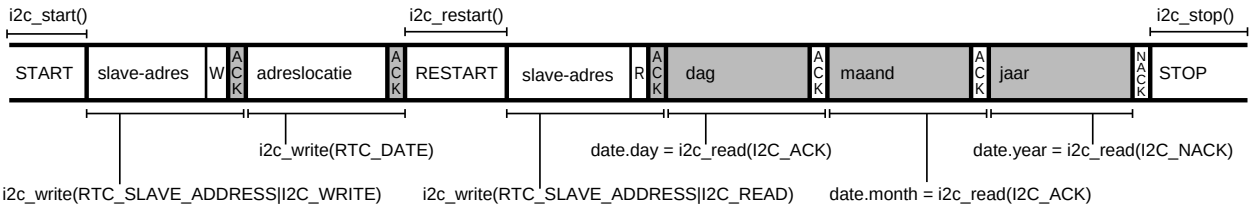
Voor het instellen van een DS1307 real time clock wordt na het slave-adres en het schrijfbit eerst de adreslocatie verstuurd van waar af de bytes ingesteld moet worden. Na het adres van de geheugenlocatie volgen de bytes.
De functie `rtc_set_time` uit code 21.14 stelt de tijd van de DS1307 in. Na het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

```
char *rtc_time_to_string(char *s)
{
  s[0] = 0x30 | ((time.hour & 0x30) >> 4);
  s[1] = 0x30 |  (time.hour & 0x0F);

  s[3] = 0x30 | ((time.minute & 0x70) >> 4);
  s[4] = 0x30 |  (time.minute & 0x0F);

  s[6] = 0x30 | ((time.second & 0x70) >> 4);
  s[7] = 0x30 |  (time.second & 0x0F);

  return s;
}
```

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

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.

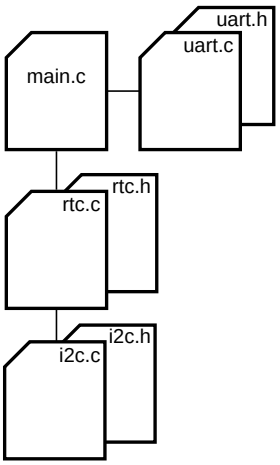
De functie `string_to_rtc_time` doet het omgekeerde. De karakters `s[1]`, `s[4]` en `s[7]` bevatten de eenheden van de uren, minuten en seconden. Het masker `0x0F` geeft alleen het lage *nibble* door. Dit wordt samengevoegd met de relevante bits van de tientallen, die vier posities naar links zijn geschoven.

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

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "uart.h"
#include "i2c.h"
#include "rtc.h"

#define BAUD   38400   // F_CPU is 8000000 Hz

int main(void)
{
  char t[]="HH:MM:SS";

  i2c_init();
  uart_init(UART_BAUD_SELECT(BAUD,F_CPU));
  sei();

  string_to_rtc_time("08:53:38");
  rtc_set_time();
  while(1) {
    rtc_get_time();
    uart_puts(rtc_time_to_string(t));
    _delay_ms(1000);
  }
}
```

Het hoofdprogramma uit code 21.16 stelt, na de initialisatie van de I$^2$C-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 I$^2$C-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 I$^2$C 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.**

# 22

# Pulsbreedtemodulatie

Doelstelling

Dit hoofdstuk behandelt de pulsbreedtemodulatie. Het is een aanvulling op hoofdstuk 13 over de timers. De mogelijkheden om met behulp van de timers van de ATmega32 een pulsbreedtegemoduleerd signaal te maken, worden besproken.

Onderwerpen

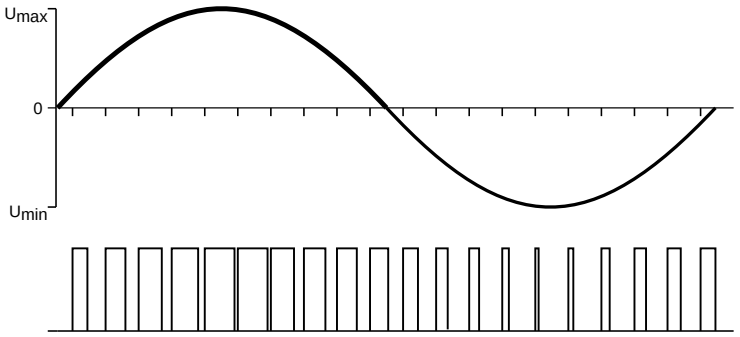
De behandelde onderwerpen zijn:

- Het principe van pulsbreedtemodulatie, PWM (*Pulse Width Modulation*).
- De timers van de ATmega32.
- De verschillende soorten technieken om met de timers een PWM-signaal te maken.
- De fast-, phase-correct-, phase-and-frequency-correct-, CTC- en de normale modus.
- De waveformgeneratoren van de timers met de mogelijkheden voor de uitgangssignalen.
- De rgb-led.
- DC-motoren en servomotoren.
- De aansturing van motoren met behulp van een H-brug.
- De aansturing van een luidspreker, een piëzo-elektrische en een magnetische buzzer.

Het gebruik van PWM-signalen wordt gedemonstreerd met deze voorbeelden:

- Het regelen van de led-intensiteit met fast-PWM.
- Het aansturen van DC-motoren bij een robotwagen met phase-correct-PWM.
- Het aansturen van servomotoren bij robots met phase-and-frequency-correct-PWM.
- Het afspelen van muziek met behulp van de CTC-modus.

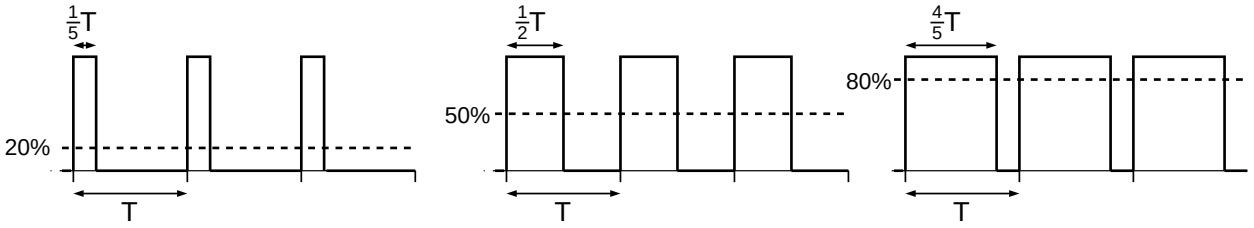
Pulsbreedtemodulatie is een modulatietechniek waarbij de informatie wordt vastgelegd in de breedte van de puls van een pulsvormig signaal. Het signaal kent een hoog en een laag signaalniveau en de frequentie ligt vast. In figuur 22.1 komt de breedte van de puls overeen met de grootte van het sinusvormige signaal. Bij het maximum van de sinus zijn de pulsen breed en bij het minimum zijn de pulsen smal. De breedte van de pulsen bevat alle informatie om het sinusvormige signaal te kunnen reconstrueren.
Pulsbreedtemodulatie of PWM, *Pulse Width Modulation*, wordt gebruikt voor onder andere communicatie, de aansturing van servomotoren, de aansturing van gelijkstroommotoren en om geluid en muziek te maken en leds te dimmen.




**Figuur 22.1 : Pulsbreedtemodulatie van een sinusvormig signaal.**

Een led, die via een vaste weerstand aangesloten is op een microcontroller, brandt altijd even fel. De uitgangsspanning van de microcontroller is laag (0 V) of hoog (5 V of 3,3 V). Tussenliggende waarden zijn er niet. Met pulsbreedtemodulatie is dit probleem te omzeilen.

Figuur 22.2 toont drie PWM-signalen met steeds een andere pulsbreedte. Als de puls $\frac{1}{5}$ van de periodetijd $T$ hoog is, is het signaalniveau gemiddeld $\frac{1}{5}$ van de maximum waarde. Bij een maximum van 5 V is het gemiddelde signaalniveau 1 V, oftewel 20%. Een pulsbreedte van $\frac{1}{2}T$ geeft een signaalniveau van 50% en een pulsbreedte van $\frac{4}{5}T$ geeft een signaalniveau van 80%. Mits de frequentie van het signaal voldoende hoog is, bijvoorbeeld 1 kHz, is het aan en uit gaan van de led niet zichtbaar en brandt de led zwakker.




**Figuur 22.2 : Drie PWM-signalen met een verschillende pulsbreedte.**

Het gemiddelde signaalniveau komt overeen met de *duty cycle*. Per definitie is dit de pulsduur, of de breedte van de puls, $T_{\text{pw}}$ gedeeld door de periodetijd $T$:

$$\Delta = \frac{T_{\text{pw}}}{T} \tag{22.1}$$

De *duty cycle* is de relatieve pulsduur. In het Nederlands wordt dit ook duty-cycle genoemd.

In principe is de *duty cycle* $\Delta$ een getal tussen 0 en 1, maar meestal wordt dit uitgedrukt in procenten. Een signaal met een periodetijd van 100 $\mu$s en een pulsbreedte van 25 $\mu$s heeft een duty-cycle van 0,25 of 25%.
Een led, die op de gebruikelijke manier op een uitgang — zodanig dat de microcontroller de stroom afvoert — aangesloten is, zal als de uitgang een PWM-signaal is met een duty-cycle van 20%, 80% van de tijd aan zijn. Bij een pulsduur van 80% brandt de led slechts 20% van de tijd en zal dan duidelijk minder fel branden.

## 22.1 De timers van de ATmega32

Tijd speelt bij embedded systemen een belangrijke rol. Elke microcontroller heeft daarom een of meer timers. De microcontroller gebruikt deze timers of tellers om:

- gebeurtenissen te tellen;
- een tijdsduur te definiëren;
- een periodetijd of een pulsduur te meten;
- een PWM-signaal te genereren;
- een frequentiegenerator te maken.

De timers van de ATmega32 kennen alle drie in ieder geval deze modi:

- de normale modus,
- *Clear Timer on Compare match* (CTC),
- *fast PWM*,
- *Phase correct PWM*.

Timer 1 kent naast deze vier modi ook een *Phase and frequency correct PWM* en een *Input Capture* modus.

In hoofdstuk 13 zijn timer 0 en timer 2 toegepast in de normale modus om een tijdsduur te definiëren. Ook voor het tellen van gebeurtenissen is de normale modus geschikt. De *Input Capture*-modus wordt gebruikt om de periodetijd of een pulsduur van een extern signaal te bepalen. De CTC-modus is geschikt voor het maken van een blokgolf met een specifieke frequentie en een duty-cycle van 50%. De andere PWM-modi zijn juist handig voor het maken van een PWM-signaal met een vaste frequentie en een specifieke pulsduur. De PWM-modus wordt ingesteld met de *waveform generation mode*-bits. Deze WGM-bits staan in de TCCR-registers of besturingsregisters van de timers.

Bij de voorbeelden van hoofdstuk 13 is de normale modus gebruikt. Het blokschema van timer 0 is daar grotendeels besproken en staat in figuur 13.2 van paragraaf 13.1. De comparator, het *compare*-register OCR0, en de waveformgenerator zijn in hoofdstuk 13 niet gebruikt. Dit zijn juist belangrijke onderdelen voor het creëren van PWM-signalen. Elke timer heeft minstens één, zogenoemde *output-compare*-uitgang of OC-uitgang. De comparator vergelijkt de waarde uit het OCR-register met de huidige waarde van teller TCNT. Als er een *match* is, bepaalt de waveformgenerator de waarde van de uitgang OC.
De ATmega32 heeft in het totaal vier van deze uitgangen: OC0, OC1A, OC1B en OC2. Pin PB3 is de OC0-uitgang van timer 0 en pin PD7 is de OC2-uitgang van timer 2. Timer 1 heeft twee comparatoren, twee waveformgeneratoren en twee uitgangen. De uitgang OC1A valt samen met PD5 en uitgang OC1B met pin PD4.
De uitgangen worden aangestuurd door de waveformgeneratoren, die ingesteld worden met de *Compare match Output Mode*-bits uit de TCCR-registers. De betekenis van de verschillende COM-bits is bij elke modus anders. De volledige beschrijving van de COM-bits is daarom omvangrijk. Bijlage G geeft een bondig overzicht van de TCCR-registers en de bijbehorende WGM- en COM-bits.

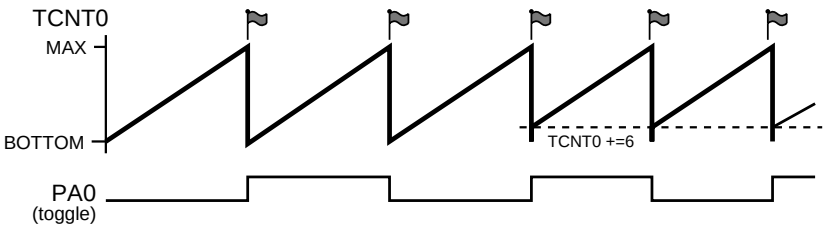
Timer 0 en 2 hebben allebei twee interruptsignalen, namelijk een *timer overflow flag* (TOV0 en TOV2) en een *output compare flag* (OCF0 en OCF2). Timer 1 heeft vier interruptsignalen: een *timer overflow flag* (TOV1), twee *output compare flags* (OCF1A en OCF1B) en een *input compare flag* (ICF1).

## 22.2 De beschrijving van de modi van timers

In grote lijnen is de functionaliteit van de drie timers identiek. Toch zijn er ook verschillen, vooral timer 1 heeft veel meer mogelijkheden. Dat is een 16-bits teller met een *input compare*-ingang en twee *output compare*-uitgangen. Deze teller kent bovendien een *Phase and frequency correct PWM* modus.
De voorbeelden van deze paragraaf gebruiken voornamelijk timer `0`. Alles wat voor deze timer geldt, is ook bij timer 1 en timer 2 toepasbaar. Timer 1 heeft een aantal unieke eigenschappen. De voorbeelden, die deze eigenschappen demonstreren, gebruiken natuurlijk deze timer.

### De normal modus

In de normale modus worden de comparator, register `OCR0`, uitgang `OC0` en de waveformgenerator niet gebruikt. Om een PWM-signaal te maken, is de ontwerper genoodzaakt zelf een pin van de microcontroller aan te sturen. Figuur 22.3 toont de waarde van register `TCNT0` voor de normale modus. De teller telt van `BOTTOM` tot `TOP`. `TOP` is bij de normale modus altijd de maximale waarde `MAX`. Voor timer `0` is dat `0xFF`. `BOTTOM` is altijd nul, bij timer `0` is dat `0x00`. De teller telt bij `MAX` gewoon verder. Vanwege de beperkte bitbreedte wordt de volgende waarde `BOTTOM`.




**Figuur 22.3** : **De waarde van `TCNT0` bij timer `0` in de normale modus.** Teller `TCNT0` loopt steeds van `BOTTOM` tot `MAX`. Bij de overgang van `MAX` naar `BOTTOM` wordt de `TOV0`-vlag hoog. Dit is aangegeven met een vlaggetje.
Signaal `PA0` is het signaal dat op pin `0` van poort A ontstaat, als er een interruptfunctie is die de uitgang laat omklappen. Bij de streeplijn is in de `ISR` de waarde van `TCNT0` opgehoogd, zodat de frequentie hoger is.

Bij de overgang van `MAX` naar `BOTTOM` wordt de overflow-vlag (`TOV0`) hoog. Als bovendien de betreffende interruptvlag (`TOIE0`) hoog is en de globale interrupt actief is, wordt de interruptfunctie uitgevoerd. De ISR kan dan bijvoorbeeld de uitgang `0` van `PORTA` laten omklappen:

```
ISR(TIMER0_OVF_vect)
{
  PORTA = PORTA ^ _BV(0);
}
```

Er ontstaat een PWM-signaal met een duty-cycle van 50%, zoals figuur 22.3 laat zien. De frequentie van het signaal hangt af van $f_{\text{cpu}}$ en de prescaling $P$. Er zijn twee interrupts binnen een periode. Met formule 13.2 en het gegeven dat $m$ gelijk is aan de maximale waarde, is de frequentie $f_{\text{normal}}$:

$$f_{\text{normal}} = \frac{f_{\text{cpu}}}{2\,m\,P} = \frac{f_{\text{cpu}}}{512\,P} \tag{22.2}$$

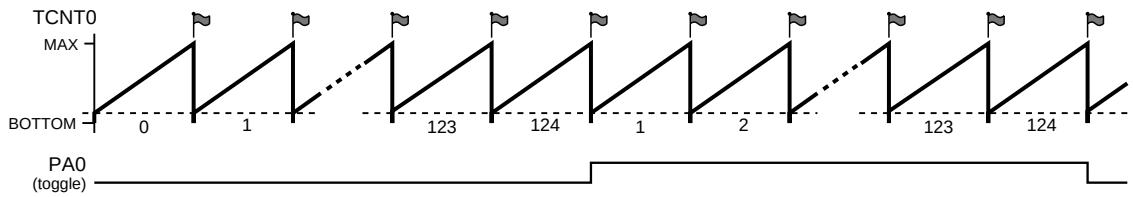
Als er een andere frequentie nodig is, kan de prescaling worden aangepast. Meer mogelijkheden ontstaan er door in de ISR de waarde van `TCNT0` te verhogen.

```
ISR(TIMER0_OVF_vect)
{
  TCNT0 += 6;
  PORTA = PORTA ^ _BV(0);
}
```

In paragraaf 13.4 is deze methode al gebruikt en in figuur 22.3 is dit gevisualiseerd. In code 13.1 is bovendien de frequentie aanmerkelijk verlaagt door een keer per $n$ interrupts de pin te laten omklappen. De frequentie $f_{\text{normal,n}}$ van het signaal hangt af van $f_{\text{cpu}}$, de prescaling $P$, het aantal interrupts `n` en het aantal stappen `m`:

$$f_{\text{normal,n}} = \frac{f_{\text{cpu}}}{2\,n\,m\,P} \tag{22.3}$$

Figuur 22.4 toont de verandering van `TCNT0` en `PA0` voor deze situatie. In paragraaf 13.4 is $f_{\text{cpu}} = 4$ MHz, $P$ =64, $n = 125$ en $m = 250$, zodat de frequentie $f_{\text{normal,n}}$ van het signaal gelijk is aan 1 Hz.



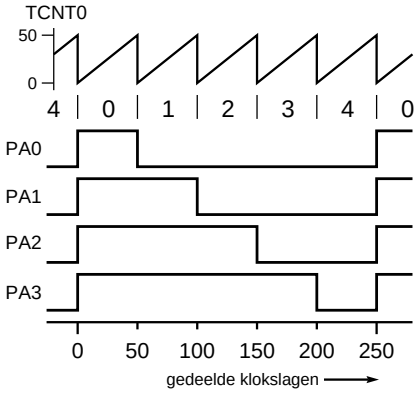

**Figuur 22.4 : Het gedrag van pin `PA0` uit code 13.1.** Na elke 125 interrupts (`TOV0`) klapt uitgang `PA0` om.

Het uitgangssignaal heeft in de voorafgaande voorbeelden een relatieve pulsduur van 50%. In de normale modus kunnen ook signalen met een andere relatieve pulsduur worden gemaakt door alternerend met de teller `pulse_width` klokslagen te tellen en het aantal klokslagen van een complete periode vermindert met `pulse_width` te tellen. In onderstaand voorbeeld is de pulsduur `pulse_width` gelijk aan 50. De periodetijd is 250 klokslagen en de duty-cycle 20 %. De waarde van `pulse_width` moet kleiner zijn dan 250.

```
ISR(TIMER0_OVF_vect)
{
  if ( bit_is_clear(PINA,0) ) {
    PORTA |= _BV(0);
    TCNT0 = 256-pulse_width;      // count pulse_width clockcounts
  } else {
    PORTA &= ~_BV(0);
    TCNT0 = pulse_width+6;        // count 250 - pulse_width clockcounts
  }
}
```

De ATmega32 heeft drie timers met in het totaal vier PWM-uitgangen. Als er meer PWM-signalen nodig zijn, is de normale modus geschikt. Code 22.1 geeft een interruptfunctie voor het aansturen van vier PWM-signalen met ieder een eigen duty-cycle.

Er zijn, zoals figuur 22.5 laat zien vijf toestanden. De variabele `i` bevat de toestand. Als deze 4 is, worden de uitgangen hoog gemaakt en bij de andere toestanden wordt steeds een van de uitgangen laag gemaakt. De teller telt steeds `pw[i+1]-pw[i]` gedeelde klokslagen. Bij vijf toestanden moet array `pw` zes waarden bevatten. De laatste waarde geeft de periodetijd van de PWM-signalen.




**Figuur 22.5 : De vier PWM-signalen van code 22.1 met een verschillende duty-cycle.**

**Code 22.1 : Een ISR voor het aansturen van vier PWM-signalen met timer 0.**

```
 1  volatile int pw[] = {0,50,100,150,200,250};
 2
 3  ISR(TIMER0_OVF_vect)
 4  {
 5    static int i=4;
 6
 7    TCNT0 = 256-pw[i+1]+pw[i];
 8    if (i == 4) {
 9      PORTA |= _BV(3)|_BV(2)|_BV(1)|_BV(0);
10      i = 0;
11    } else {
12      PORTA &= ~_BV(i);
13      i++;
14    }
15  }
```

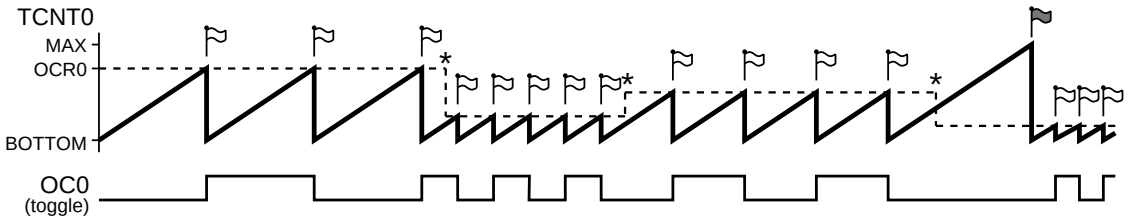
### Clear Timer Compare match mode

In CTC-modus telt de timer steeds van `BOTTOM` tot de waarde die in het `OCR0`-register staat en maakt juist gebruik van de comparator, het `OCR0`-register, uitgang `OC0` en de waveformgenerator. Deze modus vergelijkt, elke keer als de teller opgehoogd wordt, de waarde van de teller `TCNT0` met de waarde in het `OCR0`-register. Als deze gelijk zijn, is de *Output Compare Flag* (`OCF0`) hoog en wordt teller nul gemaakt. Als bovendien de betreffende interruptvlag (`OCIE0`) hoog is en de globale interrupt is actief, treedt er een zogenoemde *compare overflow*-interrupt op en wordt de bijbehorende interruptfunctie uitgevoerd. Deze functie kan bijvoorbeeld pin `0` van poort A laten omklappen:

```
ISR(COMP0_OVF_vect)
{
  PORTA = PORTA ^ _BV(0);
}
```

Deze ISR levert een PWM-signaal met een relatieve pulsduur van 50% op. De frequentie is eenvoudig in te stellen door een andere waarde in `OCR0` te zetten.

Overigens is een interruptfunctie niet nodig. Er is een betere en eenvoudigere oplossing met de `OC0`-pin en de waveformgenerator. Het uitvoeren van een interruptfunctie kost altijd een flink aantal klokslagen, omdat de huidige situatie op de stack gezet moet worden en na afloop weer teruggezet moet worden.
Gebruik daarom de `OC0`-pin als uitgang en stel de waveformgenerator in op de *toggle*-mode. De bits `COM01` en `COM00` uit register `TCCR0` zijn dan respectievelijk 1 en 0. Er ontstaat dan een signaal, zoals in figuur 22.6 is getekend. Elke keer als de `OCF0`-vlag hoog wordt, klapt de uitgang `OC0` om.




**Figuur 22.6 : De waarde van `TCNT0` en uitgang `OC0` bij de CTC-modus.** Teller `TCNT0` loopt steeds van `BOTTOM` tot `OCR0`. Als de `OCF0`-vlag hoog wordt, staat er een witte vlag. Bij een kleinere waarde van `OCR0` volgen de `OCF0`-vlaggen elkaar sneller en is de frequentie van `OC0`-signaal hoger.
De verandering van `OCR0` is aangegeven met een ∗. Als `OCR0` een waarde krijgt die lager is dan de huidige `TCNT0`, loopt de teller door tot de maximale waarde en wordt de `TOV0`-vlag hoog.

In de formule is de waarde van `OCR0` met 1 verhoogd, omdat `OCF0` hoog wordt als `OCR0` naar `BOTTOM` gaat.

De frequentie $f_{\text{ctc}}$ van het signaal hangt af van $f_{\text{cpu}}$, de prescaling $P$ en de waarde van register `OCR0`. Er zijn twee interrupts binnen een periode. Met formule 13.2 en het gegeven dat $m$ gelijk is aan `OCR0`+1, is de frequentie $f_{\text{ctc}}$:

$$f_{\text{ctc}} = \frac{f_{\text{cpu}}}{2\,(\texttt{OCR0}+1)\,P} \tag{22.4}$$

Bij de CTC-modus heeft een verandering van `OCR0` onmiddellijk effect. In figuur 22.6 is dit aangegeven met een asterisk. Een probleem kan optreden als `OCR0` wijzigt in een lagere waarde dan de huidige waarde van `TCNT0`. De teller loopt dan door tot `MAX`.
In code 22.2 staat een minimaal programma om uitgang `OC0` in de CTC-modus aan te sturen. De bits van register `TCCR0` bevatten de instelling van timer `0`. Pin 3 van poort B is ook de `OC0`-uitgang. Om `OC0` te gebruiken moet deze pin als uitgang gedefinieerd zijn. De prescaling is 8, `OCR` is 124 en met $f_{\text{cpu}}$ 8 MHz is de frequentie van het signaal 4 kHz.

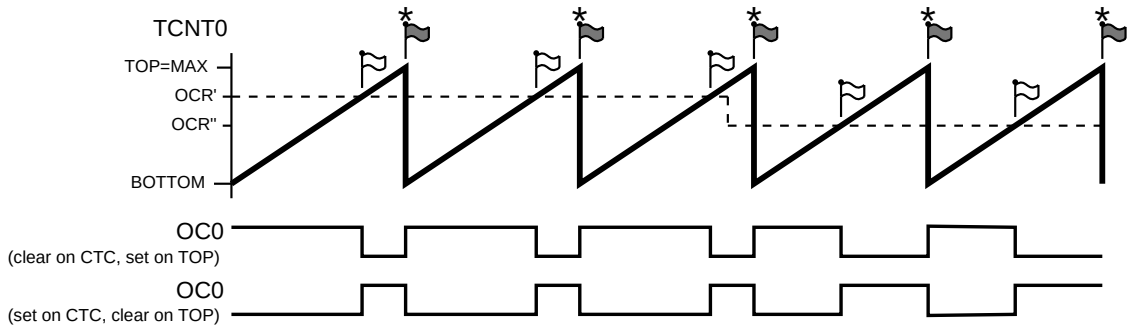
**Code 22.2 : Een minimale configuratie voor de CTC-mode van timer `0`. ($f_{\text{cpu}} = 8$ MHz)**

```
1  #include <avr/io.h>
2
3  int main (void)
4  {
5    TCCR0 =  _BV(WGM01)| !_BV(WGM00) |             // CTC-mode
6            !_BV(COM01)|  _BV(COM00) |             // toggle output
7            !_BV(CS02) |  _BV(CS01)  | !_BV(CS00); // prescaling Fcpu/8
8    OCR0  = 124;
9    DDRB  = _BV(3);                                // Pin PB3/OC0 is output
10
11   while(1) asm volatile ("nop");
12 }
```

### Fast PWM

Bij timer 1 is `TOP` afhankelijk van de `WGM`-bits en kan gelijk zijn aan `0x00FF`, `0x01FF`, `0x03FF`, `OCR1A` of `ICR1`.

In de fast-PWM-modus telt de timer steeds van `BOTTOM` tot de waarde `TOP`. Dat is bij timer `0` en timer 2 het maximale bereik `MAX` van de teller. Bij `TOP` wordt de `TOV0` hoog en is de teller weer nul. Als bovendien `TOIE0` hoog is en de globale interrupt actief is, is er *timer overflow*-interrupt.
Als tijdens het tellen de waarde in `TCNT0` gelijk is aan `OCR0` is `OCF0` hoog. Er treedt een *compare overflow* interrupt op als bovendien de globale interrupt aan staat en `OCIE0` hoog is.




**Figuur 22.7 :** **De waarde van `TCNT0` en uitgang `OC0` bij fast-PWM.** Teller `TCNT0` loopt steeds van `BOTTOM` tot `MAX`. Als de `OCF0`-vlag hoog is, staat er een witte vlag. Als de `TOV0`-vlag hoog is, staat er een donkere vlag. Een wijziging van `OCR0` wordt pas doorgevoerd als de teller gelijk is aan `TOP`. Dit is aangegeven met een $*$. De uitgang `OC0` kan laag gemaakt bij `OCF0` en hoog bij `TOP` of omgekeerd.

Figuur 22.7 toont het gedrag van de uitgang `OC0` voor twee situaties. In de ene situatie, de niet-inverterende uitgangsmodus, wordt de uitgang laag gemaakt bij de `OCF0`-vlag en hoog als `TOP` bereikt is. In de andere situatie, de inverterende uitgangsmodus, is dat precies andersom. In het voorbeeld heeft het signaal met de niet-inverterende uitgangsmodus aanvankelijk een duty-cycle van 80% en het signaal met de inverterende uitgangsmodus een duty-cycle van 20%. Na verloop van het tijd verandert `OCR0` en worden de duty-cycles voor beide signalen 50%.

Bij de fast-PWM modus kan de pulsbreedte eenvoudig worden aangepast. Een andere waarde van `OCR0` geeft een andere pulsbreedte. De duty-cycle bij de niet-inverterende uitgangsmodus is gelijk aan:

$$\Delta_{\text{fast}} = \frac{\texttt{OCR0} + 1}{\texttt{TOP} + 1} \tag{22.5}$$

De frequentie $f_{\text{fast}}$ van het signaal hangt af van de frequentie van de systeemklok, de prescaling $P$ en de waarde van `TOP`. Voor timer `0` en timer 2 is `TOP` gelijk aan 255.

$$f_{\text{fast}} = \frac{f_{\text{cpu}}}{(\texttt{TOP} + 1)\,P} = \frac{f_{\text{cpu}}}{256\,P} \tag{22.6}$$

Voor timer `0` en timer 2 zijn bij een bepaalde systeemklok maar een paar frequenties mogelijk. Bij timer 1 is `TOP` instelbaar en kan iedere frequentie gemaakt worden.

In code 22.3 staat een minimaal programma om uitgang `OC0` in de fast-PWM aan te sturen. De bits van register `TCCR0` bevatten de instelling van timer `0`. Pin 3 van poort B is ook de `OC0`-uitgang. Om `OC0` te gebruiken moet deze pin als uitgang gedefinieerd zijn. De prescaling is 8, `OCR0` is 127 en met $f_{\text{cpu}}$ 8 MHz is de frequentie van het signaal ongeveer 3,9 kHz. Bij een `OCR0` van 127 is de relatieve pulsduur 50%.

Bij timer 1 is `TOP` afhankelijk van de `WGM`-bits en kan gelijk zijn aan `0x00FF`, `0x01FF`, `0x03FF`, `OCR1A` of `ICR1`.

### Phase correct PWM

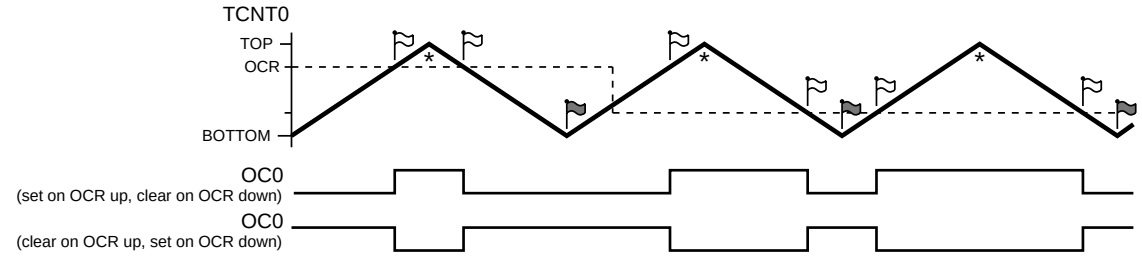
In de phase-correct-PWM modus telt de timer steeds van `BOTTOM` tot de waarde `TOP` en daarna terug tot `BOTTOM`. Bij timer `0` en timer 2 is `TOP` het maximale bereik `MAX` van de teller. Bij `BOTTOM` wordt de `TOV0` hoog en wordt de teller weer nul gemaakt.

**Code 22.3 : Een minimale configuratie bij timer 0 voor de fast-PWM met duty-cycle 50%. ($f_{cpu} = 8$ MHz)**

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

int main (void)
{
  DDRB  = _BV(3);                                  // Pin PB3/OC0 is output
  OCR0  = 127;                                     // duty cycle (127+1)/(255+1) = 50%
  TCCR0 =  _BV(WGM01)|  _BV(WGM00) |               // fast PWM
           _BV(COM01)| !_BV(COM00) |               // clear on OC/set at TOP
          !_BV(CS02) |  _BV(CS01)  | !_BV(CS00); // prescaling Fcpu/8

  while(1) asm volatile ("nop");
}
```

Als bovendien TOIE0 hoog is en de globale interrupt actief is, is er *timer overflow*-interrupt. De vlag OCF0 is hoog als tijdens het tellen de waarde in TCNT0 gelijk is aan OCR0. Er treedt een *compare overflow*-interrupt op als bovendien de globale interrupt actief is en OCIE0 hoog is.

Bij de ATmega32 negeert AVRstudio bij timer 1 en timer 2 het hoog worden van TOV0 en TOV2




**Figuur 22.8 : De waarde van TCNT0 en uitgang OC0 bij phase-correct-PWM.** Teller TCNT0 loopt steeds van BOTTOM tot TOP en weer terug naar BOTTOM. Als de OCF0-vlag hoog is, staat er een witte vlag. Als de TOV0-vlag hoog is, staat er een donkere vlag.
Een wijziging van OCR0 wordt pas doorgevoerd als de teller gelijk is aan TOP. Dit is aangegeven met een $*$. De uitgang OC0 kan laag gemaakt worden bij een *match* (OCF0) tijdens het optellen en hoog gemaakt worden tijdens het aftellen of omgekeerd.

Bij de phase-correct-PWM modus kan — net als bij de fast-PWM modus — de pulsbreedte eenvoudig worden aangepast. Een andere waarde van OCR0 geeft een andere pulsbreedte. De duty-cycle $\Delta_{\text{phase}}$ bij de niet-inverterende uitgangsmodus is gelijk aan:

$$\Delta_{\text{phase}} = \frac{\texttt{OCR0}}{\texttt{TOP}} \tag{22.7}$$

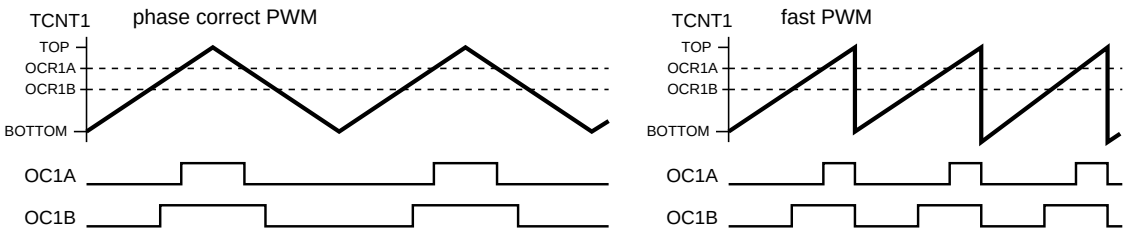
en bij de geïnverteerde uitgangsmodus is dat:

$$\Delta_{\text{phase}} = \frac{255 - \texttt{OCR0}}{\texttt{TOP}} \tag{22.8}$$

De frequentie $f_{\text{phase}}$ van het signaal hangt af van de frequentie van de systeemklok, de prescaling $P$ en de waarde van TOP. Voor timer 0 en timer 2 is TOP gelijk aan 255.

$$f_{\text{phase}} = \frac{f_{\text{cpu}}}{2(\texttt{TOP})\,P} = \frac{f_{\text{cpu}}}{510\,P} \tag{22.9}$$

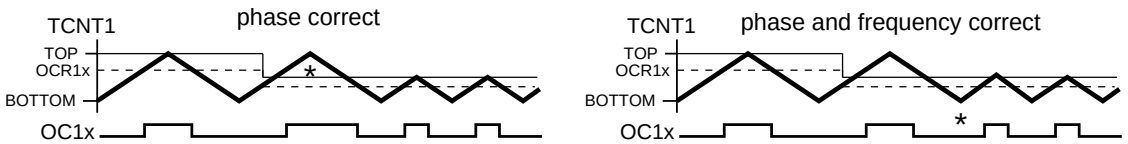
Voor timer 0 en timer 2 zijn bij een bepaalde systeemklok maar een beperkt aantal frequenties mogelijk. Bij timer 1 is TOP instelbaar en kan iedere gewenste frequentie gemaakt worden. Een ander voordeel van timer 1 is dat deze twee comparatoruitgangen heeft en er twee PWM-signalen gemaakt kunnen worden.




**Figuur 22.9 :** **Het gedrag van phase-correct-PWM en fast-PWM.** De uitgangen OC1A en OC1B zijn ingesteld op de inverterende uitgangsmodus. De frequentie van fast-PWM is twee zo groot als die van phase-correct-PWM. Bij fast-PWM worden de uitgangen OC1A en OC1B gelijktijdig laag en bij phase-correct-PWM veranderen OC1A en OC1B nooit allebei tegelijk.

Het verschil tussen fast-PWM en phase-correct-PWM is het meest duidelijk bij timer 1. In figuur 22.9 staat het gedrag van de uitgangen OC1A en OC1B voor deze beide typen. In alle gevallen is de inverterende uitgangsmodus gebruikt. Er zijn twee belangrijke verschillen. Ten eerste — dit geldt ook voor timer 0 en timer 2 — is de frequentie van de signalen in fast-PWM twee keer zo groot. Ten tweede worden in fast-PWM de signalen gelijktijdig laag en veranderen bij phase-correct-PWM de PWM-signalen nooit tegelijkertijd. Bij de niet-inverterende uitgangsmodus worden bij fast-PWM de signalen gelijktijdig hoog.

Als de signalen twee motoren aansturen betekent dit dat in de fast-PWM modus beide motoren tijdelijk een grote stroom trekken. In phase-correct modus is de stroom die motoren trekken beter verdeeld. Bij motoren en andere actuatoren, die een grote stroom gebruiken, heeft de phase-correct modus de voorkeur. Voor timer 0 en timer 2 is dit minder relevant. Beide hebben slechts een uitgang. Als de timer 0 een motor aanstuurt en timer 2 een andere motor, worden er twee tellers gebruikt. Als de tellers niet op hetzelfde tijdstip gestart zijn, is de kans klein dat de uitgang OC0 en OC2 gelijktijdig veranderen.




**Figuur 22.10 :** **Het verschil tussen phase-correct-PWM en phase-and-frequency-correct-PWM.** Het veranderen van TOP, OCR1A/OCR1B is aangegeven met een ∗. De duty-cycle blijft 33%, omdat TOP, OCR1A/OCR1B met dezelfde verhouding veranderen. De verandering gebeurt bij phase-correct-PWM bij TOP en bij phase-and-frequency-correct-PWM bij BOTTOM. Bij phase-correct-PWM is de duty-cycle gedurende één periode geen 33%.

Timer 1 kent naast de phase-correct-modus ook de phase-and-frequency-correct-modus. Het verschil tussen deze twee is dat OCR1A, OCR1B en TOP bij de eerste de nieuwe waarde bij TOP krijgen en bij de tweede bij BOTTOM. Dit verschil geeft alleen een ongewenst effect bij phase-correct-PWM als de waarde van TOP verandert. In figuur 22.10 veranderen TOP, OCR1A en OCR1B met dezelfde verhouding. Het PWM-signaal houdt een duty-cycle van 33%. Bij de phase-correct-modus is de duty-cycle gedurende één periode geen 33%.

**Code 22.4 : Minimale configuratie timer 0 voor de phase-correct-PWM met duty-cycle van bijna 50%. ($f_{cpu} = 8$ MHz)**

```
 1  #include <avr/io.h>
 2
 3  int main (void)
 4  {
 5    DDRB  = _BV(3);                                          // Pin PB3/OC0 is output
 6    OCR0  = 127;                                             // duty cycle 50%
 7    TCCR0 = !_BV(WGM01)|  _BV(WGM00) |                       // phase correct PWM
 8             _BV(COM01)| !_BV(COM00) |                       // clear on OC/set at TOP
 9            !_BV(FOC0) | !_BV(CS02)  | _BV(CS01)  | !_BV(CS00); // prescaling Fcpu/8
10
11    while(1) asm volatile ("nop");
12  }
```

In code 22.4 staat een minimaal programma om voor uitgang OC0 een signaal met phase-correct-PWM-modus te maken. De bits van register TCCR0 bevatten de instelling van timer 0. Om OC0 te gebruiken moet Pin 3 van poort B als uitgang gedefinieerd zijn. De prescaling is 8, OCR0 is 127 en met $f_{cpu} = 8$ MHz is de frequentie van het signaal ongeveer 1,96 kHz. Als OCR0 de waarde 127 heeft, is de relatieve pulsduur 49,8%.

## 22.3 Fast-PWM: een regeling voor intensiteit led

Een typisch voorbeeld voor een toepassing van fast-PWM is het genereren van een PWM-signaal waarbij het gemiddelde gelijkspanningsniveau geregeld wordt door de pulsbreedte te variëren. Als dit signaal een led aanstuurt, regelt dit de intensiteit waarmee de led brandt. Een probleem bij leds is dat de stroom door de led niet lineair is met de spanning en dat de hoeveelheid uitgezonden licht ook niet lineair is met de stroom. Bovendien is het oog ook niet lineair. Het oog kan heel weinig en heel veel licht waarnemen. Het dynamisch bereik van het oog is extreem groot. Het oog ervaart een vertienvoudiging van de lichtsterkte niet als tien keer zo fel.

Veel voorbeelden uit de literatuur en op het internet, die met behulp van PWM leds aansturen, houden geen rekening met de niet-lineariteit tussen het gemiddelde gelijkspanningsniveau en de lichtsterkte die het oog ervaart. In code 22.5 is enigszins rekening gehouden met de niet-lineariteit. De intensiteit wordt in elf niveaus verdeeld. De breedte van de puls van het PWM-signaal wordt bepaalt met OCR0. De elf waarden voor OCR0 staan in een array level. Het oog ziet verschillen in de felheid beter als de led zwak brandt. Daarom liggen in array level de kleine waarden dicht bij elkaar. Array level bevat elf waarden uit de reeks van Fibonacci. Aanvankelijk wordt, op regel 10, OCR0 ingesteld op 21, de middelste waarde uit de reeks.

**Code 22.5 : Regeling voor lichtintensiteit van led op basis van fast-PWM. ($f_{cpu} = 8$ MHz)**

```
 1  #include <avr/io.h>
 2
 3  volatile int i=5;
 4  volatile uint8_t level[11] = {2,3,5,8,13,21,34,55,89,144,233};
 5
 6  int main(void) {
 7    DDRA  = !_BV(0)|!_BV(1);                                  // PA0 en PA1 input
 8    PORTA =  _BV(0)| _BV(1);                                  // enable pullups
 9    DDRB  =  _BV(3);                                          // PB3/OC0 output
10    OCR0  = level[i];
11    TCCR0 =  _BV(WGM01)|  _BV(WGM00)|                         // fast PWM
12             _BV(COM01)| !_BV(COM00)|                         // clear OCR/set TOP
13            !_BV(FOC0) | !_BV(CS02) | _BV(CS01) | _BV(CS00);  // prescaling clk/64
14
15    while (1) {
16      if(bit_is_clear(PINA, 0)){          // increase duty cycle
17        if (i < 10) i++;
18        OCR0 = level[i];
19        loop_until_bit_is_set(PINA, 0);
20      }
21      if(bit_is_clear(PINA, 1)) {         // decrease duty cycle
22        if (i > 0) i--;
23        OCR0 = level[i];
24        loop_until_bit_is_set(PINA, 1);
25      }
26    }
27  }
```

De anode van de led is verbonden met de uitgang van de microcontroller. Dat betekent dat de microcontroller de stroom levert. De uitgangsmodus is niet-geïnverteerd. Deze keuzes leiden er toe dat een hogere waarde van `OCR` een grotere duty-cycle geeft en dat de led feller zal branden.

Timer `0` is ingesteld op fast-PWM, de prescaling is 64 en de uitgangsmodus is niet-geïnverteerd. Na regel 13 gaat de teller lopen en staat er een PWM-signaal op pin 3 van poort B.
In de oneindige lus wordt getest of een van de ingangen `0` en 1 van poort A laag is. Als ingang `0` laag is, wordt de index `i` opgehoogd, krijgt `OCR0` een hogere waarde en zal de led feller branden. Als ingang 1 laag is, wordt de index `i` verlaagd, krijgt `OCR0` een lagere waarde en zal de led minder fel gaan branden.

De frequentie van het PWM-signaal is bij een systeemklok van 8 MHz ruim 488 Hz. Voor de aansturing van een led is de frequentie niet belangrijk, maar als er per se een frequentie van exact 500 Hz nodig is, kan dit met de truc uit paragraaf 13.4 worden opgelost door `TCNT0` vanaf 6 te laten tellen. Er is dan een interruptfunctie nodig, die `TCNT0` ophoogt:

```
ISR(TIMER0_OVF_vect)
{
  TCNT0 += 6;
}
```

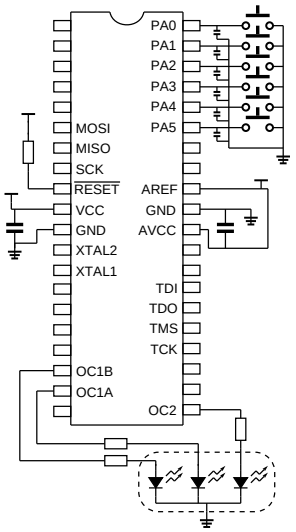
Bovendien moeten de waarden van de array `level` worden aangepast en moet de interrupt worden geactiveerd door dit aan de initialisatie toe te voegen:

```
  TIMSK |= _BV(TOIE0);
  sei();
```

Als een exacte frequentie nodig is, is het beter om de CTC-mode te gebruiken of om voor timer 1 te kiezen.

**Code 22.6 : Het headerbestand `fade.h`.**

```
1  #include <stdint.h>
2
3  #define UP    1
4  #define DOWN  0
5
6  extern volatile uint8_t level[11];
7
8  void fade(int dir, volatile int *p_index, volatile uint8_t *p_pin,
9                               uint8_t bit, volatile uint8_t *p_ocr);
```




**Figuur 22.11 : Schema voor aansturing rgb-led.**
Er is een rgb-led met een gemeenschappelijke kathode gebruikt. Het nadeel is dat de microcontroller de bron is. Het voordeel is dat de code duidelijker is. Signalen met een grotere duty-cycle laten de leds feller branden.

Een rgb-led bestaat uit een rode, een groene en een blauwe led in een behuizing. Met een rgb-led kunnen alle kleuren worden gemaakt door de stroom door de drie leds te variëren. Rgb-leds zijn verkrijgbaar met een gemeenschappelijke anode en een gemeenschappelijke kathode. In dit geval, zie figuur 22.3, is er een gemeenschappelijke kathode. Timer 1 heeft twee `OC`-uitgangen. De drie PWM-signalen worden met timer `0` en timer 1 gemaakt. Om code 22.5 geschikt te maken voor een rgb-led moet de initialisatie van timer 1 worden toegevoegd. Voor de regeling van de intensiteit van de leds zijn zes drukknoppen nodig. Dit kan door de code van regel 16 tot en met regel 20 zes keer op te nemen en aan te passen voor de verschillende `OC`-uitgangen. In plaats daarvan wordt de functie `fade` uit het bestand `fade.c` gebruikt. Code 22.7 toont deze functie en stelt de intensiteit in voor een dimrichting `dir`, een niveau-index `p_index`, een `PIN`-register `p_pin`, een bitnummer `bit` en een `OCR`-register `p_ocr`. Het headerbestand `fade.h` uit code 22.6 geeft het prototype van deze functie en de definities van de dimrichtingen `UP` en `DOWN`.

**Code 22.7 : Het bestand `fade.c` met de functie `fade`.**

```
1  #include <avr/io.h>
2  #include "fade.h"
3
4  volatile uint8_t level[11] = {2,3,5,8,13,21,34,55,89,144,233};
5
6  void fade(int dir, volatile int *p_index, volatile uint8_t *p_pin,
7                               uint8_t bit, volatile uint8_t *p_ocr)
8  {
9    if (bit_is_clear(*p_pin, bit)) {
10     if ( dir==UP)  {
11       if (*p_index < 10) (*p_index)++;
12     } else {
13       if (*p_index > 0) (*p_index)--;
14     }
15     *p_ocr = level[*p_index];
16     loop_until_bit_is_set(*p_pin, bit);
17   }
18 }
```

De variabele `p_pin` is een pointer naar het PIN-register en `bit` is het bitnummer van de aansluiting waarop de drukknop is aangesloten. Op regel 9 wordt met de functie `bit_is_clear` gekeken of het signaal op de ingang laag is. Als dat zo is, stelt de functie een nieuw niveau in. In het hoofdprogramma is voor elke kleur een

**Code 22.8 : Het hoofdprogramma voor de aansturing van een rgb-led. ($f_{cpu} = 8$ MHz)**

```
1  #include <avr/io.h>
2  #include "fade.h"
3
4  volatile int r=5, g=5, b=5;
5
6  int main(void) {
7    DDRA   = !_BV(0)|!_BV(1);  // PA0 en PA1 input
8    PORTA  =  _BV(0)| _BV(1);  // enable pullup PA0 en PA1
9    DDRB   =  _BV(3);          // PB3/OC0 output
10   DDRD   =  _BV(4)|_BV(5);   // PB4/OC1B and PB5/OC1A output
11
12   OCR0   = level[r];
13   TCNT0  = 0;
14   TCCR0  =  _BV(WGM01) |  _BV(WGM00) |                          // fast PWM
15             _BV(COM01) | !_BV(COM00) |                          // clear OCR/set TOP
16            !_BV(CS02)  |  _BV(CS01)  | _BV(CS00);               // prescaling clk/64
17   OCR1B  = level[g];
18   OCR1A  = level[b];
19   TCNT1  = 0;
20   TCCR1A =  _BV(COM1A1)| !_BV(COM1A0)| _BV(COM1B1)| !_BV(COM1B0)|  // clear OCR/set TOP
21            !_BV(WGM11) |  _BV(WGM10);                          // 8-bit fast-PWM
22   TCCR1B = !_BV(WGM13) |  _BV(WGM12) |                          // WGM=0101
23            !_BV(CS12)  |  _BV(CS11)  |  _BV(CS10);             // prescaling clk/64
24
25   while (1) {
26     fade(UP,   &r, &PINA, 0, &OCR0);
27     fade(DOWN, &r, &PINA, 1, &OCR0);
28     fade(UP,   &g, &PINA, 2, &OCR1BL);
29     fade(DOWN, &g, &PINA, 3, &OCR1BL);
30     fade(UP,   &b, &PINA, 4, &OCR1AL);
31     fade(DOWN, &b, &PINA, 5, &OCR1AL);
32   }
33 }
```

niveau-index gedeclareerd. De pointervariabele `p_index` wijst naar een van deze indices en wordt opgehoogd als de richting `UP` is en anders wordt deze verlaagd. De bijbehorende waarde uit de array `level` wordt op regel 15 aan het `OCR`-register toegekend waar de pointervariabele `p_ocr` naar wijst. Tenslotte wordt er gewacht totdat het signaal op de ingang weer hoog is.

In code 22.8 staat het hoofdprogramma. Er zijn drie indices voor de niveaus: `r` voor de rode, `g` voor de groene en `b` voor de blauwe led. Er zijn drie `OC`-uitgangen: `OC0`, `OC1B` en `OC1A`. De instelling van timer `0` is identiek met die uit code 22.5. Timer 1 is ingesteld op de 8-bit fast-PWM modus. De waarde van `TOP` is bij timer 1 gelijk aan `0xFF`. Timer 1 wordt als een 8-bits teller gebruikt. De prescaling en de uitgangsmodus zijn hetzelfde als bij timer `0`.

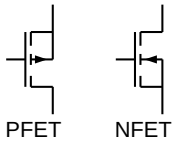
De hoofdlus roept zes keer de functie `fade` aan met steeds een andere richting en bitnummer en een adres voor de index, het `PIN`-register en `OCR`-register. Van de `OCR1`-registers wordt alleen het lage byte gebruikt. Deze bytes staan in `OCR1AL` en `OCR1BL`.

## 22.4 Phase-correct-PWM: een robotwagen met DC-motoren

Hét voorbeeld voor het gebruik van PWM-signalen is de aansturing van motoren. Er bestaan vele soorten motoren, onder andere: wisselstroom- en gelijkstroommotoren, motoren met en zonder koolborstels, stappenmotoren en servomotoren. Motoren gebruiken relatief veel stroom. De microcontroller stuurt een motor nooit rechtstreeks aan. Een gelijkstroommotor of DC-motor kan zowel vooruit als achteruit draaien door de polariteit van de spanningsbron te verwisselen en daarmee de richting van de stroom om te keren.

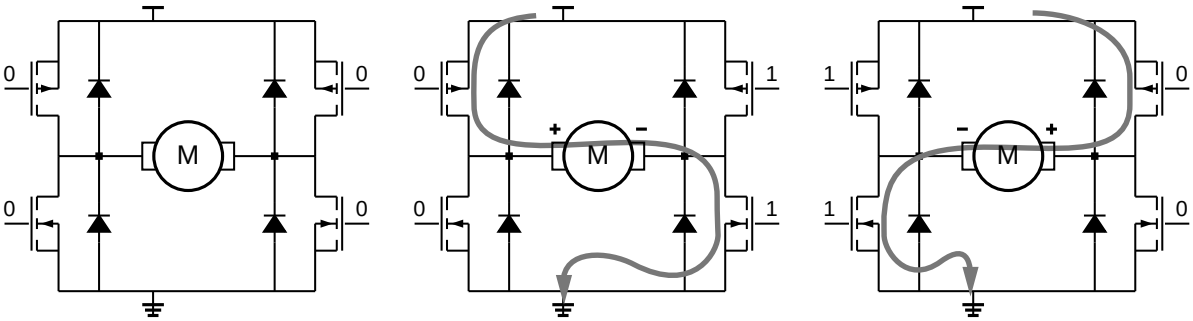
Een FET is een *Field Effect Transistor*. Dit kan een MOSFET of een JFET, *Junction* FET, zijn. Een NFET is een N-*channel* FET en een PFET is een P-*channel* FET.

Een NFET geleidt als de gate van de FET hoog is. Een PFET geleidt als de gate laag is.




**Figuur 22.12 : De symbolen van de NFET en de PFET.**

Voor de aansturing van een gelijkstroommotor, die beide kanten op moet kunnen draaien, wordt meestal een H-brug gebruikt. Figuur 22.13 geeft het principe van een zogenoemde *full h-bridge*, uit twee PFET's en twee NFET's bestaat. Als de gate van een NFET hoog en de schuin tegenoverliggende PFET laag is, loopt er een stroom via de motor van VCC naar massa. Als de gate van de andere NFET hoog en de andere PFET laag is, loopt de stroom in de andere richting en is de draairichting van de motor tegengesteld. Als alle gates laag zijn, geleiden alleen de PFET's. De spanning over de motor is dan nul en de motor staat stil.




**Figuur 22.13 : Drie verschillende situaties voor de H-brug bij de aansturing van een motor.** Bij de linker H-brug zijn de gates van de FET laag en staat de motor stil. Bij de middelste en de rechter H-brug geleiden twee schuin tegenover elkaar liggende FET's. Er loopt dan een stroom door de motor. Afhankelijk van de richting zal de motor links of rechtsom draaien.

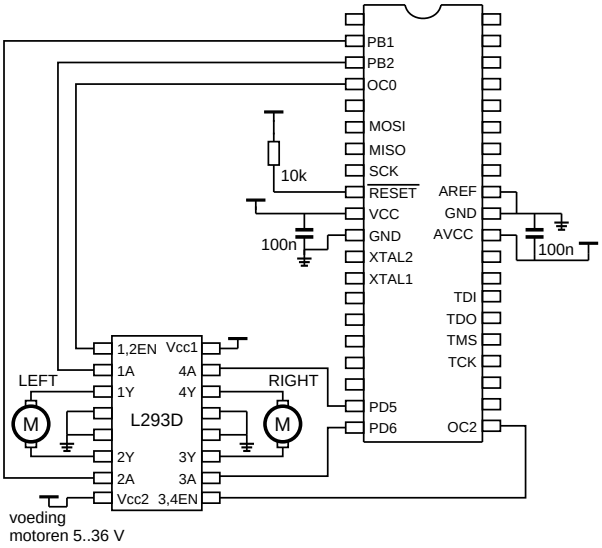
Zolang er een stroom door de motor en de H-brug loopt, spelen de vier diodes geen enkele rol. De motor bouwt door het draaien intern zelf een elektromagnetisch veld op. Als de stroom wordt uitgeschakeld, of als de stroomrichting verandert, geleiden de FET's niet meer en werkt de motor tijdelijk als dynamo. Deze stroom wordt via de diodes afgevoerd. Vanwege hun snelle schakelgedrag worden hier meestal schottkydiodes voor gebruikt.

Als de linker P- en NFET of de rechter P- en NFET allebei aan staan, kan de stroom door de FET's te groot zijn. De ontwerper moet deze situatie vermijden.

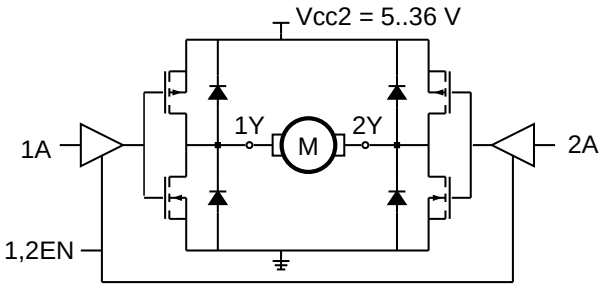
H-bruggen zijn ook als complete geïntegreerde schakeling verkrijgbaar. In figuur 22.14 staat een implementatie met een ATmega32, twee DC-motoren en een L293D. Deze laatste component bevat vier halve H-bruggen die als twee volledige H-bruggen gebruikt kunnen worden. Aan de H-bruggen is een enable-ingang toegevoegd. Het schema van deze volledige H-brug staat in figuur 22.15 met de betreffende aansluitpunten van de L293D.

De L293D kan 600 mA leveren. Voor de aansturing van een motor is dat vaak te weinig. Een alternatief is de LMD18200 die 3 A kan leveren. Voor nog zwaardere motoren is bijvoorbeeld de VNH3SP30 of de BTN7971B een alternatief.

Ingang 1,2EN is de enable-ingang van de halve H-bruggen 1 en 2. De motor wordt geplaatst tussen de uitgangen 1Y en 2Y. Als de ingangen 1A en 2A van de halve bruggen een tegengestelde waarde hebben, draaien de motoren links- of rechtsom. De snelheid kan geregeld worden door de enable-ingang met een PWM-signaal aan te sturen. Als de duty-cycle hoog is, is snelheid hoog en als deze laag is, is de snelheid ook laag.




**Figuur 22.14 : Het schema voor de aansturing van twee DC-motoren.**




**Figuur 22.15 : Een H-brug met extra logica voor de aansturing.** Er is een enable-ingang die de ingangen 1A en 1B loskoppelt van de feitelijke H-brug. De namen van de in- en uitgangen komen overeen met die van de L293D. De L203D is intern anders opgebouwd dan deze figuur suggereert.

Figuur 22.14 heeft twee motoren. Dit kunnen de motoren van een robotwagen zijn. De uitgang `OC0` van timer `0` is aangesloten op de enable-ingang van de linker motor. De draairichting wordt bepaald met de uitgangen `PB1` en `PB2`. De uitgang `OC2` van timer 2 is aangesloten op de enable-ingang van de rechter motor. De draairichting wordt bepaald met de uitgangen `PD5` en `PD6`.

**Code 22.9 : De functie `motor_on` om een motor te laten draaien.**

```
 1  void motor_on(uint8_t m, uint8_t dir, uint8_t duty)
 2  {
 3    uint8_t ocr;
 4
 5    ocr = 255-(duty*255)/100;
 6    if ( m == LEFT ) {
 7      if (dir == CLOCKWISE) {
 8        PORTB |=  _BV(2);
 9        PORTB &= ~_BV(1);
10      } else {
11        PORTB &= ~_BV(2);
12        PORTB |=  _BV(1);
13      }
14      OCR0   = ocr;
15      TCCR0 |= _BV(COM01)|_BV(COM00);
16    }
17    if ( m == RIGHT ) {
18      if (dir  == CLOCKWISE) {
19        PORTD |=  _BV(6);
20        PORTD &= ~_BV(5);
21      } else {
22        PORTD &= ~_BV(6);
23        PORTD |=  _BV(5);
24      }
25      OCR2   = ocr;
26      TCCR2 |= _BV(COM21)|_BV(COM20);
27    }
28  }
```

In code 22.9 staat een opzet voor de functie `motor_on` die een motor `m` in een bepaalde draairichting `dir` en met een snelheid `duty` laat draaien. De snelheid is een getal van `0` tot `100` en komt overeen met de duty-cycle. De functie bepaalt op regel 5 met formule 22.8 de waarde `ocr` die in het OCR-register moet komen te staan. Als `m` gelijk is aan `LEFT` wordt de linker motor aangestuurd en wordt `ocr` op

regel 14 toegekend aan OCR0. Als de draairichting CLOCKWISE is, is PB2 hoog en PB1 laag en anders is PB2 laag en PB1 hoog. Op regel 15 wordt OC0 aangesloten en ingesteld op de geïnverteerde uitgangsmodus. Als m gelijk is aan RIGHT wordt hetzelfde gedaan, maar dan voor timer 2 en de uitgangen OC2, PD5 en PD6.

De functie init_motor uit code 22.10 initialiseert de timers. Beide timers zijn ingesteld op phase-correct-PWM, hebben een prescaling van 64 en de OC-uitgang is aanvankelijk niet aangesloten. De functie motor_off uit code 22.11 stopt een motor. De OC-uitgang van motor m wordt losgekoppeld en de betreffende uitgangen van de microcontroller laag gemaakt.

**Code 22.10 : De initialisatiefunctie `init_motor`.**

```
1  void init_motor(void)
2  {
3    PORTB |= ~(_BV(1)|_BV(2)|_BV(3));              // outputs low
4    DDRB  &=  _BV(3);                              // PB3/OC0 output
5    DDRB  &=  _BV(1)|_BV(2);                       // PB1 and PB2 output
6    TCCR0  = !_BV(WGM01)|  _BV(WGM00)|             // phase correct PWM
7             !_BV(COM01)| !_BV(COM00)|             // OC0 disconnected
8             !_BV(CS02) |  _BV(CS01) |  _BV(CS00); // prescaling fcpu/64
9    PORTD |= ~(_BV(5)|_BV(6)|_BV(7));              // outputs low
10   DDRD  &=  _BV(7);                              // PD7/OC2 output
11   DDRD  &=  _BV(5)|_BV(6);                       // PD5 and PD6 output
12   TCCR2  = !_BV(WGM21)|  _BV(WGM20)|             // phase correct PWM
13            !_BV(COM21)| !_BV(COM20)|             // OC2 disconnected
14             _BV(CS22) | !_BV(CS21) | !_BV(CS20); // prescaling fcpu/64
15 }
```

De functie motor_off had ook de prescaling nul kunnen maken. Het voordeel is dat de teller stopt met tellen. Voor timer 0 betekent dit dat regel 4 uit code 22.11 vervangen kan worden door:

```
TCCR0 = 0;
```

In functie motor_on moet dan de toewijzing TCCR0 hetzelfde zijn als die van regel 6 tot en met 8 uit code 22.10.

**Code 22.11 : De functie `motor_off` om een motor te stoppen.**

```
1  void motor_off(uint8_t m)
2  {
3    if ( m == LEFT ) {
4      TCCR0 &= ~(_BV(COM01)|_BV(COM00));          // OC0 disconnected
5      PORTB &= ~(_BV(1)|_BV(2)|_BV(3));           // outputs low
6    }
7    if ( m == RIGHT ) {
8      TCCR2 &= ~(_BV(COM21)|_BV(COM20));          // OC2 disconnected
9      PORTD &= ~(_BV(5)|_BV(6)|_BV(7));           // outputs low
10   }
11 }
```

Met de functie motor_on en motor_off kunnen een aantal hulpfuncties worden gemaakt om de robotwagen te besturen. De functies car_forward en car_backward uit code 22.12 laten de robotwagen met een snelheid speed respectievelijk recht vooruit en recht achteruit rijden. De snelheid is een getal van 0 tot en met 100. De functie car_stop stopt de wagen. De functie car_left laat de wagen om zijn eigen as naar links draaien en car_left_curve laat de wagen een flauwe bocht naar links maken.

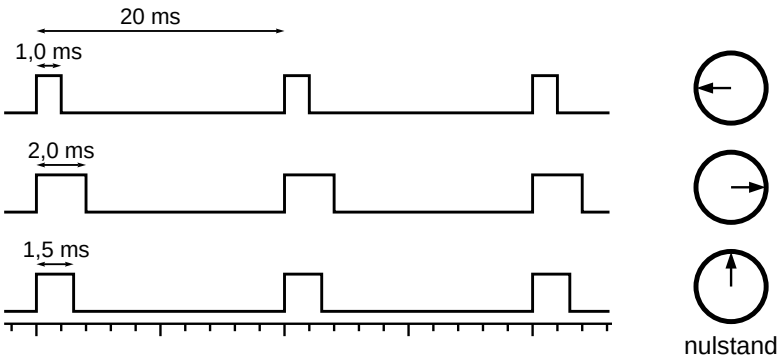
**Code 22.12 : Een aantal functies om een robotwagen te besturen.**

```
1  void car_forward(unsigned char speed)
2  {
3    motor_on(LEFT, CLOCKWISE, speed);
4    motor_on(RIGHT, CLOCKWISE, speed);
5  }
6
7  void car_backward(uint8_t speed)
8  {
9    motor_on(LEFT, COUNTERCLOCKWISE, speed);
10   motor_on(RIGHT, COUNTERCLOCKWISE, speed);
11 }
12
13 void car_stop()
14 {
15   motor_off(LEFT);
16   motor_off(RIGHT);
17 }
18
19 void car_left(uint8_t speed)
20 {
21   motor_on(LEFT, COUNTERCLOCKWISE, speed);
22   motor_on(RIGHT, CLOCKWISE, speed);
23 }
24
25 void car_left_curve(uint8_t speed)
26 {
27   motor_on(LEFT, CLOCKWISE, speed>>1);
28   motor_on(RIGHT, CLOCKWISE, speed);
29 }
```

Het nadeel van een gewone DC-motor is dat niet bekend is hoe snel en hoe scherp een bocht wordt gemaakt. Er is wel een verband tussen de gemiddelde spanning en het aantal toeren per minuut. Met de omtrek van het wiel en de eventuele mechanische overbrenging kan heel ruw de afgelegde weg berekend worden. Een groot probleem is dat door wrijving dit niet klopt. Bij het maken van een bocht naar links, ondervindt het linker wiel veel meer wrijving dan het rechter wiel en zal de bocht scherper worden.

## 22.5 Phase-and-frequentie-correct-PWM: aansturing servomotor

Een servomotor is een AC- of DC-motor met een terugkoppelmechanisme. Op de as van de motor zit een encoder die informatie over de positie en de snelheid terugkoppelt. Er bestaan heel veel verschillende uitvoeringen. Een veel voorkomende vorm is een servomotor die aangestuurd wordt met een PWM-signaal waarvan de periode 20 ms is en de pulsduur tussen 1,0 tot 2,0 ms ligt. Deze servo kan bewegen over een hoek van 180$^o$ en is bedoeld voor onder andere robots. Bij een pulsduur van 1,5 ms bevindt de servo zich in de zogenoemde tussenstand. Figuur 22.16 toont de PWM-signalen voor de nulstand en de twee uitersten.




**Figuur 22.16 : De PWM-signalen voor een servomotor bij de nulstand en bij de twee uitersten.**

De servo kan worden aangestuurd met dan wel fast-, phase-correct- of phase-and-frequency-correct-PWM. Het is gebruikelijk om de servo aan te sturen met een PWM-signaal van ongeveer 50Hz of 60Hz. Als de phase-correct modus gebruikt wordt, geldt voor de frequentie formule 22.9. Bij timer 0 is TOP gelijk aan 255. Met $f_{\text{cpu}}$ 8 MHz en een prescaling 256 volgt uit formule 22.9 dat de frequentie ongeveer 61,3 Hz is.
Belangrijk is dat de nulstand goed ingesteld kan worden. Er is een pulsbreedte nodig van 1,5 ms. Met formule 22.7 voor de relatieve pulsduur en formule 22.9 voor de frequentie $f_{\text{phase}}$ geldt voor de pulsbreedte $T_{\text{pw}}$:

$$T_{\text{pw}} = \frac{\texttt{OCR}}{\texttt{TOP}\ f} = \frac{2\,\texttt{OCR}\,P}{f_{\text{cpu}}} \tag{22.10}$$

Dit geeft dat OCR 23,44 moet zijn. Afgerond is dat 23. De fout is dan 2% en dat is te veel voor de nulstand. Bovendien moet de pulsbreedte variëren van 1,0 tot 2,0 ms. De waarde van OCR ligt dan tussen 15 en 31. Het aantal posities is dan slechts 17. Daarom is timer 0 — en timer 2 — niet echt geschikt voor de aansturing van een servo.

Omdat timer 1 een 16-bits teller is en omdat de frequentie op elke waarde — dus ook op een handig gekozen waarde — instelbaar is, is deze timer zeer geschikt voor het aansturen van een servomotor. De waarde voor TOP komt in het ICR1-register te staan. De waarde in het OCR1A-register bepaalt de pulsbreedte. Met OCR 750, TOP 10000 en een prescaling 8 wordt de duty-cycle exact 1,5 ms en is de frequentie exact 50 Hz. Als de waarde van OCR tussen 500 en 1000 ligt, ligt de pulsbreedte tussen 1,0 tot 2,0 ms.

In code 22.13 varieert de pulsbreedte in stappen van 5. De maximale hoek is 180° en komt overeen met 100 stappen. Een stap is dan gelijk aan een verdraaiing van 1,8°. Een stapgrootte 1 verdeelt de hele hoek zelfs in 500 stappen van 0,36°.
Timer 1 is ingesteld op de phase-and-frequency-correct-PWM met ICR1 als TOP. De WGM1-bits zijn gelijk aan 1000, zie ook tabel G.5 uit bijlage G. De prescaling is 8 en OC1A heeft de niet-geïnverteerde modus; OC1A wordt laag bij het optellen en hoog bij het aftellen. De waarde van ICR1 is 10000 en OCR1A krijgt aanvankelijk de waarde van de nulstand, namelijk 750. In dit voorbeeld wordt OC1A gebruikt en is pin 5 van poort D de uitgang.

**Code 22.13 : De besturing van een servomotor. ($f_{cpu}$ = 8 MHz)**

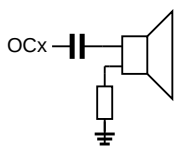
```
#include <avr/io.h>

#define STEP   5

int main(void) {
  DDRA  = !_BV(0)|!_BV(1);  // PA0 en PA1 input
  PORTA =  _BV(0)| _BV(1);  // enable pullup PA0 en PA1
  DDRD  =  _BV(5);          // PD5/OC1A output

  OCR1A  = 750;
  ICR1   = 10000;
  TCNT1  = 0;
  TCCR1A =  _BV(COM1A1)| !_BV(COM1A0)|  _BV(COM1B1)| !_BV(COM1B0)| // clr OCR1x up/set OCR1x down
           !_BV(WGM11) | !_BV(WGM10);                             // ph&freq.corr. PWM, ICR1 as TOP
  TCCR1B =  _BV(WGM13) | !_BV(WGM12) |                             // WGM=1000
           !_BV(CS12)  |  _BV(CS11)  | !_BV(CS10);                // prescaling clk/8

  while (1) {
    if(bit_is_clear(PINA, 0)){
      if ( OCR1A <= (1000 - STEP) ) OCR1A += STEP;
      loop_until_bit_is_set(PINA, 0);
    }
    if(bit_is_clear(PINA, 1)) {
      if ( OCR1A >= (500 + STEP) )  OCR1A -= STEP;
      loop_until_bit_is_set(PINA, 1);
    }
  }
}
```

Pin 0 en pin 1 van poort A zijn met twee drukknoppen verbonden. Als de knop van pin 0 ingedrukt is, wordt OCR1A met de stapgrootte STEP opgehoogd en de motor draait 1,8° met de klok mee. Als de knop van pin 1 ingedrukt is, wordt OCR1A met de stapgrootte STEP verlaagd en de motor draait 1,8° tegen de klok in.

Een toon afspelen kan met een luidspreker of een buzzer. Er zijn magnetische en piëzo-elektrische buzzers.

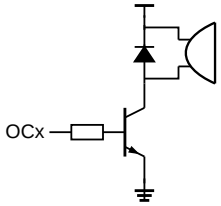
## 22.6 CTC-modus: het afspelen van muziek

Muziek bestaat uit meerdere tonen. Tonen worden na elkaar of gelijktijdig afgespeeld. Als er in een muziekstuk meerdere tonen tegelijk worden afgespeeld, is het een meerstemmige of polyfone melodie. Een muziekstuk waarin de tonen alleen na elkaar komen is eenstemmig of homofoon. Een toon heeft een bepaalde frequentie en een bepaalde tijdsduur. De tijdsduur wordt bepaald door de relatieve toonduur en het tempo van de muziek.




**Figuur 22.17 : Aansluiting met luidspreker.**
De weerstand beperkt de stroom en limiteert het volume.

Deze paragraaf bespreekt het af spelen van een eenstemmig muziekstuk met de microcontroller. De informatie over frequentie, relatieve toonduur en tempo moeten worden vastgelegd. Het is natuurlijk mogelijk om zelf een formaat te verzinnen en muziekstukken om te zetten naar dat formaat, maar het is handiger om een bestaand formaat te kiezen. Daarvoor bestaan verschillende alternatieven, zoals RTTTL en MML. MML staat voor *Music Markup Language* en wordt gebruikt in webapplicaties. RTTTL staat voor *Ring Tone Text Transfer Langua-*



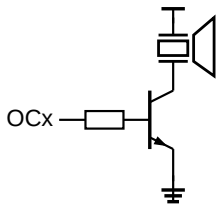

**Figuur 22.18 : Aansluiting bij een magnetische buzzer.** De diode voert de inductiestroom af.

*ge* en is in de jaren negentig door Nokia ontwikkeld om homofone beltonen op mobiele telefoons af te spelen. RTTTL is compacter en eenvoudiger dan MML. Bovendien zijn er op het internet honderden RTTTL-beltonen te downloaden. De specificatie van RTTTL staat in paragraaf E.1 van bijlage E.

Op het internet zijn vele oplossingen voor het afspelen van muziek met een microcontroller te vinden. Meestal gebruikt men een eigen formaat voor de muziek. Soms ontwikkelt men daarbij een aparte pc-applicatie die de RTTTL-beltonen converteert naar het eigen formaat. RTTTL beschrijft een muziekstuk in één grote tekststring. Een voorbeeld staat in code 22.14.

**Code 22.14 : Een RTTTL-beschrijving als string in C.**

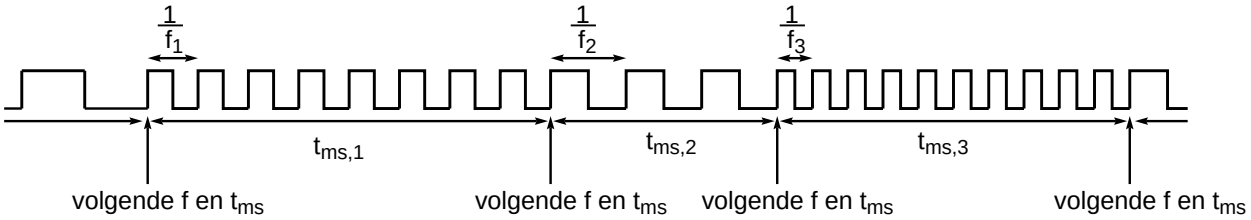
```
char rtttl[] = "Cocacola:d=4,o=5,b=125:8f#6,8f#6,8f#6,8f#6,g6,8f#6,e6,8e6,8a6,f#6,d6,2p";
```




**Figuur 22.19 : Aansluiting bij een piëzo-elektrische buzzer.**

Het formaat is zeer compact en relatief eenvoudig en is daarom zeer geschikt als formaat om eenstemmige muziekstukken in op te slaan. Een groot voordeel om RTTTL te gebruiken is dat er op het internet honderden beltonen in dit formaat beschikbaar zijn en dat deze niet geconverteerd hoeven te worden.

Bij het ontwikkelen van de microcontrollerapplicatie is eerst een pc-applicatie gemaakt die RTTTL-beltonen afspeelt. Voor deze applicatie zijn twee functies `readRTTTLdefaults` en `readRTTTLnote` ontwikkelt, die ook gebruikt kunnen worden bij de microcontrollerapplicatie. De beschrijving van de bibliotheek met deze functies staat in paragraaf E.2 van bijlage E. De functie `readRTTTLdefaults` leest de standaardparameters uit de RTTTL-string. Dat zijn de waarden van `d`, `p` en `b` die in de RTTTL-string tussen de dubbele punten staan. In code 22.14 zijn dat respectievelijk 4, 5 en 125. De functie `readRTTTLnote` leest een noot uit de RTTTL-string en geeft de frequentie en de tijdsduur van de noot via de parameterlijst terug. In code 22.14 is de eerste noot `8f#6` en de laatste noot `2p`.

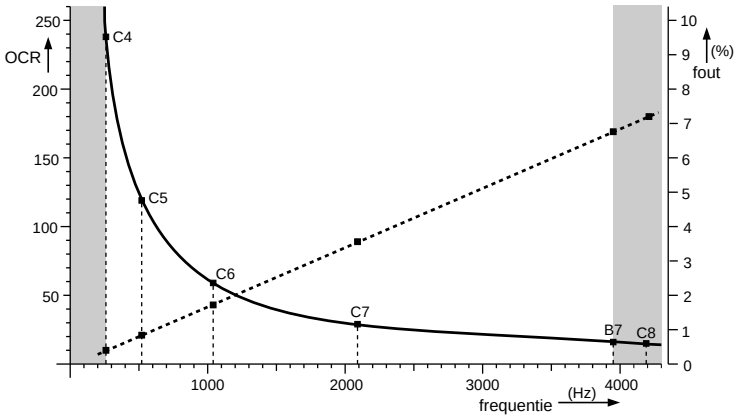



**Figuur 22.20 : Het uitgangssignaal voor het afspelen van muziek.** De microcontroller doet tijdens het afspelen niets, alleen als de tijdsduur $t_{ms}$ verstreken is, wordt een nieuwe frequentie en tijdsduur ingesteld.

De pc-applicatie uit paragraaf E.3 van bijlage E gebruikt een **`while`**-lus om een noot te lezen en af te spelen. De microcontroller gebruikt een timer die een PWM-signaal levert met de juiste frequentie. Het ligt voor de hand om de `OC`-uitgang van de timer te gebruiken. Tijdens het afspelen van de beltoon kan de microcontroller andere activiteiten uitvoeren. Alleen als de tijdsduur verstreken is, is er een interrupt nodig die uit de RTTTL-string een volgende noot leest en hieruit de nieuwe frequentie en tijdsduur bepaalt en instelt. In figuur 22.20 staat het uitgangssignaal met de overgangen tussen de verschillende noten.
Voor het vastleggen van de tijdsduur is eveneens een timer nodig. Er zijn dus twee timers nodig: één voor frequentie $f$ en één voor de tijdsduur $t_{ms}$.

**Argumenten voor de keuze van de timer voor de frequentie**

Voor het creëren van een PWM-signaal met een variabele frequentie en een duty-cycle van 50% is de CTC-modus geschikt. Een alternatief is om eventueel timer 1 in de fast-PWM modus te gebruiken met een variabele waarde voor TOP. Bij de CTC-modus geldt voor de frequentie vergelijking 22.4. Het frequentiebereik van de beltonen ligt tussen noot C4 (262 Hz) en noot B7 (3951 Hz). Als $f_{\text{cpu}}$ 8 MHz en de prescaling $P$ 64 is, ligt de waarde van OCR tussen 238 en 15. Deze waarden liggen binnen het bereik van timer 0 en timer 2 en royaal binnen het bereik van timer 1.




**Figuur 22.21 : De waarde van OCR als functie van de frequentie van de noot voor een prescaling van 64 en een systeemklok van 8 MHz.** Voor hoge frequentie is OCR klein en kan de afrondingsfout, aangegeven met de streeplijn, relatief groot zijn. De grijze gebieden vallen buiten de RTTTL-specificatie.

In figuur 22.21 is de waarde van OCR uitgezet tegen de frequentie $f$. Voor hoge frequenties liggen de waarden van OCR voor de verschillende noten dicht bij elkaar en wegen afrondingsfouten zwaar. Een afrondingsfout bij noot B7 is ongeveer 7%. Noot B7 kan daardoor als noot A7 of noot C8 klinken. Daar tegenover staat dat de meeste beltonen nauwelijks noten uit octaaf 7 bevatten en meestal zijn dat dan juist de lage tonen uit deze octaaf.

Als voor de prescaling 8 wordt genomen, ligt de waarde van OCR tussen 1907 en 125. Dit valt buiten het bereik van timer 0 en timer 2. Er moet dan gebruik gemaakt worden van timer 1. De afrondingsfout is over het hele bereik kleiner dan 1%. Bij een prescaling 1 is de afrondingsfout zelfs nihil en liggen de waarden van OCR tussen 15266 en 1011. Dit valt binnen het bereik van timer 1.

**Argumenten voor de keuze van de timer voor de tijdsduur**

De maximale tijdsduur die met een timer gemeten kan worden, hangt in lichte mate af van de gekozen PWM-modus. Ruwweg geldt:

$$T_{\text{max}} \approx \texttt{TOP}\, P\, T_{\text{cpu}} \tag{22.11}$$

Als $f_{\text{cpu}}$ 8 MHz is en de maximale prescaling $P$ 1024, is dat voor een 8-bits timer ongeveer 33 ms en voor een 16-bits timer 8,3 s. De periode van 33 ms is te kort. Er moet dus een 16-bits timer gebruikt worden. Of er kan, met een extra variabele

en een interruptfunctie, een bepaald aantal interrupts geteld worden en zo een langere tijd gemeten worden. Het nadeel is dat het interrumperen tijd kost en het voordeel is dat timer 1 beschikbaar blijft voor andere functies.

### Keuze van timers

Er is gekozen om timer 2 te gebruiken voor de frequentie en timer 0 met een interruptfunctie te gebruiken voor het definiëren van de tijdsduur. Timer 1 is dan beschikbaar voor andere functies. Als de onnauwkeurigheid in de frequentie te groot blijkt te zijn, kan timer 2 vervangen worden door timer 1.

**Code 22.15 : Het hoofdprogramma voor het afspelen van RTTTL-beltonen. ($f_{cpu} = 8$ MHz)**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include "rtttl.h"
4  #include "rtttl_lib.h"
5  #include "timers.h"
6
7  volatile char *rtttl;
8  volatile unsigned int freq;
9  volatile unsigned int ms;
10
11 void playRTTTL(char *p)
12 {
13   rtttl = readRTTTLdefaults(p);
14   ms = 0;
15   start_ms_timer();
16 }
17
18 int main(void)
19 {
20   DDRD  |= _BV(7);   // PD7/OC2 is output
21   PORTD &= _BV(7);   // PD7 low
22   sei();
23
24   playRTTTL(Furelise);
25   while (playingRTTTL) asm volatile ("nop");
26   playRTTTL(Wilhelmus);
27   while ( 1 ) asm volatile ("nop");
28 }
```

```
29 ISR(TIMER0_COMP_vect)
30 {
31   if (ms == 0) {
32     if ( readRTTTLtoken(rtttl) == '\0') {
33       stop_freq_timer();
34       stop_ms_timer();
35       return;
36     }
37     rtttl = readRTTTLnote((char *) rtttl,
38                           (unsigned int *) &freq,
39                           (unsigned int *) &ms);
40     if (freq == 0) {
41       stop_freq_timer();
42     } else {
43       OCR2 = (((uint32_t) F_CPU/freq) >> 7) - 1;
44       if ( freq_timer_off ) {
45         start_freq_timer();
46       }
47     }
48   }
49   ms--;
50 }
```

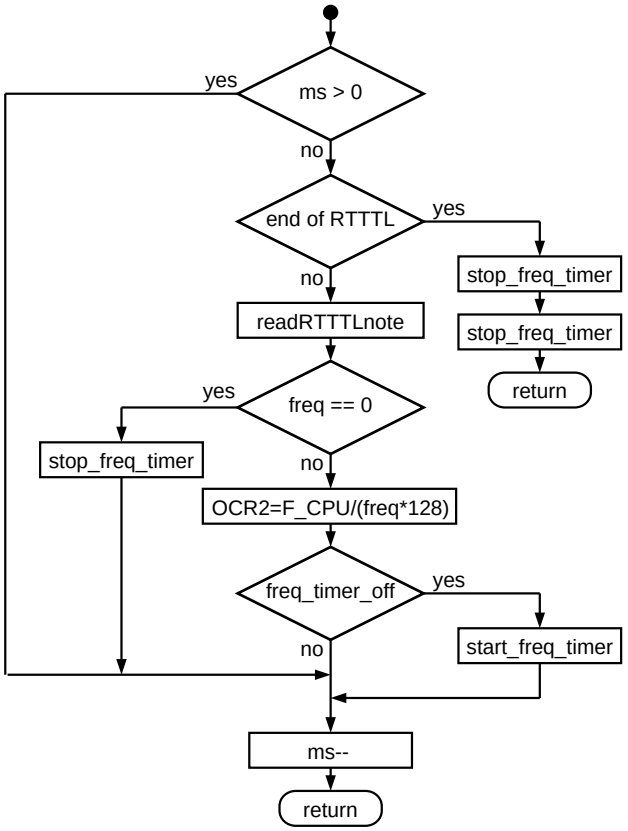
### Software voor het afspelen van muziek

In code E.9 van de bijlage E staat een functie `playRTTTL` die een RTTTL-beltoon afspeelt. Deze functie bevat een while-lus die steeds een noot leest en deze vervolgens laat horen. Bij de microcontrollerapplicatie kan het afspelen van een toon simultaan met andere activiteiten gedaan worden. Er is een interruptfunctie nodig, die geactiveerd wordt als de tijdsduur verstreken is, de volgende noot uit de RTTTL-string leest en hierna de nieuwe frequentie en tijdsduur instelt.

De `main` in code 22.15 initialiseert aansluiting 7 van poort D en zet de globale interrupt aan. Regel 24 start de beltoon *Für Elise* en bij de volgende regel wordt er gewacht tot de melodie klaar is. Daarna start op regel 26 het Wilhelmus en komt het programma in een oneindige wachtlus.

De standaardwaarden staan niet code 22.15. Het zijn globale variabelen, die gedeclareerd zijn in `rtttl_lib.c`, zie het fragment uit code E.2.

De functie `PlayRTTTL` van regel 11 leest uit de RTTTL-string de standaardwaarden, maakt `ms` nul en start de timer voor het meten van de tijdsduur. Omdat de timer voor het meten van de frequentie nog niet gestart is, zal er nog niets gebeuren. Na verloop van tijd treedt er een *output compare*-interrupt voor timer 0 op en wordt de interruptfunctie van regel 29 gestart.




**Figuur 22.22 : Het stroomdiagram met het algoritme voor de `ISR`.**

In figuur 22.22 staat de flowchart van de `ISR`. Er vijf scenario's:

- Als `ms` groter is dan nul, is de huidig tijdsduur nog niet verstreken. De waarde van `ms` wordt met één verlaagd.
- Als de RTTTL-string leeg is, is de beltoon helemaal afgespeeld. De timers stoppen en de ISR wordt verlaten.
- Als de frequentie van de nieuwe noot nul is, is er een rust. De timer van frequentie stopt. Er dan geen toon te horen. De tijdsduur krijgt wel een nieuwe waarde en de timer voor de tijdsduur blijft aan.
- Als de timer voor de frequentie uit staat en de nieuwe frequentie is ongelijk aan nul, wordt deze timer gestart.
- Tenslotte is er de normale gang van zaken: er wordt een nieuwe noot gelezen, een nieuwe waarde voor `OCR2` berekend en de nieuwe `ms` met één verlaagd.

De start- en stopfuncties voor de timers staan in een apart bestand `timers.c`. Er is een headerbestand `timers.h`, zie code 22.16, met de prototypen van deze functies en de macro's `playingRTTTL` en `freq_timer_off`.

**Code 22.16 : Het headerbestand `timers.h`.**

```
#define  playingRTTTL      (TCCR0 & (_BV(CS02)|_BV(CS01)|_BV(CS00)))
#define  freq_timer_off  (!(TCCR2 & (_BV(CS22)|_BV(CS21)|_BV(CS20))))

void start_ms_timer(void);
void stop_ms_timer(void);
void start_freq_timer(void);
void stop_freq_timer(void);
```

In code 22.17 staat de inhoud van `timers.c`. De functie `start_ms_timer` stelt timer `0` in. Het `OCR0`-register krijgt de waarde 124. De modus is CTC, de `OC0`-uitgang is niet aangesloten en de prescaling is 64. Met een $f_{\text{cpu}}$ van 8 MHz volgt uit formule 22.4 dat de frequentie 1 ms is. Nadat de output-compare interrupt is aangezet, zal er iedere milliseconde een interrupt zijn.
De functie `stop_ms_timer` maakt de `CS0`-bits laag. Timer `0` heeft dan geen klok en stopt met tellen. Bovendien wordt de *output compare*-interrupt uitgezet.
De macro `playingRTTTL` uit code 22.16 test of een of meer van de `CS0`-bits hoog is. Als dat het geval is, wordt er muziek afgespeeld en is `PlayingRTTTL` ongelijk aan nul. Als alle bits laag zijn, wordt er geen muziek afgespeeld en is `PlayingRTTTL` gelijk aan nul.

**Code 22.17 : Het bestand `timers.c` met de start- en stopfuncties voor de timers.**

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

void start_ms_timer(void)
{
  OCR0   = 124;                          // count from 0 to 124 = 125 counts
  TCNT0  = 124;                          // force interrupt next prescaled clock
  TCCR0  = _BV(WGM01)|_BV(CS01)|_BV(CS00);  // CTC, OC0 disconnected, clock/64 = 125 kHz
  TIMSK |= (1<<OCIE0);                   // enable interrupt timer 0
}

void stop_ms_timer(void)
{
  TCCR0  =  _BV(WGM01);                  // CTC, OC0 disconnected, clock none
  TIMSK &= ~_BV(OCIE0);                  // disable interrupt timer 0
}

void start_freq_timer(void)
{
  TCCR2 =_BV(WGM21)|_BV(COM20)|_BV(CS22);   // CTC,OC2 toggle, clock/64 = 125 kHz
}

void stop_freq_timer(void)
{
  TCCR2 = _BV(WGM21);                    // CTC,0C0 disconnected, clock none
}
```

De functie `start_freq_timer` stelt timer 2 in. De modus is CTC, de `OC2`-uitgang is aangesloten en is ingesteld op de *toggle*-modus. De prescaling is 64. Voor de frequentie $f$ van de noot geldt een vergelijking die overeenkomt met formule 22.4. De interruptfunctie berekent hieruit `OCR2` met:

$$\texttt{OCR2} \;=\; \frac{f_{\text{cpu}}}{2\,f\,P} - 1 \;=\; \frac{f_{\text{cpu}}}{128\,f} - 1; \tag{22.12}$$

Het delen door 128 is in code 22.15 geïmplementeerd door het getal 7 posities naar rechts te schuiven.
De functie `stop_freq_timer` maakt de `CS2`-bits laag. Timer 2 heeft dan geen klok en stopt met tellen. De macro `freq_timer_off` uit code 22.16 test of een of meer van de `CS2`-bits hoog is. Als `freq_timer_off` nul is, zijn de `CS2`-bits laag en staat de timer uit. Als `freq_timer_off` ongelijk aan nul is, is er op de timer een gedeelde klok aangesloten en telt de timer.

### De RTTTL-melodie in RAM, flash of EEPROM

De `rtttl`-bibliotheek bestaat uit het c-bestand `rtttl.c` en twee headerbestanden `rtttl_lib.h` en `rtttl.h`. Aan `rtttl.c` en `rtttl_lib.h` hoeft niets te worden aangepast. De definities van de af te spelen melodieën staan in `rtttl.h` en mogen aangevuld of aangepast worden.

In principe kunnen datagegevens op drie plaatsen staan: in RAM, flash of EEPROM. Het probleem is dat bij het lezen uit flash en EEPROM er speciale functies nodig zijn. Bij flash is dat de functie `pgm_read_byte` en bij EEPROM is dat `eeprom_read_byte`. De `rtttl`-bibliotheek uit bijlage E is zo opgezet dat alle drie de geheugens gebruikt kunnen worden. De bibliotheekfuncties `readRTTTLdefaults` en `readRTTTLnote` gebruiken twee macro's `readRTTTLtoken` en `readnextRTTTLtoken` die verschillend zijn voor RAM, flash en EEPROM. Het headerbestand `rtttl_lib.h` uit code E.3 bevat deze twee macro's. In `rtttl.h` staat eventueel de extra informatie, die er voor zorgt dat het flash of het EEPROM gebruikt wordt.

**Code 22.18 : Het headerbestand `rtttl.h` met de melodieën in RAM.**

```
1  #if !defined(__RTTTL_LIB__)
2
3  char Furelise[] = "FurElise:d=8,o=5,b=125:\
4  32p,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,32p,c,e,a,4b.,\
5  32p,e,g#,b,4c.6,32p,e,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,\
6  32p,c,e,a,4b.,32p,d,c6,b,2a";
7
8  char Wilhelmus[] = "Wilhelmus:d=4,o=6,b=90:d5,\
9  g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,2g5,\
10 p,d5,g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,\
11 2g5,p,8b5,8c,2d,e,2d,c,b5,8a5,8b5,c,b5,a5,g5,2a5,p,d5,\
12 8g5,8f#5,8g5,8a5,b5,2a5,g5,f#5,d5,8e5,8f#5,g5,g5,f#5,2g5";
13
14 #endif
```

Code 22.18 toont een voorbeeld waarbij de melodieën in RAM geplaatst worden. De voorwaardelijke preprocessoropdracht is nodig omdat dit headerbestand ook door `rtttl.c` gebruikt wordt en variabelen maar een keer gedeclareerd mogen worden. Verder bevat het alleen twee gewone stringdeclaraties.

De samenvatting van het geheugengebruik wordt bij AVRstudio gemaakt door het programma `avr-size` uit de AVR-bibliotheek.
Het flash (`Program`) bevat de programmacode (`.text`) en de variabelen (`.data`), die in de code een beginwaarde hebben. Deze zelfde variabelen staan ook in het RAM (`Data`), dat ook de variabelen zonder beginwaarde (`.bss`) bevat.
De term `.bss` staat voor *Block Started by Symbol*.

Na het bouwen van het `hex`-bestand meldt de compiler dat er 1762 bytes voor het programma nodig zijn en dat er 430 databytes zijn:

```
AVR Memory Usage
----------------
Device: atmega32

Program:    1762 bytes (5.4% Full)
(.text + .data + .bootloader)

Data:        430 bytes (21.0% Full)
(.data + .bss + .noinit)
```

In code 22.19 is met het attribuut `PROGMEM` aangegeven dat `Furelise` en `Wilhelmus` in het programmageheugen komen te staan. Het insluiten van `pgmspace.h` is noodzakelijk, omdat het de definitie van `PROGMEM` bevat en bovendien `__PGMSPACE_H_` dan gedefinieerd is. De test op regel 1 uit code E.3 is dan waar en de gegevens worden gelezen met `pgm_read_byte`.

**Code 22.19 : Het headerbestand `rtttl.h` met de melodieën in flash.**

```
1  #include <avr/pgmspace.h>
2
3  #if !defined(__RTTTL_LIB__)
4
5  char Furelise[] PROGMEM = "FurElise:d=8,o=5,b=125:\
6  32p,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,32p,c,e,a,4b.,\
7  32p,e,g#,b,4c.6,32p,e,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,\
8  32p,c,e,a,4b.,32p,d,c6,b,2a";
9
10 char Wilhelmus[] PROGMEM = "Wilhelmus:d=4,o=6,b=90:d5,\
11 g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,2g5,\
12 p,d5,g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,\
13 2g5,p,8b5,8c,2d,e,2d,c,b5,8a5,8b5,c,b5,a5,g5,2a5,p,d5,\
14 8g5,8f#5,8g5,8a5,b5,2a5,g5,f#5,d5,8e5,8f#5,g5,g5,f#5,2g5";
15
16 #endif
```

Na het bouwen van het `hex`-bestand meldt de compiler dat er 1866 bytes voor het programma nodig zijn en dat er 38 databytes zijn:

```
Program:    1864 bytes (5.7% Full)
(.text + .data + .bootloader)

Data:         38 bytes (1.9% Full)
(.data + .bss + .noinit)
```

De gegevens van de melodieën staan alleen in het programmageheugen. Het datageheugen bevat nu alleen de globale variabelen uit `rtttl_lib.c` en uit het hoofdprogramma.

In code 22.20 is met het attribuut `EEMEM` aangegeven dat `Furelise` en `Wilhelmus` in het EEPROM komen te staan. Het insluiten van `eeprom.h` is noodzakelijk, omdat het de definitie van `EEMEM` bevat en bovendien `_AVR_EEPROM_H_` dan gedefinieerd is. De test op regel 4 uit code E.3 is dan waar en de gegevens worden gelezen met `eeprom_read_byte`.

**Code 22.20 : Het headerbestand `rtttl.h` met de melodieën in EEPROM.**

```
1  #include <avr/eeprom.h>
2
3  #if !defined(__RTTTL_LIB__)
4
5  char Furelise[] EEMEM = "FurElise:d=8,o=5,b=125:\
6  32p,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,32p,c,e,a,4b.,\
7  32p,e,g#,b,4c.6,32p,e,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,\
8  32p,c,e,a,4b.,32p,d,c6,b,2a";
9
10 char Wilhelmus[] EEMEM = "Wilhelmus:d=4,o=6,b=90:d5,\
11 g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,2g5,\
12 p,d5,g5,g5,8a5,8b5,8c,8a5,b5,8a5,8b5,c,b5,8a5,8g5,a5,\
13 2g5,p,8b5,8c,2d,e,2d,c,b5,8a5,8b5,c,b5,a5,g5,2a5,p,d5,\
14 8g5,8f#5,8g5,8a5,b5,2a5,g5,f#5,d5,8e5,8f#5,g5,g5,f#5,2g5";
15
16 #endif
```

Na het bouwen meldt de compiler dat er 1580 bytes voor het programma nodig zijn, 38 databytes zijn en dat het EEPROM 391 bytes bevat:

```
Program:    1578 bytes (4.8% Full)
(.text + .data + .bootloader)

Data:         38 bytes (1.9% Full)
(.data + .bss + .noinit)

EEPROM:      391 bytes (38.2% Full)
(.eeprom)
```

Het datageheugen bevat bij het flash en EEPROM 38 bytes. Het EEPROM bevat 391 bytes. Het datageheugen van het EEPROM bevat 430 bytes. Dat is een byte meer dan de som van 391 en 38. Dat komt omdat 391 oneven is en 2-bytes grote integers altijd op een even adres staan.

Er worden nu twee bestanden gemaakt: een `hex`-bestand met de gegevens van het programma en een `eep`-bestand met de gegevens voor het EEPROM. Om de applicatie af te spelen moeten beide bestanden in de microcontroller geprogrammeerd worden.

Bij het vergelijken van de drie implementaties, valt op dat het benodigde programmageheugen bij de uitvoering met het RAM relatief groot is. Dat komt omdat het `.data`-segment met de vooraf gedefinieerde variabelen ook in het programmageheugen staat. Wanneer de microcontroller gestart wordt, plaatst de microcontroller eerst deze variabelen in het werkgeheugen. Bij de implementaties met flash en EEPROM worden de melodieën direct uit het flash of eeprom gelezen. Het initialiseren van het RAM is bij een simulatie met AVRstudio ook op een andere manier zichtbaar. Bij de implementatie met RAM zijn er voor de initialisatie 3877 klokslagen nodig en bij de andere twee implementaties zijn dat 349 klokslagen.

# 23

# Nog meer ATmega32

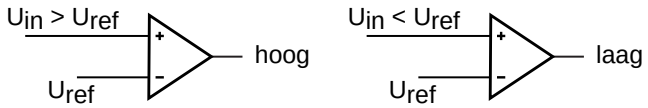
| | |
|---|---|
| Doelstelling | Dit hoofdstuk behandelt in vogelvlucht een aantal mogelijkheden van de ATmega32 die in de vorige hoofdstukken nog niet aan de orde zijn gekomen. |
| Onderwerpen | De behandelde onderwerpen zijn:<br>▪ De analoge comparator.<br>▪ Hysterese.<br>▪ De *input capture*-modus van timer 1.<br>▪ De slaapstanden van de ATmega32.<br>▪ Het gebruik van de zogenoemde *watchdog*.<br>▪ De brownoutdetectie.<br>Voorbeelden tonen:<br>▪ de toepassing van de analoge comparator;<br>▪ de toepassing van de analoge comparator met hysterese;<br>▪ het gebruik van de *input capture*-modus;<br>▪ het gebruik van de *input capture*-modus in combinatie met de analoge comparator;<br>▪ de slaapstand met de externe interrupt 0 als wekker;<br>▪ de slaapstand met timer 0 als wekker;<br>▪ de slaapstand met timer 2 als wekker;<br>▪ het gebruik van de watchdog;<br>▪ het gebruik van de watchdog om de microcontroller in een veilige toestand te zetten. |

De hoofdstukken 11 tot en met 13 bespreken een aantal basisaspecten van de microcontroller, zoals de generieke in- en uitgangen, het interruptmechanisme en timers. In een paar gevallen zijn daar ook een aantal specifieke oplossingen en mogelijkheden besproken, bijvoorbeeld de benadering van het flashgeheugen en het gebruik van opzoektabellen.
In hoofdstuk 18 tot en met 22 zijn het gebruik van de ADC, de aansturing van een LCD, de communicatie via de UART, de SPI en de $I^2C$-interface, de toepassing van het interne en externe EEPROM en het gebruik van de timers voor PWM-signalen uitgebreid besproken. Deze hoofdstukken tonen dat er vaak meerdere oplossingen zijn en laten zien dat het belangrijk is om een bepaalde eigenschap goed te bestuderen voordat deze gebruikt wordt.
Dit hoofdstuk geeft voor de nog niet besproken eigenschappen van de ATmega32 een beknopte beschrijving.

## 23.1 Analoge comparator

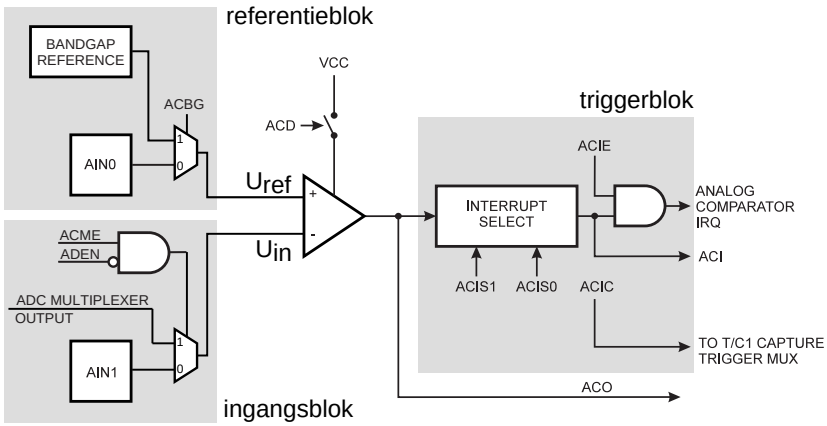
De ATmega32 heeft naast een ADC, *Analog-to-Digital Converter*, ook de beschikking over een analoge comparator. Een analoge comparator vergelijkt de ingangsspanning van de ene ingang met de andere ingang. Als de ingangsspanning van de ene ingang groter is dan de referentiespanning van de andere ingang is de uitgang van de comparator hoog en anders is de uitgang laag.

De keuze om een van de ingangen de referentie te noemen, is arbitrair. Een comparator vergelijkt twee ingangsspanningen. Als de spanning op de positieve ingang groter is dan de negatieve ingang, is de uitgang hoog.




**Figuur 23.1** : **Principe van een analoge comparator.** De uitgang is hoog als de ingangsspanning $U_{in}$ groter is dan de referentiespanning $U_{ref}$ en laag als deze kleiner is dan $U_{ref}$.

Een analoge comparator kan bijvoorbeeld gebruikt worden: om een batterijspanning te controleren, om de pulsbreedte van een signaal te meten of om nuldoorgangen te meten.

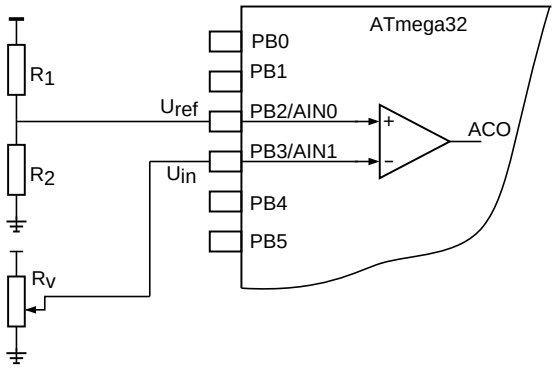



**Figuur 23.2** : **Blokschema van de analoge comparator van de ATmega32.** Het referentieblok geeft de referentiespanning $U_{ref}$, namelijk de interne referentie of aansluiting `AIN0` (pin 2 van poort B). Het ingangsblok levert het ingangssignaal $U_{in}$ via de multiplexer van de ADC of via aansluiting `AIN1` (pin 3 van poort B). Het triggerblok genereert uit de verandering van de uitgang `ACO` diverse triggersignalen.

In figuur 23.2 staat het blokschema van de analoge comparator van de ATmega32. De referentie-ingang $U_{ref}$ is de aansluiting `AIN0`, pin 2 van poort B, of het is een interne referentie van 1,23 V. De signaalingang $U_{in}$ van de comparator is de aansluiting `AIN1`, pin 3 van poort B, of is een van de ingangen van poort A. Als de ADC niet voor iets anders gebruikt wordt, kan de multiplexer een van de ingangen van `ADC0` tot en met `ADC7` selecteren en intern door verbinden met ingang $U_{in}$ van de comparator. De uitgang `ACO` van de comparator is verbonden met een triggerblok dat onder andere een interrupt kan geven als `ACO` van waarde verandert.

De referentie $U_{ref}$ is aangesloten op de positieve ingang van de comparator en ingang $U_{in}$ op de negatieve ingang. Bij de ATmega32 is de uitgang `ACO` hoog als $U_{in}$ kleiner is dan $U_{ref}$.

Voor de juiste instelling gebruikt de comparator het ACSR-register (*Analog Comparator control and Status Register*), het ACME-bit uit het SFIOR-register en voor het geval de multiplexer van de ADC gebruikt wordt ook het ADEN-bit uit het ADCSRA-register en het ADMUX-register om de juiste ingang te selecteren.




**Figuur 23.3 : Deel schema voor demonstratie analoge comparator.** De referentiespanning $U_{\text{ref}}$ is de spanningsdeling van de weerstanden $R_1$ en $R_2$. Het ingangssignaal $U_{\text{in}}$ kan worden ingesteld met de potentiometer $R_v$.

Figuur 23.3 toont een spanningsdeler met twee weerstanden $R_1$ en $R_2$ voor de referentiespanning $U_{\text{ref}}$. De ingang $U_{\text{in}}$ is aangesloten op een potentiometer. Het voorbeeldprogramma staat in code 23.1 en in code 23.2. Het laat een led, die aangesloten is op pin 0 van poort A, aan gaan als $U_{\text{in}}$ groter is dan de referentiespanning en uit gaan als deze lager is.

**Code 23.1 : De initialisatiefunctie voor de analoge comparator.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  void ac_init(void)
5  {
6    DDRB  &= ~(_BV(3)|_BV(2)); // AIN1 and AIN0 are inputs
7    PORTB &= ~(_BV(3)|_BV(2)); // no pull up
8    SFIOR &= ~(_BV(ACME));     // ACME,       AC multiplexer enable   : disable, use AIN0
9    ACSR   =                   // ACD,        AC Disable              : no disable
10                              // ACBG,       AC Band gap Reference   : no band gap, use AIN1
11                              // ACO,        AC Output               : (non relevant, status bit)
12                              // ACI,        AC Interrupt Flag       : (non relevant, status bit)
13           _BV(ACIE);         // ACIE,       AC Interrupt Enable     : interrupt on
14                              // ACIC,       AC Input Capture        : no input capture
15                              // ACIS1,ACIS0 AC Interrupt Mode Select: toggle mode
16 }
```

**Tabel 23.1 : Trigger modi voor analoge comparator.**

| ACIS1..0 | mode |
|---|---|
| 00 | toggle |
| 10 | falling edge |
| 11 | rising edge |

Code 23.1 bevat de initialisatiefunctie ac_init voor de analoge comparator. De aansluitingen 2 en 3 zijn de ingangen AIN0 en AIN1 van de comparator. De pullup-weerstanden moeten uitgeschakeld zijn. Het ACME-bit in het SFIOR-register is laag, omdat AIN1 als ingang gebruikt wordt. Het ACBG-bit is laag als AIN0 de referentie is. Het ACIE-bit is hoog omdat de applicatie gebruik maakt van de interrupt. De trigger modus is ingesteld op de *toggle mode*, zie ook tabel 23.1.

**Code 23.2 : De interruptfunctie en de hoofdroutine voor de analoge comparator.**

```
16  ISR(ANA_COMP_vect) {
17    if ( bit_is_set(ACSR, ACO) ) {
18      PORTA &= ~_BV(0);       // LED is on
19    } else {
20      PORTA |= _BV(0);        // LED is off
21    }
22  }
```

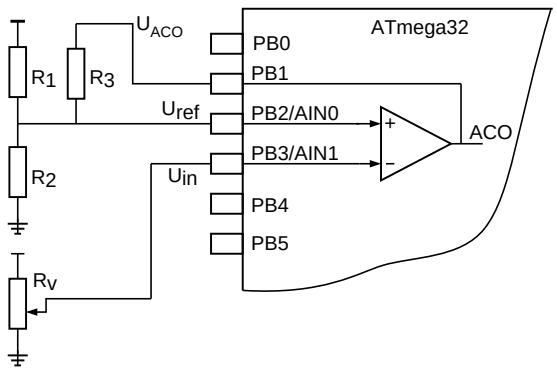
```
24  int main(void)
25  {
26    DDRA  |= _BV(0);
27    PORTA |= _BV(1);          // turn LED off
28    ac_init();
29    sei();
30
31    while(1) {
32      asm volatile ("nop");   // do nothing
33    }
34  }
```

Code 23.2 bevat de interruptfunctie en de hoofdroutine voor de applicatie uit figuur 23.3. Na de initialisaties van regel 26 tot en met 29 komt de hoofdroutine `main` in een oneindige lus, waarin niets gebeurt. Als de ingangsspanning $U_{\text{in}}$, die aangesloten is op pin `AIN1`, hoger of lager dan de referentiespanning $U_{\text{ref}}$ wordt, is er een interrupt en wordt de interruptfunctie op regel 16 uitgevoerd. Deze functie maakt de uitgangspin `0` van poort A laag, als de uitgang van de comparator `ACO` hoog is. De led gaat dan aan. Als de uitgang van de comparator `ACO` laag is, is de uitgang hoog en gaat de led uit.

**Tabel 23.2 : De signaalniveaus bij de applicatie met de comparator.**

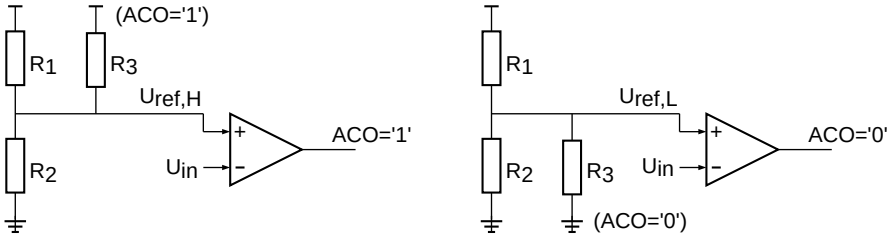
| **AIN1** ($U_{\text{in}}$) | **ACO** | **PA0** | led |
|---|---|---|---|
| laag | hoog | laag | aan |
| hoog | laag | hoog | uit |

In het geval dat de waarde van het ingangssignaal vlakbij de referentiewaarde ligt en er sprake van ruis is, zal de uitgang voortdurend wisselen. Vaak is dat niet gewenst. Het effect, dat op contactdender lijkt, kan soms softwarematig worden tegengegaan. Een interessante remedie is om hysterese toe te voegen. Paragraaf D.9 beschrijft de schmitttrigger en behandelt het verschijnsel hysterese. Hysterese ontstaat door een positieve terugkoppeling. Het effect van de hysterese is dat voor een toenemende waarde het omslagpunt bij hogere spanning en voor een afnemende waarde bij een lagere spanning ligt.




**Figuur 23.4 : Deel schema analoge comparator met positieve terugkoppeling.** De referentiespanning $U_{\text{ref}}$ hangt af van de weerstanden $R_1$, $R_2$ en $R_3$ en van de uitgangsspanning $U_{\text{ACO}}$.

In figuur 23.4 is de uitgang van de comparator softwarematig verbonden met pin 1 van poort B en via een weerstand $R_3$ verbonden met de aansluiting voor de referentiespanning `AIN0`. De referentiespanning $U_{\text{ref}}$ hangt af van de waarde van de weerstanden $R_1$, $R_2$ en $R_3$ en de waarde van de uitgangsspanning $U_{\text{ACO}}$. Deze spanning is gelijk aan $U_{\text{cc}}$ als `ACO` hoog is en `0` V als `ACO` laag is. Hierdoor zijn er twee verschillende configuraties met twee verschillende referentiespanningen. Figuur 23.5 toont de twee configuraties.




**Figuur 23.5 : De twee configuraties met de terugkoppelweerstand $R_3$.** Links staat de configuratie als $U_{\text{ACO}}$ hoog is en rechts die als $U_{\text{ACO}}$ laag is.

Als $U_{\text{in}}$ laag is, is $U_{\text{ACO}}$ hoog en heeft de referentiespanning een hoge waarde ($U_{\text{ref,H}}$) en als $U_{\text{in}}$ hoog is, heeft de referentiespanning een lage waarde ($U_{\text{ref,L}}$). Voor deze referentiespanningen geldt:

$$U_{\text{ref,H}} = \frac{R_2R_3 + R_1R_3}{R_1R_2 + R_2R_3 + R_1R_3}U_{\text{cc}} \tag{23.1}$$

$$U_{\text{ref,L}} = \frac{R_2R_3}{R_1R_2 + R_2R_3 + R_1R_3}U_{\text{cc}} \tag{23.2}$$

Als $U_{\text{cc}}$ gelijk is aan 5 V en $R_1$, $R_2$ en $R_3$ alle drie 10 kΩ zijn, dan is $U_{\text{ref,H}}$ 3,33 V en $U_{\text{ref,L}}$ 1,67 V. Voor grotere waarden van $R_3$ wordt de hystereselus smaller. Een weerstand van 120 kΩ geeft een $U_{\text{ref,L}}$ van 2,4 V en een $U_{\text{ref,H}}$ van 2,6 V.

De initialisatiefunctie voor de comparator met hysterese staat in code 23.3. Op regel 4 is aansluiting 1 van poort B als uitgang toegevoegd. De rest van de functie is ongewijzigd gebleven.

**Code 23.3 : De initialisatiefunctie voor de analoge comparator met hysterese.**

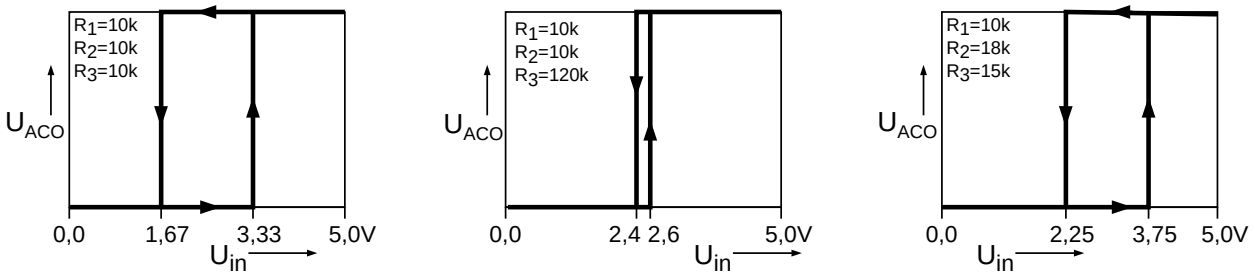
```
1  void ac_init(void)
2  {
3    DDRB  |= _BV(1);           // PB1 is output
4    DDRB  &= ~(_BV(3)|_BV(2)); // AIN1 and AIN0 are inputs
5    PORTB &= ~(_BV(3)|_BV(2)); // no pull up
6    SFIOR &= ~(_BV(ACME));     // ACME,       AC multiplexer enable   : disable, use AIN0
7    ACSR   =                   // ACD,        AC Disable              : no disable
8                               // ACBG,       AC Band gap Reference   : no band gap, use AIN1
9                               // ACO,        AC Output               : (non relevant, status bit)
10                              // ACI,        AC Interrupt Flag       : (non relevant, status bit)
11           _BV(ACIE);         // ACIE,       AC Interrupt enable     : interrupt on
12                              // ACIC,       AC Input Capture        : no input capture
13                              // ACIS1,ACIS0 AC Interrupt Mode Select: toggle mode
14 }
```

In code 23.4 staat de interrupt voor de oplossing met hysterese. De interruptfunctie maakt aansluiting 1 van poort B hoog als het ACO-bit hoog is en laag als dit bit laag is. De hoofdroutine van het programma blijft ongewijzigd.

**Code 23.4 : De interruptfunctie voor de analoge comparator met hysterese.**

```
17  ISR(ANA_COMP_vect) {
18    if ( bit_is_set(ACSR, ACO) ) {
19      PORTA &= ~_BV(0);       // LED is on
20      PORTB |=  _BV(1);       // PB1 is hoog
21    } else {
22      PORTA |=  _BV(0);       // LED is off
23      PORTB &= ~_BV(1);       // PB1 is laag
24    }
25  }
```

Figuur 23.6 geeft drie resultaten met verschillende verhoudingen voor de weerstanden $R_1$, $R_2$ en $R_3$. Als $R_1$ en $R_2$ even groot zijn, zijn de overgangen gecentreerd rond 2,5 V. Als $R_2$ groter is dan $R_1$ liggen de overgangen dichter bij 5 V. Een hoge waarde voor $R_3$ geeft een smalle hystereselus. De hysteresespanningen $U_{\mathrm{ref,H}}$ en $U_{\mathrm{ref,L}}$ liggen dan dicht bij elkaar.




**Figuur 23.6 : Drie verschillende hystereselussen.** De verhoudingen tussen de weerstanden $R_1$, $R_2$ en $R_3$ bepalen de hysteresespanningen $U_{\mathrm{ref,H}}$ en $U_{\mathrm{ref,L}}$.

## 23.2 Input capture

Timer 1 heeft een *input capture*-modus. Dat betekent dat de timer kan reageren op externe gebeurtenissen. De timer kan het tijdstip van de gebeurtenis vastleggen. Feitelijk is dat de waarde van de teller op het moment dat de gebeurtenis plaatsvindt. Met de *input capture*-modus kan de periodetijd van een periodiek signaal of de pulsbreedte van een puls worden bepaald.
De ingang van timer 1 is pin ICP1 — dat is pin 6 van poort D — of het is de uitgang ACO van de analoge comparator. Een gebeurtenis is een verandering van de ingang. De gevoeligheid wordt ingesteld met het bit ICES1, *Input Capture Edge Select* uit het register TCCR1B. Als dit bit hoog is, is de ingang gevoelig voor de opgaande flank en als het laag is, is de ingang gevoelig voor de neergaande flank. Het bit ICNC1 zet de *Input Capture Noise Canceler* aan. Als dit bit hoog is, is de ingang is dan minder gevoelig voor variaties.

Op het moment dat er een gebeurtenis optreedt, wordt de huidige waarde van TCNT1 in het ICR1-register geplaatst en wordt ICF1, *Input Capture Flag 1* hoog gemaakt. Als de interruptvlag TICIE1 actief is, wordt er bovendien de bijbehorende interruptfunctie uitgevoerd.

**Code 23.5 : Het meten van de pulsbreedte van een signaal. ($f_{cpu} = 1$ MHz)**

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>
#include "lcd.h"

volatile uint16_t pulse_start;
volatile uint16_t pulse_width;

ISR(TIMER1_CAPT_vect)
{
  if ( bit_is_set(PIND,6) ) {         // rising edge
    pulse_start = ICR1;
    TCCR1B &= ~(_BV(ICES1));          //   sensitive for falling edge
  } else {                            // falling edge
    pulse_width = ICR1 - pulse_start;
    TCCR1B |= _BV(ICES1);             //   sensitive for rising edge
    TCNT1=0;
  }
}

int main(void) {
  char buffer[16];

  lcd_init(LCD_DISP_ON);              // RS:PB0 R/W:PB1 E:PB2  DATA:PA7..4

  DDRD  &= ~(_BV(6));                 // PD6/ICP1 is input
  TCCR1A = 0x0;                       // normal mode
  TCCR1B = _BV(ICNC1)|_BV(ICES1)|     // noise canceler on & select rising edge
           _BV(CS10);                 // no prescaling
  TIMSK  = _BV(TICIE1);               // input capture interrupt enabled
  sei();

  while (1) {
    utoa(pulse_width, buffer, 10);
    lcd_clrscr();
    lcd_puts(buffer);
    _delay_ms(100);
  }
}
```

Code 23.5 meet de pulsbreedte van een periodiek signaal met een frequentie groter dan 20 Hz en plaatst de gemeten tijd op een LCD-scherm. Het signaal is aangesloten op ingang ICP1. De tijd is het aantal klokpulsen. Voor een klokfrequentie van 1 MHz komt dit overeen met de tijd in microseconden. Timer 1 werkt in de normale modus en de klok is niet gedeeld. De *input capture*-interrupt is geactiveerd.

Aanvankelijk is de timer gevoelig voor de opgaande flank van het signaal. Het ICR1-register krijgt de waarde van timer 1 op het moment dat deze gebeurtenis optreedt. De interruptfunctie test op regel 12 of pin 6 van poort D hoog is. Bij de opgaande flank is deze test waar. De waarde van ICR1 wordt onthouden in de variabele pulse_start en e ingang wordt gevoelig gemaakt voor de neergaande.

Na verloop van tijd wordt de puls laag. Dit activeert opnieuw het *input capture*-mechanisme en de interruptfunctie. De test van regel 12 is nu niet waar. Vervolgens wordt op regel 16 de pulsbreedte berekend uit de huidige waarde van ICR1 en start_puls en wordt de ingang weer gevoelig gemaakt voor de opgaande flank.

Op regel 18 wordt de teller op nul gezet. Dit voorkomt dat de waarde bij de neergaande klokflank lager is dan die bij de opgaande flank. Als de teller continu doorloopt kan de maximale waarde bereikt worden tussen de opgaande en de neergaande flank van het signaal en dat levert een verkeerde waarde op. In plaats van TCNT1 nul te maken, kan bij het berekenen van de pulsbreedte hiermee rekening worden gehouden.

```
        if (ICR1 > pulse_width) {
          pulse_width = ICR1 - pulse_start;
        } else {
          pulse_width = 65536 + pulse_start - ICR1;
        }
```

Het nadeel van deze oplossingen is dat deze meting alleen gebruikt kan worden bij signalen met een frequentie die groter is dan 20 Hz. Voor klokfrequenties die hoger zijn dan 1 MHz ligt de grens nog hoger.

Een heel andere oplossing is om niet alleen te reageren op veranderingen van het signaal, maar om ook het aantal keer dat de timer vol is te tellen en dit mee te nemen in de berekening. Code 23.7 is ook gevoelig voor de *timer overflow*-interrupt en de bijbehorende interruptfunctie hoogt een variable overflow met één op. Bij de flank van een puls wordt op regel 22 overflow nul gemaakt.
De pulsbreedte is het verschil tussen ICR1 en pulse_start verhoogt met overflow keer 65536. De variabele pulse_width is nu een 32-bits unsigned integer uint32_t. Bij de berekening op regel 22 is de vermenigvuldiging is geïmplementeerd met behulp van een schuiffunctie en is overflow getypecast naar uint32_t. Bovendien wordt nu op regel 41 de conversie ultoa gebruikt in plaats van utoa.

**Code 23.6 : Oneindige lus van hoofdprogramma met variabele new_pulse_width.**

```
40    while (1) {
41      if ( new_pulse_width ) {
42        new_pulse_width = 0;
43        ultoa(pulse_width, buffer, 10);
44        lcd_clrscr();
45        lcd_puts(buffer);
46      }
47      _delay_us(100);
48    }
```

**Code 23.7 : Het meten van de pulsbreedte via het tellen van de overflow. ($f_{cpu} = 1$ MHz)**

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>
#include "lcd.h"

volatile uint16_t pulse_start;
volatile uint32_t pulse_width = 0;
volatile uint16_t overflow = 0;

ISR(TIMER1_OVF_vect)
{
  overflow++;
}

ISR(TIMER1_CAPT_vect)
{
  if ( bit_is_set(TCCR1B,ICES1) ) {
    pulse_start = ICR1;
    overflow = 0;
  } else {
    pulse_width = ((uint32_t) overflow<<16) + ICR1 - pulse_start;
    new_pulse_width = 1;
  }
  TCCR1B ^= _BV(ICES1);
}

int main(void) {
  char buffer[16];

  lcd_init(LCD_DISP_ON);              // RS:PB0 R/W:PB1 E:PB2  DATA:PA7..4

  DDRD  &= ~(_BV(6));                 // PD6/ICP1 is input
  TCCR1A = 0x0;                       // normal mode
  TCCR1B = _BV(ICNC1)| _BV(ICES1) |   // noise canceler on & select rising edge
           _BV(CS11) | _BV(CS10);     // prescaling fcpu/64
  TIMSK  = _BV(TICIE1)|_BV(TOIE1);    // input capture & timer overflow interrupt
  sei();

  while (1) {
    ultoa(pulse_width, buffer, 10);
    lcd_clrscr();
    lcd_puts(buffer);
    _delay_ms(100);
  }
}
```

Het hoofdprogramma van code 23.7 ververst, net als dat van code 23.5, iedere 100 ms de informatie op het LCD. Beter is het om de informatie alleen te verversen als er ook een nieuwe pulsbreedte gemeten is. Er kan een **volatile** `uint8_t` variabele `new_pulse_width` worden gedeclareerd, die door de interrupt functie hoog gemaakt wordt als er een puls gemeten is. De oneindige lus uit code 23.6 ververst het LCD alleen als `new_pulse_width` ongelijk is aan nul.

In plaats van ingang ICP1 kan ook de analoge comparator worden gebruikt. In paragraaf 23.1 is de referentiespanning steeds verbonden met AIN0 en het te meten signaal met AIN1. Uitgang ACO is dan laag als het te meten signaal hoog is, zie ook tabel 23.2. Door de referentie met AIN0 en het te meten signaal met AIN1 te verbinden zal AC0 hoog zijn als het signaal hoog is.

**Code 23.8 : De initialisatie voor de analoge comparator voor input capture.**

```
1  void ac_init()
2  {
3    DDRB  &= ~(_BV(3)|_BV(2));  // AIN0 is here reference and AIN1 is input!
4    PORTB &= ~(_BV(3)|_BV(2));  // no pull up
5    SFIOR &= ~(_BV(ACME));      // disable multiplexer
6    ACSR  =                     // ACD,        AC Disable              : NO disable
7                                // ACBG,       AC Band gap Reference   : NO band gap
8                                // ACO,        AC Output               : (non relevant, status bit)
9                                // ACI,        AC Interrupt Flag       : (non relevant, status bit)
10                               // ACIE,       AC Interrupt enable     : interrupt OFF
11             _BV(ACIC);        // ACIC,       AC Input Capture        : input capture ON
12                               // ACIS1,ACIS0 AC Interrupt Mode Select: (non relevant, toggle mode)
13 }
```

De functie ac_init uit code 23.8 initialiseert de comparator. Pin 2 en 3 van poort B zijn de signaalingang en de referentie van de comparator. Omdat het ACIC-bit in het ACSR-register hoog is, is de uitgang ACO doorverbonden met de ingang van timer 1.

## 23.3 De slaapstanden

Tegenwoordig is het vermogensverbruik bij microcontrollers belangrijk. Zeer veel producten zijn draagbaar en gebruiken batterijen als voedingsbron. Het is natuurlijk plezierig als de batterijen klein zijn en lang meegaan. Het is daarom zaak de vermogensdissipatie van de microcontroller en de rest van de elektronica te beperken. Het gedissipeerde vermogen van een digitaal CMOS-IC hangt af van de totale capaciteit, de voedingsspanning $U$ en de frequentie $f$. De totale capaciteit $C$ is de capaciteit van de gates van de MOS-transistoren, van de verbindingen in het IC en de externe capaciteit die op de microcontroller aangesloten is. Het door het IC gedissipeerde vermogen $P$ is:

$$P = fCV^2 \tag{23.3}$$

Opmerkelijk is dat dit niet afhangt van de weerstand in het IC. De weerstanden bepalen samen met de capaciteiten wel de schakelsnelheid van het IC. Het gedissipeerde vermogen kan gereduceerd worden door de voedingsspanning te verlagen, de capaciteit klein te maken en frequentie laag te houden.
De meeste microcontrollers en microprocessoren kennen een of meer slaapstanden. Bij een slaapstand worden bepaalde delen van het IC uitgezet. Dit wordt gedaan door de klok van het betreffende deel uit te zetten. Uit de systeemklok

worden op de ATmega32 klokken afgeleid voor de CPU, het flash, de ADC. Daarnaast kan timer 2 nog een eigen asynchrone klok hebben.
De ATmega32 kent zes slaapstanden: *idle*, *ADC noise reduction*, *power-down*, *power-save*, *standby* en *extended standby*. Tabel 23.3 toont deze slaapstanden, de klokken die wel of niet actief zijn en de interrupts waarmee de microcontroller uit de slaapstand gehaald kan worden.

**Tabel 23.3 : De slaapstanden van de ATmega32.** De linker kolom geeft de modus. De volgende vijf kolommen geven de klokken die bij de slaapstanden uit of aan staan. In de laatste zes kolommen staan de interrupts waarmee de microcontroller weer wakker gemaakt kan worden.

| **M**odus | CPU/ flash | IO | ADC | Systeem **k**lok | **A**synchrone **o**scillator **t**imer 2 | **I**nterne interrupts | TWI **a**dres **m**atch | **T**imer 2 | SPM of EEPROM **r**eady | ADC | **O**verige **i**nterrupts |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Idle | uit | aan | aan | aan | aan[2] | X | X | X | X | X | X |
| ADC Noise Reduction | uit | uit | aan | aan | aan[2] | X[3] | X | X | X | X | |
| Power-down | uit | uit | uit | uit | uit | X[3] | X | | | | |
| Power-save | uit | uit | uit | uit | aan[2] | X[3] | X | X[2] | | | |
| Standby[1] | uit | uit | uit | aan | uit | X[3] | X | | | | |
| Extended standby[1] | uit | uit | uit | aan | aan[2] | X[3] | X | X[2] | | | |

[1] Er moet een extern kristal of resonator geselecteerd zijn als systeemklok.

[2] Het `AS2`-bit uit het `ASSR`-register moet hoog zijn.

[3] Geldt alleen voor interrupt `INT2` en voor `INT1` en `INT0` als deze niveaugevoelig zijn.

Bij *idle* is alleen de klok van de CPU en het flash uitgeschakeld. Met iedere interrupt kan de microcontroller uit deze toestand wakker gemaakt worden. De *ADC noise reduction* wordt bij metingen met de ADC gebruikt voor ruisonderdrukking. *Power-down* en *power-save* zijn de diepste slaapstanden. Het verschil is dat bij de laatstgenoemde timer 2 wel actief is. In tegenstelling tot *power-down* en *power-save* blijven bij de *stand-by*-standen de klokcircuits actief. Het voordeel hiervan is dat de microcontroller sneller uit de slaapstand is.
De *avr-libc*-bibliotheek bevat een headerbestand `sleep.h` met een aantal voorgedefinieerde functies om de microcontroller in een slaapstand te zetten. Code 23.9 brengt de microcontroller in de slaapstand *idle*. Als er een externe interrupt `0` is, wordt de microcontroller wakker en laat een led vijf keer knipperen en gaat opnieuw slapen.
Op regel 18 wordt de slaapmodus ingesteld op `SLEEP_MODE_IDLE`. In de oneindige lus wordt op regel 22 de slaapstand geactiveerd. De functies `set_sleep_mode` en `sleep_mode` zijn gedefinieerd in het headerbestand `sleep.h` dat op regel 3 is ingesloten. De microcontroller gaat na regel 22 slapen en wordt pas weer wakker als er een externe interrupt `0` is. Op regel 16 wordt deze interrupt aangezet. Bovendien wordt hiervoor op regel 19 het globale interrupt mechanisme aangezet.
Bij de interrupt wordt de microcontroller wakker en gaat dan verder met de interruptfunctie voor de externe interrupt `0`. Als deze `ISR` niet gedefinieerd is, krijgt de microcontroller automatisch een reset. Om dat te voorkomen is op regel 16 een lege `ISR` gedefinieerd. Na de interrupt gaat de microcontroller verder met de code van regel 24 die de led op uitgang `PA0` laat knipperen.

Het knipperen staat in de oneindige lus na de opdracht `sleepmode` van regel 22. Dit wordt pas uitgevoerd als er een interrupt is die de microcontroller weer wakker maakt. Als het knipperen in de `ISR` geplaatst wordt, doet het programma exact

**Code 23.9 : De slaapstand idle met een interrupt0 als wekker.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <avr/sleep.h>
4  #include <util/delay.h>
5
6  ISR(INT0_vect)
7  {
8  }
9
10 int main(void)
11 {
12   uint8_t i;
13
14   DDRA  = _BV(0);
15   PORTA = _BV(0);                    // output high, led off
16   GICR  = _BV(INT0);
17   MCUCR = _BV(ISC01)|!_BV(ISC00);    // interrupt falling edge
18   set_sleep_mode(SLEEP_MODE_IDLE);
19   sei();
20
21   while (1) {
22     sleep_mode();                    // go to sleep
23
24     for (i=0; i<10; i++) {           // blink five times
25       PORTA ^= _BV(0);
26       _delay_ms(500);
27     }
28   }
29 }
```

hetzelfde. Alleen demonstreert het dan niet dat het hoofdprogramma wakker is. Bovendien moet een gewone interruptfunctie, omdat deze niet interrumpeerbaar is, altijd zo leeg mogelijk zijn en een korte doorlooptijd hebben.

Op regel 17 is in code 23.9 de interrupt ingesteld op de neergaande flank. Deze interrupt werkt niet bij de *power-down*-modus. De externe interrupt 0 moet, om de microcontroller uit deze modus wakker te maken, niveaugevoelig zijn. Om de *power-down*-modus te gebruiken, moet regel 17 tot en met 18 vervangen worden door:

```
17   MCUCR = !_BV(ISC01)|!_BV(_BV(ISC00);     // level sensitive
18   set_sleep_mode(SLEEP_MODE_PWR_DOWN);
```

Code 23.10 gebruikt timer 0 om de microcontroller uit de slaapstand te halen. De timer is ingesteld op de CTC-modus, de prescaling is 1024 en telt tot 250. De *output compare*-interrupt en de globale interrupt zijn aangezet. Er is weer een lege interruptfunctie.
Bij een frequentie van 1 MHz is de tijd tussen de *output compare*-interrupts 256 ms. Deze interrupts zorgen er voor dat de microcontroller uit de slaapstand komt. Het effect is dat de led op uitgang PA0 met een frequentie van 0,5 s knippert.

**Code 23.10 : De slaapstand Idle met een timer 0 als wekker.**

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

ISR(TIMER0_COMP_vect)
{
}

int main(void)
{
  DDRA   = _BV(0);
  PORTA  = _BV(0);
  OCR0   = 250;
  TCCR0  =  _BV(WGM01)| !_BV(WGM00)|             // CTC mode
           !_BV(COM01)| !_BV(COM01)|             // OC0 disconnected
            _BV(CS02) | !_BV(CS01) | _BV(CS00);  // prescaling 1024
  TIMSK |= _BV(OCIE0);
  set_sleep_mode(SLEEP_MODE_IDLE);
  sei();

  while (1) {
    sleep_mode();
    PORTA ^= _BV(0);
  }
}
```

Het hoofdprogramma uit code 23.11 gebruikt uitgang OC0 om de led te laten knipperen. De uitgang OC0 is met de uitgangspin verbonden en ingesteld op de *toggle*-modus.

**Code 23.11 : De slaapstand Idle met een timer 0 als wekker en OC0 uitgang.**

```
int main(void)
{
  DDRB   = _BV(3);                              // OC0 output
  OCR0   = 250;
  TCCR0  =  _BV(WGM01)| !_BV(WGM00)|             // CTC mode
           !_BV(COM01)|  _BV(COM01)|             // OC0 toggle mode
            _BV(CS02) | !_BV(CS01) | _BV(CS00);  // prescaling 1024
  TIMSK |= _BV(OCIE0);
  set_sleep_mode(SLEEP_MODE_IDLE);
  sei();

  while (1) {
    sleep_mode();
  }
}
```

Timer 0 kan alleen bij de *idle*-modus gebruikt worden om de microcontroller wakker te maken. Als de slaapstand dieper is, moet timer 2 gebruikt worden. Code 23.12 is het alternatief van code 23.10 met timer 2 en de *power-save*-modus. Het AS2-bit uit het ASSR-register moet hoog zijn en er op de pinnen TOSC1 en TOSC2

**Code 23.12 : De slaapstand power-save met een timer 2 als wekker.**

```
 9  #include <avr/io.h>
10  #include <avr/interrupt.h>
11  #include <avr/sleep.h>
12
13  ISR(TIMER2_COMP_vect)
14  {
15  }
16
17  int main(void)
18  {
19    DDRA   = _BV(0);
20    PORTA  = _BV(0);
21    OCR2   = 250;
22    ASSR   = _BV(AS2);                              // external oscillator
23    TCCR2  = !_BV(WGM21)|  _BV(WGM20)|              // CTC mode
24             !_BV(COM21)| !_BV(COM21)|              // OC2 disconnected
25              _BV(CS22) |  _BV(CS21) | _BV(CS20); // prescaling 1024
26    TIMSK |= _BV(TOIE2);
27    set_sleep_mode(SLEEP_MODE_PWR_SAVE);
28    sei();
29
30    while (1) {
31      sleep_mode();
32      PORTA ^= _BV(0);
33    }
34  }
```

moet een extern kristal zijn aangesloten. De systeemklok en alle daaruit afgeleide klokken staan uit in deze modus.

### Het gedissipeerde vermogen bij de slaapstanden

Het vermogen dat een microcontroller dissipeert, wordt geleverd door de voedingsspanning. De stroom die de microcontroller trekt is een maat voor de dissipatie. In tabel 23.4 staat de stroom $I_{cc}$ die de spanningsbron levert. Er zijn metingen gedaan bij 1 en 8 MHz. In de laatste twee kolommen staan waarden die afgelezen zijn uit de grafieken van het hoofdstuk over de typische karakteristieken van de datasheet.

**Tabel 23.4 : De stroom $I_{cc}$ bij de ATmega32 voor verschillende slaapstanden.**

| slaapmodus | gemeten zonder pullup | | gemeten met pullup | | uit datasheet | |
|---|---|---|---|---|---|---|
| | 1 MHz | 8 MHz | 1 MHz | 8 MHz | 1 MHz | 8 MHz |
| geen | ∼5 mA | ∼15 mA | 1,8 mA | 12,3 mA | 1,9 mA | 13,0 mA |
| idle | ∼3 mA | ∼9 mA | 0,8 mA | 6,4 mA | 0,8 mA | 6,7 mA |
| power-down | ∼0,5 mA | ∼0,6 mA | 3,3 $\mu$A | 3,3 $\mu$A | 2,8$\mu$A | |

Kolom twee en drie geven de voedingsstroom als de niet-gebruikte ingangen van de microcontroller zwevend (*floating*) zijn. Kolom vier en vijf geven de stroom als de niet gebruikte ingangen via de pullupweerstand met de voeding verbonden zijn. Dit is gedaan door de betreffende PORT-registers hoog te maken.
De resultaten met de pullups komen overeen met de waarden uit de datasheet.

Duidelijk is dat zonder slaapstand en in de idle-modus de dissipatie bij 8 MHz acht keer zo groot is als bij 1 MHz. Het gedissipeerde vermogen in de powerdown-modus is zeer klein en onafhankelijk van de frequentie van de systeemklok. Dat kan ook niet anders, omdat deze klok in de powerdown-modus uitstaat.

De resultaten met de zwevende ingangen zijn zeer ruw. Door elektromagnetische interferentie worden de ingangen instabiel. De hoeveelheid lading op de ingangen varieert. Er gaan stromen lopen waardoor er extra vermogen wordt gedissipeerd. Het is mogelijk dat de microcontroller niet goed gaat functioneren en het is zelfs mogelijk dat de microcontroller stuk gaat. Het is daarom verstandig om alle ongebruikte ingangen altijd te definiëren, hetzij met een interne pullup of met een externe pullup of pulldown.

Verbind ongebruikte ingangen altijd via een pullup met de voedingsspanning.

## 23.4 De mogelijkheden om de ATmega32 te herstarten

De ATmega32 kent vijf verschillende mogelijkheden om de microcontroller te herstarten:

- *Power-on reset*
  Deze reset treedt op als de microcontroller wordt aangezet. Zolang de voedingsspanning beneden de grenswaarde $V_{\text{pot}}$ ligt, blijft de interne reset actief. Als de voedingsspanning boven $V_{\text{pot}}$ komt, wordt de interne reset inactief.
- *Externe reset*
  Dit is de reset van de gebruiker. Deze reset treedt op als $\overline{\text{RESET}}$-pin minstens 1,5$\mu$s laag is. Nadat de $\overline{\text{RESET}}$-pin hoog is geworden, wordt de interne reset inactief.
- *Watchdog-reset*
  Dit is de reset die optreedt als de watchdogtimer verlopen is.
- *Brownout-reset*
  Als de voedingsspanning — gedurende een bepaalde tijd – lager is dan $V_{\text{bot}}$ is er eveneens een reset.
- *JTAG-reset*
  Deze reset wordt gebruikt bij de JTAG-interface.

In al deze vijf gevallen wordt, nadat de reset voorbij is, de interne reset na een zekere tijd inactief. Deze tijd is de *startup time* en is instelbaar met de SUT-bits uit de fuseregisters en duurt minimaal zes klokslagen.

## 23.5 Watchdog

De watchdog bestaat uit een teller die regelmatig nul gemaakt moet worden. Als dat niet gedaan wordt, treedt er na verloop van tijd een interrupt op die de microcontroller laat herstarten.
De watchdog controleert of het microcontrollerprogramma correct functioneert. Tijdens de normale loop van het programma moet op regelmatige tijden de watchdogtimer nul gemaakt worden. In geval dat het programma vastloopt, blijft de timer doorlopen en treedt er na enige tijd een interrupt op en volgt er een watchdog-reset. De microcontroller start opnieuw op en kan weer normaal functioneren.

Voor embedded systemen is het watchdogmechanisme heel belangrijk. De methode is vooral nuttig voor de veiligheid van een systeem. Maar als een programma vastloopt, is de kans groot dat dit na het herstarten opnieuw gebeurt. Het is beter om het systeeem na het herstarten in een veilige toestand te brengen door alle eventueel gevaarlijke subsystemen — zoals motoren en hoogspanningscircuits — uit te zetten.

De ATmega32 heeft een `WDTCR`, *WatchDog Timer Control Register* met vijf bits. Het `WDTOE`-bit, *Watchdog Turn-off Enable*, en het `WDE`-bit, *WatchDog Enable*, waarmee de watchdog aan- en uitgezet kan worden. Met de bits `WDP2`, `WDP1` en `WDP0` kan de klok van de watchdogtimer gedeeld worden. Tabel 23.5 geeft een overzicht. Het register `MCUCSR` bevat de statusvlag `WDRF`, *WatchDog Reset Flag*, die hoog gemaakt wordt als er watchdogreset is. Dit bit wordt laag bij een power-on-reset of door naar deze vlag een logische nul te schrijven.

**Tabel 23.5 : De `WDP`-bits en de *time-out*-tijd van de watchdog.**

| naam in `wdt.h` | `WPD2..0` | time-out bij 5V |
|---|---|---|
| `WDTO_15MS` | `000` | 17,1 ms |
| `WDTO_30MS` | `001` | 34,3 ms |
| `WDTO_60MS` | `010` | 68,5 ms |
| `WDTO_120MS` | `011` | 0,14 s |
| `WDTO_250MS` | `100` | 0,27 s |
| `WDTO_500MS` | `101` | 0,55 s |
| `WDTO_1S` | `110` | 1,1 s |
| `WDTO_2S` | `111` | 2,2 s |

De *avr-libc*-bibliotheek bevat een headerbestand `wdt.h` met drie functies om de watchdogtimer te gebruiken: `wdt_enable`, `wdt_disable` en `wdt_reset`. Deze functies zetten respectievelijk de watchdog aan, de watchdog uit en de teller op nul. Bovendien bevat `wdt.h` acht macro's voor de instelling met de `WDP`-bits. In tabel 23.5 staan de namen van instellingen en de tijden die daar bij horen.

**Code 23.13 : Een basaal voorbeeld met de watchdog.**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3  #include <avr/wdt.h>
 4  #include <util/delay.h>
 5
 6  int main(void)
 7  {
 8    DDRA  = _BV(0);
 9    PORTA = _BV(0);
10
11    wdt_enable(WDTO_2S);  // enable watchdog
12
13    while(1)
14    {
15      wdt_reset();        // reset timer watchdog
16      PORTA ^= _BV(0);    // normal operation
17      _delay_ms(250);
18    }
19  }
```

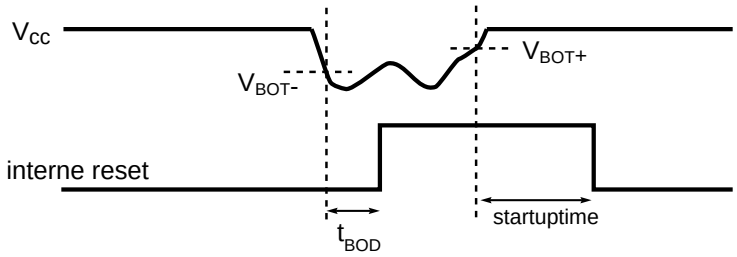
In code 23.13 staat het basisconcept voor het gebruik van de watchdogtimer. Op regel 11 wordt de watchdog aangezet en ingesteld op een wachttijd van 2,2 s. In de oneindige wachtlus wordt op regel 15 de watchdog steeds op nul gezet. In dit voorbeeld wordt bij normaal functioneren de wachtlus in 250 ms doorlopen. De led die met uitgang 0 van poort A verbonden is, zal voortdurend knipperen.
Als er een bug in de delay-functie zit en de microcontroller blijft in deze functie hangen, zal de led stoppen met knipperen. De watchdogtimer wordt niet meer op nul gezet en zal er na ruim 2 s een watchdog-interrupt zijn die de microcontroller herstart.

Als in code 23.13 het programma vastloopt bij de delay-functie, zal na het herstarten het programma waarschijnlijk weer vastlopen. In code 23.14 staat een structuur waarbij de microcontroller na het herstarten bij een watchdog-interrupt niet meer de normale functie uitvoert, maar een speciale veiligheidslus doorloopt. Na het herstarten is het WDRF-bit hoog. Op regel 10 wordt getest of dit bit hoog is. Na een power-on-reset is dit laag en gaat het programma verder bij regel 16 en voert de normale acties uit. Na een herstart door de watchdog is dit bit hoog en gaat het programma na de test van regel 16 verder met de oneindige lus van regel 11. In dit voorbeeld gaat er een andere led knipperen met een twee keer zo hoge frequentie.

**Code 23.14 : Een praktisch voorbeeld met de watchdog.**

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <avr/wdt.h>
4  #include <util/delay.h>
5
6  int main(void)
7  {
8    DDRA = _BV(1)|_BV(0);
9
10   if ( bit_is_set(MCUCSR, WDRF) ) {
11     while (1) {                        // safety loop
12       PORTA ^= _BV(1);
13       _delay_ms(125);
14     }
15   }
16   wdt_enable(WDTO_2S);
17
18   while (1) {                          // normal loop
19     wdt_reset();
20     PORTA ^= _BV(0);
21     _delay_ms(250);
22   }
23 }
```

Code 23.14 laat zowel in de normale functie als in de veiligheidsroutine een led knipperen. In een praktische situatie is de normale werking bijvoorbeeld een programma dat een aantal servomotoren aanstuurt en in de veiligheidslus de motoren in een veilige stand zet en een waarschuwingsled laat branden.




**Figuur 23.7 : Het genereren van een interne reset door middel van de brownoutdetectie.**

## 23.6 Brownoutdetectie

De ATmega32 heeft een on-chip brownoutdetector. Dit is een schakeling, die het niveau van de voedingsspanning in de gaten houdt. Als de spanning beneden een bepaald niveau komt, herstart de microcontroller door een interne reset. Figuur 23.7 laat zien dat als het niveau van de voedingsspanning te laag wordt er een interne reset gegeven wordt.
Het triggerniveau $V_{BOT}$ is instelbaar op 2,7 V of op 4,0 V. Feitelijk kennen de beide triggerniveaus twee waarden $V_{BOT+}$ en $V_{BOT-}$. De detector reageert op de lage waarde $V_{BOT-}$ en laat de interne reset los als de voedingsspanning boven $V_{BOT+}$ is gekomen. Samen met het gegeven dat het spanningsniveau minimaal een periode $t_{BOD}$ onder $V_{BOT-}$ moet zijn, zorgt deze hysterese ervoor dat de detector ongevoelig is voor spikes. Voor de ATmega32 is $t_{BOD}$ minimaal 2 $\mu$s.

De brownoutdetector wordt ingesteld met twee bits uit het fuseregister. De brownoutdetector wordt ingeschakeld door `BODEN`, *Brown-Out Detection ENable*, hoog te maken. Als `BODLEVEL`, *Brown-Out Detection LEVEL*, hoog is, is $V_{BOT}$ gelijk aan 2,7 V en anders is deze 4,0 V.

# A

# RS232

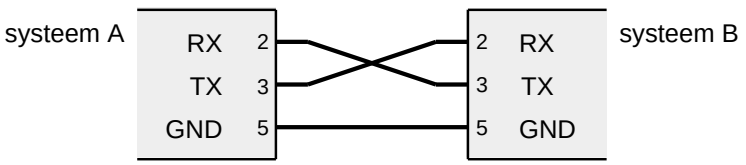
De RS232 (*Recommended Standard 232*) is een standaard voor asynchrone seriële communicatie. Het is een van de oudste communicatie protocollen voor het versturen van gegevens.

Asynchroon betekent dat er geen afspraak is tussen de bron en de ontvanger wanneer de gegevens worden verstuurd. Bij synchrone systemen is er vaak een apart kloksignaal of synchronisatiesignaal. Bij asynchrone communicatie kan de bron op elk moment zenden en dient de ontvanger op te letten of er gegevens zijn die ontvangen moeten worden.

Serieel betekent dat de informatie achter elkaar in de tijd wordt verstuurd. Bij parallelle communicatie worden de bits parallel verstuurd. Een voorbeeld is de ouderwetse parallelle printerpoort, die acht databits tegelijkertijd verstuurt. In principe is er voor het versturen met RS232 maar een lijn nodig. Over het algemeen zijn er meer lijnen bij betrokken. Bij RS232 zijn dat vaak twee signaallijnen voor het verzenden (TX) en het ontvangen (RX) en een ground-lijn (GND). Daarnaast zijn er soms een zes zogenoemde *handshake*-signalen. De DB9-connector van de seriële poort van een computer heeft daarom negen aansluitpinnen.

Voor het communiceren tussen twee computers of met een microcontroller zijn alleen de TX en de RX nodig. Figuur A.1 toont de driedraads RS232-verbinding: de TX van systeem A is verbonden met de RX van systeem B en omgekeerd.

Fullduplex is een verbinding waar de informatie in twee richtingen tegelijkertijd kan worden verstuurd. Bij halfduplex is er ook een verbinding in twee richtingen, maar dan kan de informatie na elkaar worden verstuurd. Een simplex verbinding is een verbinding in één richting.




**Figuur A.1 : Deze driedraads RS232-verbinding wordt een (fullduplex) nulmodemverbinding genoemd.** Hierbij is de TX van systeem A verbonden met de RX van systeem B en de TX van systeem B met de RX van systeem A. De getallen zijn de nummers van een standaard RS232-kabel.

Het RS232-protocol kent twee signaalniveaus: *marking* en *spacing*. Het signaalniveau van de *marking* is negatief en het niveau van de *spacing* is positief. Voor de zender ligt de *marking* tussen −5 V en −15 V en de *spacing* tussen 5 V en 15 V. Voor de ontvanger ligt de *marking* tussen −3 V en −25 V en de *spacing* tussen 3 V en 25 V. Voor elk apparaat kunnen de niveaus anders zijn. Bij een standaard desktop is dat vaak −12 en +12 V en bij een laptop is dat bijvoorbeeld −5,5 en 6,0 V.

### Het RS232-protocol

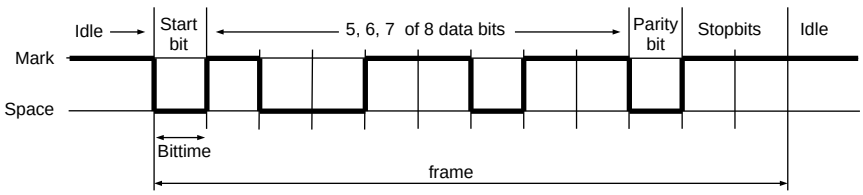
De verzender verstuurt de volgende informatie:

- een *mark* zolang de lijn *idle* is;
- een *space* als startbit voordat de databits verzonden worden;
- vijf, zes, zeven of acht databits;
- een, anderhalf of twee *marks* als stopbit nadat de databits verzonden zijn;
- tussen de databits en het stopbit mag eventueel een pariteitsbit worden meegezonden.

In figuur A.2 staat een voorbeeld met acht databits, een pariteitsbit en een stopbit. Het minst significante databit wordt eerst verzonden. In dit voorbeeld wordt de hexadecimale waarde D9 verstuurd, Binair is dat 11011001. De volgorde van de databits is dan 10011011.

**Tabel A.1 : RS232 snelheden.**

| bits per seconde |
|---|
| 300 |
| 1200 |
| 2400 |
| 4800 |
| 9600 |
| 19200 |
| 38400 |
| 57600 |
| 115200 |




**Figuur A.2 : Een voorbeeld van een RS232-protocol.** Naast het startbit heeft deze variant acht databits, een even pariteitsbit en een stopbit.

### De baud rate en het aantal bits per seconde

De *baud* (Bd) is genoemd naar Emile Baudot (1845-1903), een Frans ingenieur, die de naar hem genoemde Baudot-code voor de telex heeft bedacht. De telex wordt sinds februari 2007 in Nederland niet meer gebruikt.

De eenheid *baud* of *baud rate* is een term uit de datacommunicatie. Het geeft het aantal signaalveranderingen per seconde aan. De baud wordt vaak verward met het aantal bits dat per seconde wordt verstuurd. Door meer signaalniveaus te gebruiken kunnen er meer bits per baud verstuurd worden. Bij een eenvoudige RS232-verbinding tussen twee computers of tussen een computer en een microcontroller komt de *baud rate* wel overeen met het aantal bits per seconde.
De ontvanger moet de informatie met dezelfde snelheid lezen als de verzender de informatie verstuurt. Er is dus een afspraak tussen de ontvanger en de verzender nodig over de snelheid. Tabel A.1 geeft de standaard snelheden voor RS232-communicatie.

### Het schrijven via de RS232-poort

Code A.2 bevat de functies die in code A.1 gebruikt worden om alle 256 bytes van `0` tot `0xFF` via de RS232-poort te versturen. Code A.1 opent met `OpenComm` de communicatie, stuurt met `WriteCommBytes` de bytes naar de RS232-poort en sluit met `CloseComm` tenslotte de verbinding. Deze functies maken gebruik van de standaard communicatieroutines van Windows. Meer informatie hierover is te vinden op het Microsoft Developer Network.

**Uitleg code A.2 regel 7**
`OpenComm`

De functie `OpenComm` creëert op regel 12 met de functie `CreateFile` een zogenoemde *handle* voor de RS232-verbinding. Dit is vergelijkbaar met een filepointer en de functie `fopen`. De eerste parameter is de naam van de verbinding, bijvoorbeeld `"COM1:"`. Lukt het starten van de communicatie niet, dan wordt het programma afgesloten met `exit()`.
Met de `SetupComm` op regel 19 worden de buffergrootte voor het inkomende en uitgaande signaal op `4096` gezet.

Met `GetCommState` op regel 24 worden de instellingen van de poort opgevraagd en in de datastructuur `dcb` gezet. De toewijzingen van regel 29 tot en met 32 maken de *baud rate* 115200, het aantal databits acht en zorgen ervoor dat er geen pariteitsbit is en dat er slechts een stopbit is. Deze nieuwe instellingen worden op regel 34 met de functie `SetCommState` ingesteld.

**Regel 40**
`CloseComm`

De functie `CloseComm` sluit de verbinding met `CloseHandle`. Deze laatste functie is vergelijkbaar met de functie `fclose` voor het sluiten van bestanden.

**Regel 40**
`WriteCommByte`

De functie `WriteCommByte` verstuurt een byte over de geopende RS232-verbinding. Hiervoor wordt de functie `WriteFile` gebruikt.

**Code A.1 : Het versturen van gegevens via de COM-poort (main.c).**

De RS232-applicaties in deze bijlage zijn voor de pc. De applicaties voor de ATmega32 staan in hoofdstuk 20

```
1  #include <stdio.h>
2
3  #if defined (__CYGWIN__)
4  #include <unistd.h>
5  #else
6  #define sleep(__sec)     Sleep ((__sec)*1000)
7  #endif
8
9  void OpenComm(char *comm);
10 void CloseComm();
11 void WriteCommByte(unsigned char b);
12
13 int main(void)
14 {
15   int  i;
16
17   OpenComm("COM1:");
18
19   for (i=0; i<256; i++) {
20     WriteCommByte(i);
21     sleep(2);
22   }
23
24   CloseComm();
25
26   return 0;
27 }
```

**Uitleg code A.1 regel 3**
**`#if`**
**`#else`**
**`#endif`**
`defined`

De preprocessor heeft ook mogelijkheden voor voorwaardelijke compilatie. Dit maakt het mogelijk om C-code te schrijven, die geschikt is voor meerdere operating systems. De GNU-compiler van Cygwin kent een voorgedefinieerde macro `__CYGWIN__`. Als de code gecompileerd wordt met deze compiler is de uitdrukking `defined (__CYGWIN__)` waar en zullen de toewijzingen tussen de **`#if`** van regel 3 en de **`#else`** van 5 uitgevoerd worden. Als de code gecompileerd wordt met een andere compiler worden de statements na de **`#else`** van regel 5 en voor de **`#endif`** van regel 7 uitgevoerd.

**Regel 21**
`sleep()`

Windows kent een functie `Sleep()` en Unix een functie `sleep()` waarmee een programma een zekere tijd onderbroken kan worden. De Unix functie `sleep` onderbreekt het programma gedurende het opgegeven aantal seconden. De Windows functie `Sleep` onderbreekt het programma gedurende het opgegeven aantal milliseconden.

**Code A.2 : Het versturen van gegevens via de COM-poort (comm.c).**

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

char   comm[7];
HANDLE h;

void OpenComm(char *c)
{
  DCB dcb;

  strcpy(comm, c);
  if ( (h = CreateFile(comm, GENERIC_READ|GENERIC_WRITE,
                           0, NULL, OPEN_EXISTING,
                           FILE_ATTRIBUTE_NORMAL, NULL )) == INVALID_HANDLE_VALUE) {
    printf("Can't open comm port: %s\n", comm);
    exit(1);
  }

  if ( SetupComm(h, 4096, 4096) == 0 ) {
    printf("Can't setup comm port: %s\n", comm);
    exit(1);
  }

  if ( GetCommState(h, &dcb) == 0 ) {
    printf("Can't get state comm port: %s\n", comm);
    exit(1);
  }

  dcb.BaudRate = 115200;
  dcb.ByteSize = 8;
  dcb.Parity = NOPARITY;
  dcb.StopBits = 0;

  if ( SetCommState(h, &dcb) == 0 ) {
    printf("Can't set state comm port: %s\n", comm);
    exit(1);
  }
}

void CloseComm()
{
  CloseHandle(h);
}

void WriteCommByte(unsigned char b)
{
  unsigned long d;

  if ( WriteFile(h, &b, 1, &d, NULL) == 0 ) {
    printf("Can't write to comm port: %s\n", comm);
    exit(1);
  }
}
```

De voorwaardelijke compilatie van regel 3 tot en met 7 zorgt er voor dat onder Cygwin het juiste headerbestand wordt ingesloten en anders wordt er een `sleep` gedefinieerd op basis van de Windows `Sleep`. Op regel 21 wordt `sleep` gebruikt om twee seconden te wachten.

**Het lezen via de RS232-poort.**

Het lezen via de RS232-poort gaat op een zelfde manier. Aan code A.2 kan een functie `ReadCommByte` worden toegevoegd, die met behulp van de functie `ReadFile` een byte leest van de RS232-poort.

```
void ReadCommByte(char *buf)
{
  unsigned long n;

  ReadFile(h, buf, 1, &n, NULL);
}
```

Code A.3 past deze functie toe en drukt de hexadecimale waarde van de byte af op het scherm.

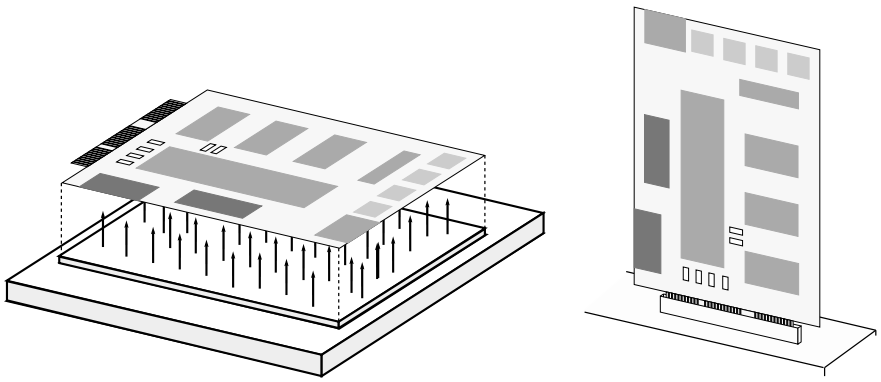
**Code A.3 : Het ontvangen van gegevens via de COM-poort (main.c).**

```
 1  #include <stdio.h>
 2
 3  void OpenComm(char *comm);
 4  void CloseComm();
 5  void ReadCommByte(unsigned char *b);
 6
 7  int main(void)
 8  {
 9    unsigned char c;
10
11    OpenComm("COM1:");
12
13    while ( 1 ) {
14      ReadCommByte(&c);
15      fprintf(stdout, "hexadecimal: %x\n", c);
16      fflush(stdout);
17    }
18
19    CloseComm();
20
21    return 0;
22  }
```

# B

# JTAG

In de jaren zeventig voorzagen belangrijke elektronicafabrikanten dat het testen van moderne elektronica steeds lastiger zou worden. De devices werden steeds kleiner en het aantal lagen in een *Printed Circuit Board* werd steeds groter.
Er zijn twee soorten testmethoden: een structurele test en een functionele test, zie ook figuur B.1. De structurele test controleert of de verbindingen correct zijn en dat er geen kortsluitingen zijn. De functionele test controleert het functionele gedrag van de schakeling.
Bij een productietest is een functionele test vaak lastig te realiseren. Bovendien duren deze tests vaak lang. Tijdens de productie is er behoefte aan een snelle methode, die controleert of de structuur van de PCB correct is en of de componenten correct gemonteerd zijn. Vroeger werd bij de structurele test een *bed of needles* toegepast. De PCB's werden op dit bed gelegd en de naalden maakten contact met de pinnen van de geïntegreerde schakelingen op de PCB. Vervolgens werden de verbindingen tussen deze componenten doorgemeten.

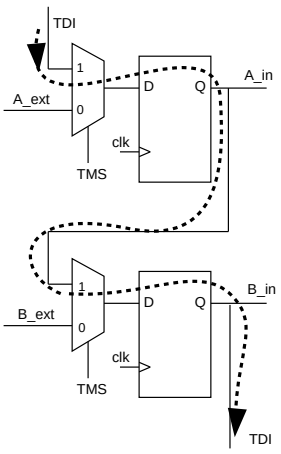


**Figuur B.1** : **Het testen van elektronica.** In de linker figuur staat een *bed of needles*. Hiermee worden de verbindingen tussen de componenten op een PCB getest. In deze opstelling wordt de fysieke structuur getest, of de verbindingen wel of niet aanwezig zijn. In de rechter figuur zijn alle normale in- en uitgangen van de PCB aangesloten op een testapparaat. Hiermee kan de elektronica functioneel worden getest.

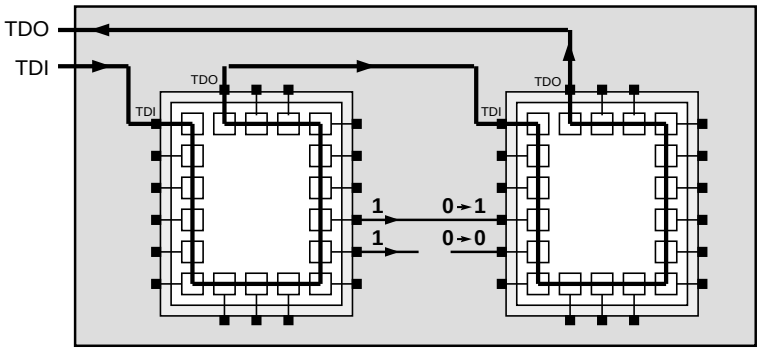
Philips is altijd zeer actief geweest met JTAG. Het bedrijf JTAG Technologies in Eindhoven is een spin-off van de vroegere activiteiten van Philips en is prominent op het gebied van *boundary scan*.

JTAG — opgericht door onder andere Philips, IBM, Texas Instruments — heeft een standaard ontwikkeld om PCB's te testen zonder een *bed of needles*. JTAG staat voor Joint Test Action Group. De standaard is de IEEE 1149.1 en wordt ook wel de JTAG-standaard of *boundary scan*-methode genoemd. Deze standaard maakt gebruik van *boundary scan*. Grote digitale IC's hadden bij de in- en uitgangen altijd al een flipflop om het in- of uitgangssignaal te bufferen. Bij *boundary*

*scan* is aan deze flipflop een multiplexer toegevoegd. De flipflop kent nu twee modes: de test mode en de normale mode. In de test mode kan via de multiplexer een testsignaal worden aangeboden, zie figuur B.2. Door alle in- en uitgangsflipflopppen via een seriële testlijn met elkaar te verbinden zijn alle in- en uitgangen van de component via een speciale testingang (TDI) bereikbaar en kan de informatie er via een speciale testuitgang (TDO) weer uit.
In de testmode vormen de in- en uitgangsflipfloppen van de IC's seriële registers. Door de testuitgang (TDO) van een IC te verbinden met de testingang (TDI) van een ander IC ontstaat er een lange seriële verbinding, zie figuur B.3.




**Figuur B.2** : **Twee *boundary scan*-flipfloppen.** In testmode (TMS hoog) is er een *scan path*, waar langs de testsignalen kunnen worden ingeklokt.



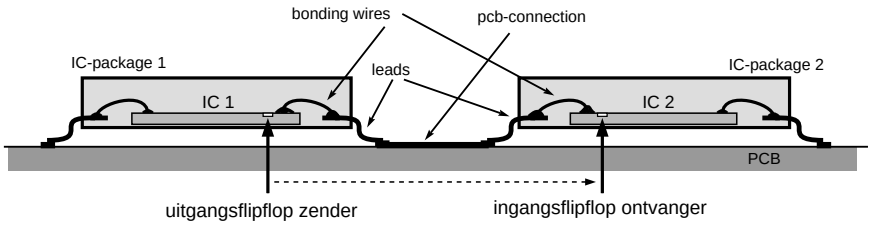

**Figuur B.3** : **Het zoeken naar fouten op een PCB met boundary scan.** Via het scan path worden in de testmode testsignalen naar de in- en uitgangen van de IC's gestuurd. Daarna functioneren de IC's een klokslag gewoon. De twee uitgangen van de IC 1 zijn allebei hoog. Als de verbinding met IC 2 in orde is, moet de ingang van IC 2 hoog zijn. Als de verbinding verbroken is, blijft de flipflop van de ingang laag. Door daarna weer in testmode te gaan en alle enen en nullen naar buiten te schuiven, kan de testengineer constateren dat een van de verbindingen tussen de IC's niet in orde is.

In de testwereld wordt een serie enen en nullen waarmee een test wordt uit gevoerd een testvector genoemd.

Op de PCB kan deze lange seriële verbinding — de *boundary scan* — gebruikt worden om testdata in en uit te lezen. Door een bekend testpatroon van enen en nullen het scanpad (*scan path*) in te schuiven en vervolgens de schakeling een klokslag normaal te laten functioneren, komt er via de PCB verbindingen een nieuw patroon in het scanpad te staan. Dit patroon van enen en nullen kan via het scanpad naar buiten worden geschoven en worden vergeleken met het verwachte patroon. Als er een defect aanwezig is op de PCB, zal het gelezen patroon verschillen van het verwachte patroon en wordt de PCB bij de productietest afgekeurd.
De *boundary scan*-methode test veel meer dan de PCB-verbinding. Ook de aansluitingen *leads* en de *bonding wires* en alle soldeerverbindingen worden getest. Alle onderdelen van de verbinding tussen het silicium van de IC's, dus ook de bonding wires in de packages, worden gecontroleerd, zie figuur B.4.

De JTAG-interface bestaat naast de boundary scan uit een zogenoemde TAP-controller. Dat is een toestandsmachine die het schuiven van de testdata regelt. JTAG heeft slechts vier of vijf signalen nodig:

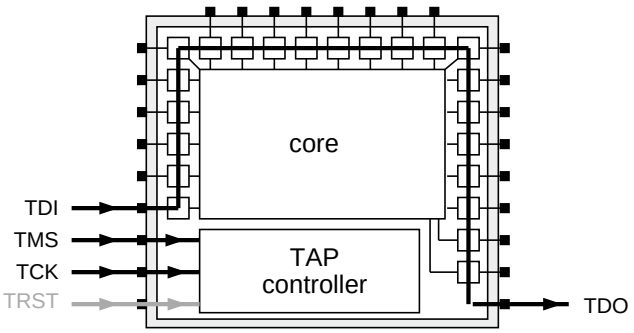
| | |
|---|---|
| `TDI` | Test Data In |
| `TDO` | Test Data Out |
| `TMS` | Test Mode Select |
| `TCK` | Test ClocK |
| `TRST` | Test ReSeT |




**Figuur B.4** : **Een dwarsdoorsnede van een PCB met IC's.** De verbinding tussen de twee IC's wordt helemaal gecontroleerd bij *boundary scan*: vanaf de flipflop van de uitgang van de zender (IC 1) tot aan de flipflop van de ingang van de ontvanger (IC 2), dus inclusief de *bonding wires*, de *leads*, de soldeerverbindingen en het PCB-spoor.

Het resetsignaal is niet altijd aanwezig en is ook niet per se noodzakelijk. Figuur B.5 toont een IC met de TAP-controller en de vijf aansluitingen.

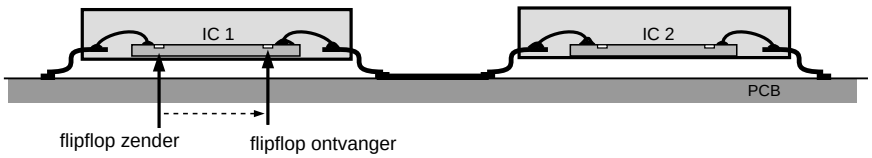
De *core* van een IC is het deel van het silicium zonder de in- en uitgangscellen. Het is de binnenkant — het hart — van het IC en bevat het functionele gedrag.




**Figuur B.5** : **Een IC met boundary scan en een TAP-controller.**

### Andere mogelijkheden met JTAG

JTAG maakt een IC serieel toegankelijk voor testvectoren. Figuur B.4 laat zien hoe de verbinding tussen de IC's getest kunnen worden. Deze test noemt men een *extest*. De *boundary scan*-methode kan gebruikt worden om de IC intern te testen. Door bij een ingangsflipflop een signaal te plaatsen kan het effect bij de uitgangsflipfloppen worden onderzocht, zie figuur B.6. Vaak worden er logica en




**Figuur B.6** : **De *intest* bij een IC op een PCB.**

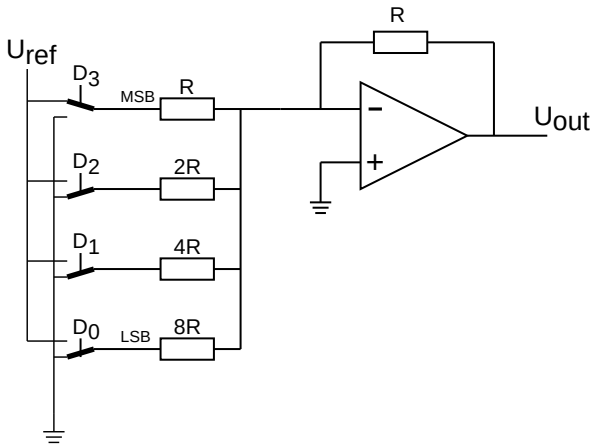
flipfloppen aan een IC toegevoegd om het volledig testbaar te maken. De registers en flipfloppen in de *core* van het IC worden aan het *scan path* toegevoegd. Als alle flipfloppen bereikbaar en leesbaar zijn is het IC volledig te testen. Het intern testen van IC's noemt men een *intest*

De toegankelijkheid via *boundary scan* kan ook gebruikt worden om een component *in system* te programmeren en te debuggen. De ATmega32 van Atmel heeft een JTAG-aansluiting, die hiervoor gebruikt kan worden. Via de JTAG-aansluiting en een *scan path* in het IC zijn alle registers uit te lezen.

# C

# Digital-to-Analog Converter

Een digitaal-analoogomzetter — *DAC, Digital-to-Analog Converter* — zet digitale signalen om in een analoog signaal. Er zijn vele methoden om een digitaal signaal analoog te maken. Deze bijlage bespreekt twee methoden: een DAC op basis van een gewogen sommatie en een DAC op basis van R/2R-laddernetwerk. De ATmega32 heeft — net als de meeste andere microcontrollers — geen DAC voor algemeen gebruik beschikbaar. De ADC van de ATmega32 gebruikt intern een DAC voor de successieve approximatie.




**Figuur C.1 : Een DAC op basis van een gewogen sommatie.** In dit voorbeeld is alleen ingang $D_3$ hoog en is alleen deze tak verbonden met $U_{\text{ref}}$.

## C.1 Een 4-bits DAC op basis van gewogen sommatie

Figuur C.1 geeft het basisschema voor een 4-bits DAC. Er zijn vier digitale ingangen $D_3$, $D_2$, $D_1$ en $D_0$, een analoge uitgang $U_{\text{out}}$ en een referentiespanning $U_{\text{ref}}$. De operationele versterker wordt als sommator gebruikt. De grootte van de bijdrage aan de som hangt af van de weerstandswaarde. De ingangstak voor $D_0$ telt acht keer minder zwaar mee dan de tak met $D_3$. De uitgangsspanning is de gewogen sommatie van de ingangen:

$$U_{\text{out}} = \left(D_3 + \frac{1}{2}D_2 + \frac{1}{4}D_1 + \frac{1}{8}D_0\right) U_{\text{ref}} \tag{C.1}$$

De vier ingangen $D_3$, $D_2$, $D_1$ en $D_0$ kunnen worden beschouwd als een digitaal getal $D$:

$$D = 8D_3 + 4D_2 + 2D_1 + D_0 \tag{C.2}$$

De uitgangsspanning hangt af van de digitale waarde $D$ en de referentiespanning:

$$U_{\text{out}} = \frac{D}{8} U_{\text{ref}} \tag{C.3}$$

De resolutie van de DAC op basis van een gewogen sommatie is gelijk aan de bijdrage van het minst significante bit.

$$\text{resolutie} = \frac{1}{8} U_{\text{ref}} = 0{,}125\ U_{\text{ref}}$$

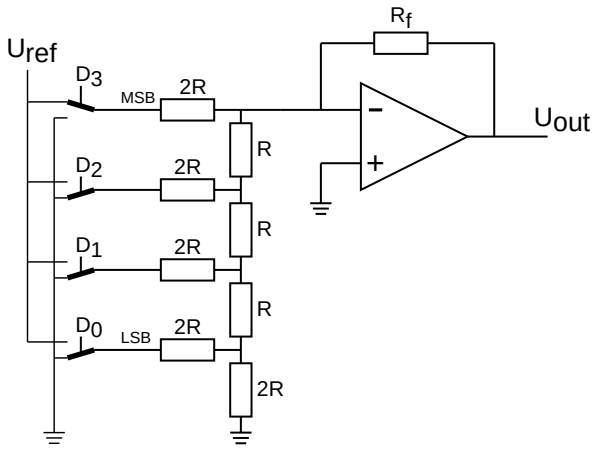
Het maximale bereik van deze DAC treedt op als $D$ gelijk is aan 1111, zodat:

$$\text{maximale bereik} = \frac{15}{8} U_{\text{ref}} = 1{,}875\ U_{\text{ref}}$$

Het nadeel van een DAC op basis van een gewogen sommatie is dat voor een DAC met een grote nauwkeurigheid zeer grote weerstanden nodig zijn en dat de weerstandswaarden een groot bereik hebben. Voor een 10-bits DAC met $R$ is 1 kΩ is de kleinste weerstandswaarde 1 kΩ en de grootste meer dan 500 kΩ. Het is uiterst moeilijk om deze grote waarde nauwkeurig op een geïntegreerde schakeling te realiseren.

## C.2 Een 4-bits DAC op basis van een laddernetwerk

In figuur C.2 staat een DAC op basis van een R/2R-laddernetwerk. Deze DAC heeft alleen maar weerstanden $R$, $2R$ en een terugkoppelweerstand $R_{\text{f}}$ die van dezelfde grootte-orde zal zijn. Dit circuit is juist heel goed als een geïntegreerde schakeling te realiseren en wordt daarom veel toegepast.




**Figuur C.2 : Een DAC op basis van een R/2R-laddernetwerk.**

De uitgangsspanning $U_{\text{out}}$ van de DAC op basis van een R/2R-laddernetwerk hangt af van de keuze voor de terugkoppelweerstand $R_f$. Als $R_f$ gelijk is aan $R$ dan geldt voor deze 4-bits DAC:

$$U_{\text{out}} = \left(\frac{1}{2}D_3 + \frac{1}{4}D_2 + \frac{1}{8}D_1 + \frac{1}{16}D_0\right) U_{\text{ref}} = \frac{D}{16} U_{\text{ref}} \tag{C.4}$$

Deze formule wordt hier niet bewezen. Wel wordt in paragraaf C.4 deze formule aannemelijk gemaakt. Het maximale bereik van deze 4-bits DAC is $\frac{15}{16} U_{\text{ref}}$. Door

een terugkoppelweerstand te kiezen die gelijk is aan $\frac{16}{15}R$, wordt het maximale bereik gelijk aan de referentiespanning en geldt voor de uitgangsspanning:

$$U_{\text{out}} = \frac{D}{15}U_{\text{ref}} \tag{C.5}$$

## C.3 Een n-bits DAC op basis van een laddernetwerk

Voor een n-bits DAC kiezen we een terugkoppelweerstand $R_f$ die $(2^n)/(2^n-1)$ keer zo groot is als $R$. De uitgangsspanning van de n-bits DAC op basis van een R/2R-laddernetwerk wordt dan:

$$U_{\text{out}} = \frac{D}{2^n-1}U_{\text{ref}} \tag{C.6}$$

De resolutie van deze DAC is weer gelijk aan de bijdrage van het minst significante bit.

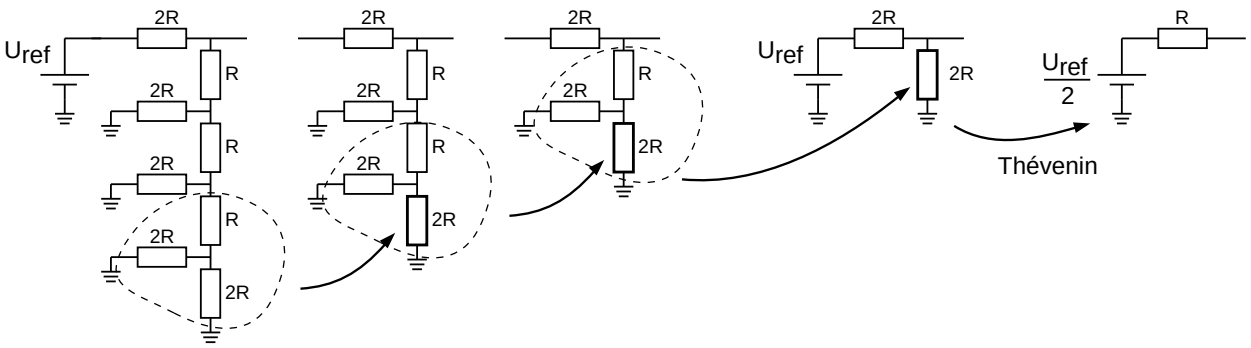
$$\text{resolutie} = \frac{1}{2^n-1}U_{\text{ref}} \tag{C.7}$$

Het maximale bereik van deze DAC vinden we als $D$ gelijk is aan $2^n-1$, zodat:

$$\text{maximale bereik} = U_{\text{ref}} \tag{C.8}$$

## C.4 Uitleg laddernetwerk

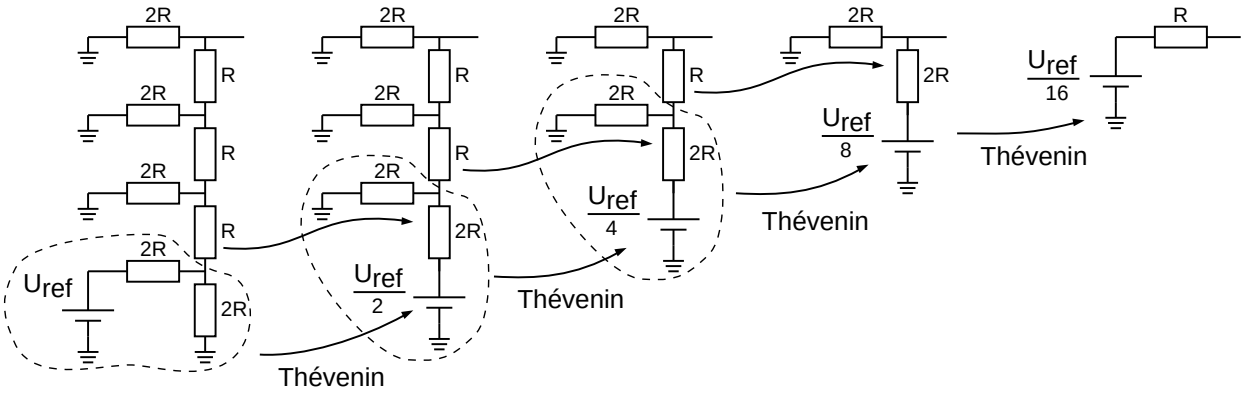
Het R/2R-laddernetwerk is een bijzondere schakeling, die ook in andere situaties wordt gebruikt. Het vervangingsschema is niet ingewikkeld, maar hoe je aan dit schema komt, wordt niet altijd begrepen. Hier worden twee bijzondere situaties besproken en daarna wordt een algemeen vervangingsschema voor de 4-bits DAC op basis van dit laddernetwerk gegeven.




**Figuur C.3** : **Het vervangingsschema van het laddernetwerk voor het meest significante bit.** Drie keer wordt een serieschakeling van een weerstand $R$ met een parallelschakeling van twee weerstand $2R$ vervangen door een weerstand $2R$. Tenslotte wordt Thévenin gebruikt om het vervangingsschema te maken.

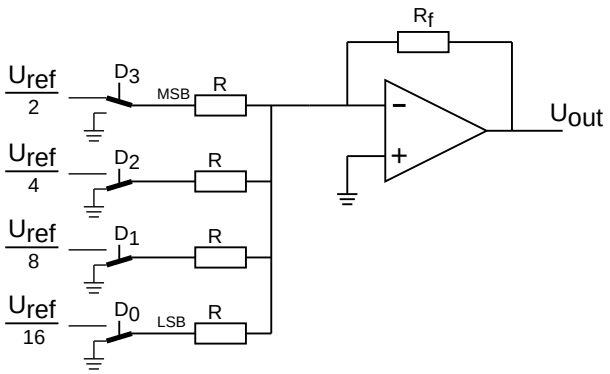
In figuur C.3 is de situatie gegeven als alleen het meest significante bit is aangesloten ($D = 1000$). Dit schema kan worden vereenvoudigd. De onderste twee weerstanden $2R$ vormen samen een parallelschakeling met als vervangingsweerstand $R$. De serieschakeling van deze $R$ en de $R$ daar direct boven geeft weer een

weerstand $2R$. Deze vervanging kan in het totaal drie keer worden uitgevoerd. Er blijft een schema over van een bron $U_{\text{ref}}$ en twee weerstanden $2R$. Het Thévenin-vervangingsschema hiervan bestaat uit een bron van $\frac{1}{2}U_{\text{ref}}$ en een weerstand ter grootte van $R$.




**Figuur C.4 : Het vervangingsschema van het laddernetwerk voor het minst significante bit.** Drie keer $R$ gecombineerd met het vervangingsschema van Thévenin. Tenslotte wordt Thévenin voor de vierde keer gebruikt om het uiteindelijke vervangingsschema te maken.

In figuur C.4 is de situatie gegeven als alleen het minst significante bit is aangesloten ($D = 0001$). De bron en de onderste twee weerstanden vormen een netwerk dat met Thévenin gereduceerd wordt tot een bron van $\frac{1}{2}U_{\text{ref}}$ en een weerstand van $R$. Deze weerstand staat in serie met de weerstand $R$ juist boven het gereduceerde netwerkdeel en geeft een vervangingsweerstand van $2R$. Door op dezelfde manier Thévenin nog twee keer toe te passen, wordt het hele netwerk gereduceerd tot een bron van $\frac{1}{8}U_{\text{ref}}$ en twee weerstanden van $2R$. Hierop kan nog een keer Thévenin worden toegepast. Het Thévenin-vervangingsschema bestaat uiteindelijk uit een bron van $\frac{1}{16}U_{\text{ref}}$ en een weerstand ter grootte van $R$.
De overige twee bits (bij $D = 0100$ en $D = 0010$) geven dezelfde vervangingsschema's. Alleen is de bron dan respectievelijk $\frac{1}{4}U_{\text{ref}}$ en $\frac{1}{8}U_{\text{ref}}$. Het laddernetwerk van de 4-bit DAC kan worden vervangen door vier aparte bronnen, zoals figuur C.5 laat zien. Formule C.4 is met dit netwerk eenvoudig af te leiden.




**Figuur C.5 : Het vervangingsschema van een 4-bit DAC op basis van een R/2R-laddernetwerk.**

# D

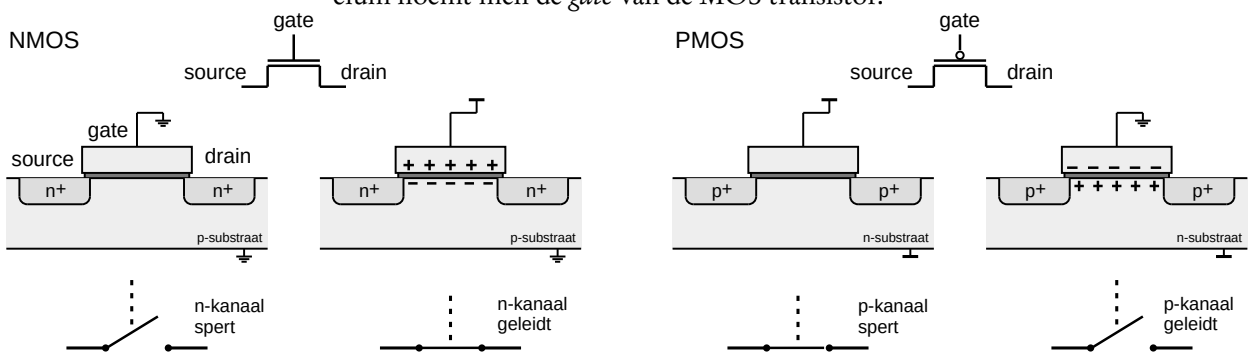
# CMOS

Een microcontroller is, net als de meeste digitale geïntegreerde schakelingen, gemaakt in CMOS (*Complementair Metal Oxide Semiconductor*). Om de datasheets van Atmel goed te kunnen begrijpen, is het handig om bekend te zijn met de CMOS-technologie.

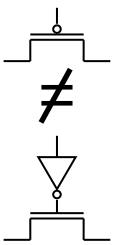
## D.1 De MOS-transistor als schakelaar

De MOS-transistor kan ook analoog gebruikt worden. De functionaliteit lijkt op de JFET. Men noemt de MOS-transistor daarom ook wel MOSFET.

Een MOS-transistor bestaat uit een substraat van halfgeleider materiaal (zwak gedoteerd silicium) met daarop een dunne isolatielaag (siliciumoxide, $SiO_2$) en daar bovenop een geleider van metaal (meestal aluminium of polysilicium). Het acroniem MOS staat dan ook voor *Metal Oxide Semiconductor*. Het metaal/polysilicium noemt men de *gate* van de MOS-transistor.

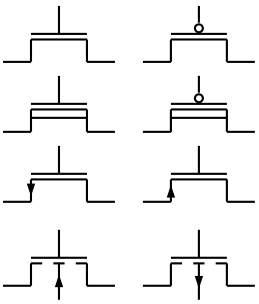



**Figuur D.1 : Een NMOS- en een PMOS-transistor.**
Links is het symbool en twee keer de dwarsdoorsnede van een NMOS-transistor getekend. Tussen de gate en het p-substraat bevindt zich de isolerende oxidelaag. Direct naast de gate bevinden zich de zwaar gedoteerde n-gebieden: de drain en de source. Bij de linker dwarsdoorsnede is geen lading op de gate aangebracht; er is dan geen kanaal en de transistor spert. Bij de rechter NMOS-transistor is wel een lading op de gate aangebracht. Er ontstaat dan een geleidend kanaal.
Rechts is het symbool en twee keer de dwarsdoorsnede van een PMOS-transistor getekend. Tussen de gate en het n-substraat bevindt zich de isolerende oxidelaag. Direct naast de gate bevinden zich de zwaar gedoteerde p-gebieden: de drain en de source. Bij de linker PMOS-transistor is geen lading op de gate aangebracht; er is dan geen kanaal en de transistor spert. Bij de rechter dwarsdoorsnede is wel een lading op de gate aangebracht. Er ontstaat dan een geleidend kanaal.



**Figuur D.2 : Let op! Een PMOS-transistor is geen inverter met een NMOS-transistor.**

Figuur D.1 laat zien dat links en rechts naast de gate twee zwaar gedoteerde gebieden zijn aangebracht. Voor een NMOS-transistor zijn dat zwaar gedoteerde n-gebieden in een zwak p-gedoteerd substraat. Voor een PMOS-transistor zijn dat zwaar gedoteerde p-gebieden in een zwak n-gedoteerd substraat.



**Figuur D.3 : Alternatieve symbolen voor MOS-transistoren.** Links staat steeds een NMOS- en rechts het bijbehorende PMOS-symbool. De onderste symbolen hebben het substraat of back-gate als vierde aansluiting.

Als bij een NMOS-transistor de gate laag (0) is sperren de pn-overgangen en kan er geen stroom lopen. Is de gate hoog (1) dan kan er wel een stroom lopen, omdat er onder het oxide een geleidend kanaal van elektronen ontstaat. Een PMOS-transistor spert als de gate hoog is en geleidt als de gate laag is; er ontstaat dan onder het oxide een geleidend 'gaten'-kanaal. Het gedrag van de p-transistor is dus complementair aan dat van de n-transistor.

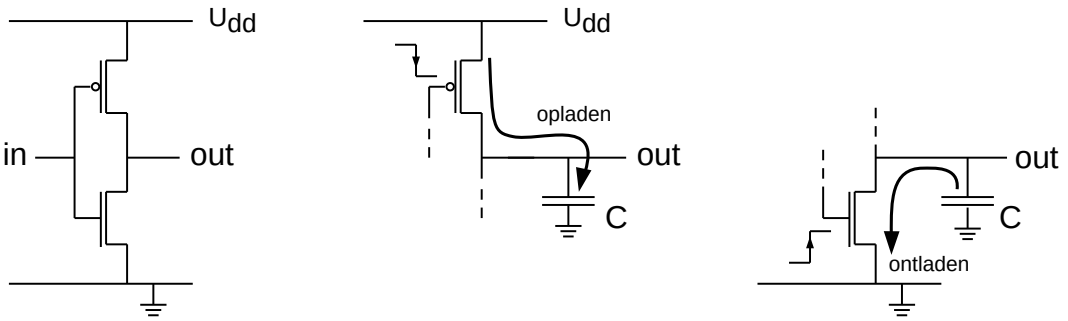
## D.2 De CMOS-inverter

De 'C' in het acroniem CMOS staat voor complementair. Dit betekent dat er in CMOS-schakelingen zowel NMOS-transistoren als PMOS-transistoren voorkomen.

In figuur D.4 staat links het schema van de CMOS-inverter, die opgebouwd is uit een NMOS- en een PMOS-transistor. Daarnaast staan de situaties bij het hoog en laag maken van de ingang. De capaciteit *C* bestaat uit de uitgangscapaciteit van de inverter, de ingangscapaciteiten van de aangesloten poorten en de bedradingscapaciteiten.

Als de ingang hoog is, geleidt de NMOS-transistor en spert de PMOS-transistor. De uitgang zal dan laag zijn. Wordt de ingang laag, dan zal de NMOS-transistor sperren en de PMOS-transistor gaan geleiden. Via deze PMOS-transistor wordt de uitgangscapaciteit (*C*) opgeladen.

Is de ingang laag, dan spert de NMOS-transistor en geleidt de PMOS-transistor, waardoor de uitgang hoog is. Wordt de ingang hoog, dan zal de PMOS-transistor sperren en de NMOS-transistor gaan geleiden. Via de NMOS-transistor wordt de uitgangscapaciteit (*C*) ontladen.




**Figuur D.4 : De bouw en de werking van de CMOS-inverter.**
Links staat de CMOS-inverter opgebouwd uit een NMOS- en een PMOS-transistor. De figuur in het midden laat zien dat de uitgang hoog wordt, als de ingang van hoog naar laag gaat. De NMOS spert en via de PMOS wordt de uitgang hoog. De rechter figuur laat zien dat de uitgang laag wordt, als de ingang van laag naar hoog gaat. De PMOS spert en via de NMOS wordt de uitgang laag.

IEEE staat voor *Institute of Electrical and Electronics Engineers* en is een internationale organisatie die elektrotechnische standaarden ontwikkelt. IEEE wordt uitgesproken als *eye-triple-e*.

Voor logische poorten bestaan twee soorten symbolen. In Amerikaanse literatuur worden symbolen uit de *ANSI/IEEE Std 91-1984*-norm gebruikt. Deze symbolen worden ANSI- of Amerikaanse symbolen genoemd. Het supplement *ANSI/IEEE Std 91a-1991* komt overeen met de symbolen uit *IEC 617-12: 1991*. Omdat deze symbolen door de IEC zijn ontwikkeld, worden deze de IEC- of Europese symbolen genoemd.

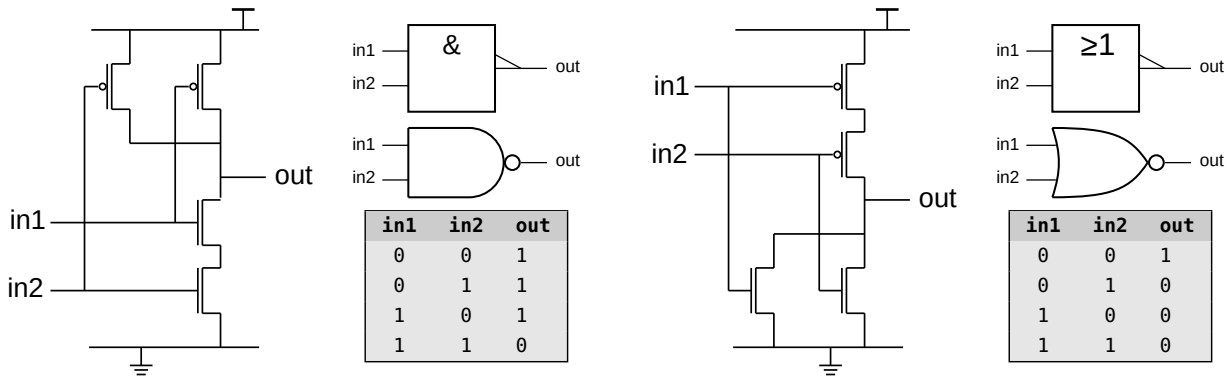

**Figuur D.5 :** **Het Amerikaanse en Europese symbool voor een inverter.** Links staat het Amerikaanse of ANSI-symbool en rechts staat het Europese of IEC-symbool.

IEC staat voor *International Electrotechnical Commission* en ontwikkelt internationale normen voor elektrische componenten en apparatuur. IEC is van oorsprong een Europese organisatie en werkt steeds meer samen met de van oorsprong Amerikaanse IEEE.

De datasheet van Atmel gebruikt de Amerikaanse symbolen. Om zo goed mogelijk bij de datasheet aan te sluiten gebruikt dit boek ook Amerikaanse symbolen. De kracht van de IEC-symbolen is dat alle eigenschappen van een component met het symbool vastliggen. Tegelijkertijd is dat ook de zwakte; vaak zijn de symbolen complex.

## D.3 CMOS-logica

In figuur D.6 zijn een NAND en een NOR met twee ingangen getekend. Voor de NAND geldt dat alleen als beide ingangen hoog zijn er een pad naar aarde is en dat in alle andere gevallen er een pad naar de voeding is. De uitgang is alleen laag als beide ingangen hoog zijn. Dit is ook te zien in de waarheidstabel, die naast de NAND staat. Voor de NOR geldt dat alleen als beide ingangen laag zijn er een pad naar de voeding is en dat in de andere gevallen er een pad naar aarde is. De uitgang is alleen hoog als beide ingangen laag zijn, zoals ook te zien is in de waarheidstabel, die naast de NOR staat.
Figuur D.6 geeft voor de beide poorten ook de Amerikaanse en de Europese symbolen.

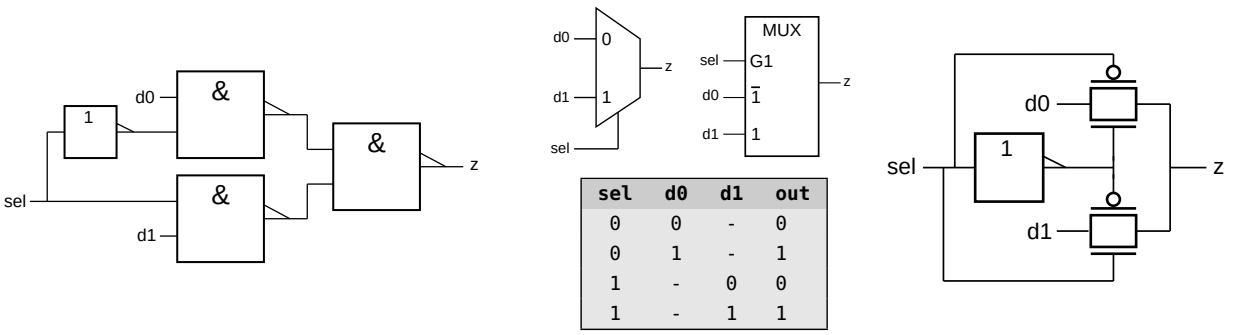



**Figuur D.6 :** **De NAND- en NOR-poort.**
Links staat de opbouw van een NAND met twee ingangen, de waarheidstabel, het logische symbool en het IEC-symbool. Het IEC-symbool staat bovenaan. Als de ingangen allebei hoog zijn, geleiden de NMOS-transistoren en sperren de PMOS-transistoren. De uitgang is dan laag. In alle andere gevallen is de uitgang hoog.
Rechts staat de opbouw van een NOR met twee ingangen, de waarheidstabel en de symbolen die voor deze poort gebruikt worden. Het IEC-symbool staat bovenaan. Als de ingangen allebei laag zijn, geleiden de PMOS-transistoren en sperren de NMOS-transistoren. De uitgang is dan hoog. In alle andere gevallen is de uitgang laag.

Omdat elektronen beter geleiden dan gaten, is het verstandig om de PMOS-transistoren niet in serie te gebruiken.
In de CMOS-technologie heeft de NAND-poort daarom de voorkeur boven de NOR-poort.

Het is niet ingewikkeld om een NAND met drie of vier ingangen te maken. Een NAND met drie ingangen heeft drie NMOS-transistoren die in serie staan en drie PMOS-transistoren die parallel staan.

Met inverters, NAND's en NOR's worden meer ingewikkelde componenten samengesteld. Figuur D.7 toont een 2-input multiplexer die is opgebouwd uit een inverter en drie NAND's. Omdat een inverter twee en een NAND vier transistoren heeft, bestaat deze multiplexer uit veertien transistoren. In figuur D.7 staat ook een multiplexer, die opgebouwd is uit een inverter en twee transmissiepoorten. De transmissiepoort wordt in paragraaf D.7 besproken. Inverters en transmissiepoorten bestaan allebei uit twee transistoren, zodat deze multiplexer uit slechts zes transistoren bestaat. In het midden staat de waarheidstabel. Als ingang `sel` laag is, staat de waarde van `d0` op de uitgang en als `sel` hoog is, staat de waarde van `d1` op de uitgang. Boven de waarheidstabel staan het Amerikaanse en het Europese symbool van de 2-input multiplexer.




| sel | d0 | d1 | out |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 0 | 1 | - | 1 |
| 1 | - | 0 | 0 |
| 1 | - | 1 | 1 |

**Figuur D.7 : Een 2-input multiplexer met NAND's en een 2-input multiplexer met transmissiepoorten.**
De multiplexer links heeft drie NAND-poorten en een inverter. Rechts staat een variant met twee transmissiepoorten en een inverter.
In het midden staat de waarheidstabel met daar boven het Amerikaanse en het IEC-symbool.

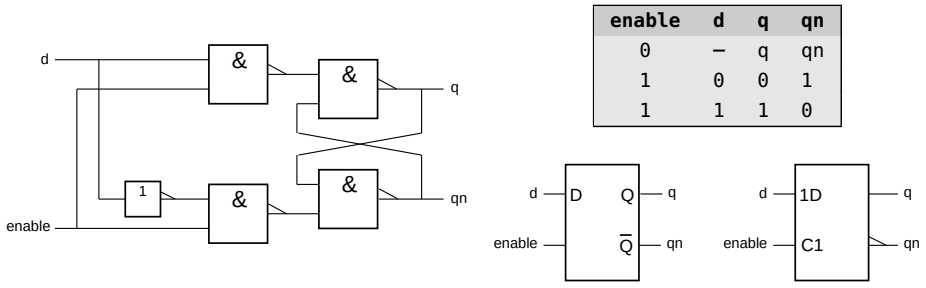
## D.4 De D-latch

Een D-latch is een component die de waarde van een signaal vast kan houden. De D-latch heeft een enable-ingang en een data-ingang. Als de enable actief is, wordt de informatie op de data-ingang direct doorgegeven naar de uitgang. Als de enable inactief is, houdt de latch de uitgangswaarde vast.

Sommige schrijvers noemen een D-latch ten onrechte een D-flipflop.

In figuur D.8 is een D-latch met een uitgang `q` en een geïnverteerde uitgang `qn` getekend. De D-latch is opgebouwd uit NAND-poorten. De waarheidstabel en de symbolen voor de D-latch staan rechts in figuur D.8.

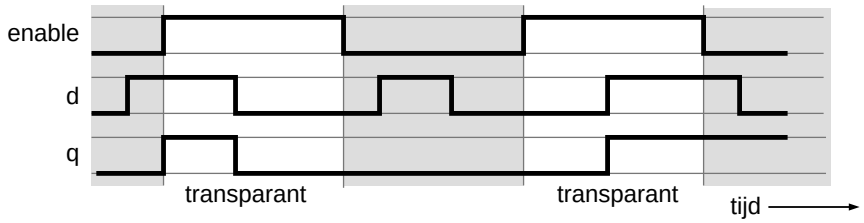
In dit boek is een D-latch altijd niveaugevoelig (*level sensitive*) en is een D-flipflop altijd flankgevoelig (*edge triggered*).




| enable | d | q | qn |
|---|---|---|---|
| 0 | – | q | qn |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Figuur D.8 : Een D-latch op basis van NAND-poorten.**
Links staat het schema. Rechts bovenaan de waarheidstabel en daar onder staan het Amerikaanse en het IEC-symbool.

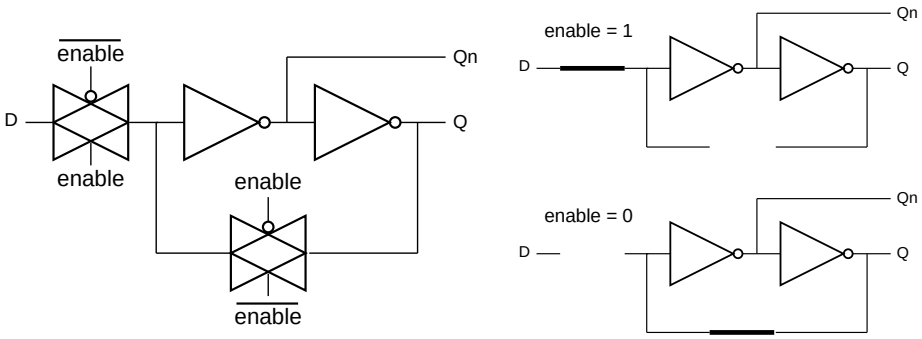
In figuur D.9 staat het tijdsdiagram van de D-latch. Duidelijk is te zien dat als het signaal `enable` hoog is — en dus actief is — uitgang `q` de data-ingang `d` volgt. De latch is dan transparant. Daarom noemt men deze component ook wel eens een transparante D-latch. Als `enable` laag is, verandert de waarde van `q` niet. De uitgang houdt de huidige waarde vast. De uitgang is van de ingang afgesloten. Dit verklaart de naam van de component: *latch* is het Engelse woord voor grendel of schuif.




**Figuur D.9 : Het tijdsdiagram van de D-latch.** De D-latch is transparant als het enable-signaal actief is en houdt de uitgangswaarde vast als het enable-signaal inactief is.

De generiek IO van de ATmega32, zie figuur 11.3, heeft een latch voor de ingangsflipflop `PINx`. Deze latch zorgt er dat het ingangssignaal gesynchroniseerd wordt.

Figuur D.10 toont een D-latch die uit twee inverters en twee transmissiepoorten bestaat. De transmissiepoort komt in paragraaf D.7 aan de orde. Voor deze D-latch zijn acht transistoren nodig in plaats van de achttien die bij het schema van figuur D.8 nodig zijn. Een nadeel is dat het enable-signaal geïnverteerd beschikbaar moet zijn.



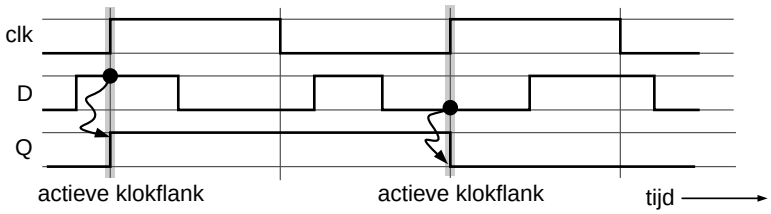

**Figuur D.10 : Een D-latch opgebouwd uit transmissiepoorten.**
Links staat het schema van een latch die is opgebouwd uit twee inverters en twee transmissiepoorten. Als `enable` hoog is, geleidt de eerste transmissiepoort en wordt het data-signaal doorgegeven naar de uitgangen. Als `enable` laag is, geleidt de transmissiepoort in de terugkoppellus en veranderen de uitgangssignalen niet.

Een bijkomend voordeel is dat de werking van een D-latch zo goed tot uiting komt. In figuur D.10 is het direct duidelijk dat als `enable` hoog is het signaal `d` doorgegeven wordt naar de uitgangen. En dat als `enable` laag is de terugkoppellus met de twee inverters het signaal vasthoudt.

Ondanks dat de D-latch de uitgangswaarde vast kan houden, is het geen ideale geheugenbouwsteen. Als het enable-signaal actief is, is de latch transparant. De D-latch is niveaugevoelig (*level sensitive*). Alleen als het enable-signaal heel kort actief is, is deze te gebruiken bij een synchrone sequentiële schakeling. Dat is ook de reden dat in deze paragraaf steeds gesproken is over een enable-signaal en niet over een kloksignaal.

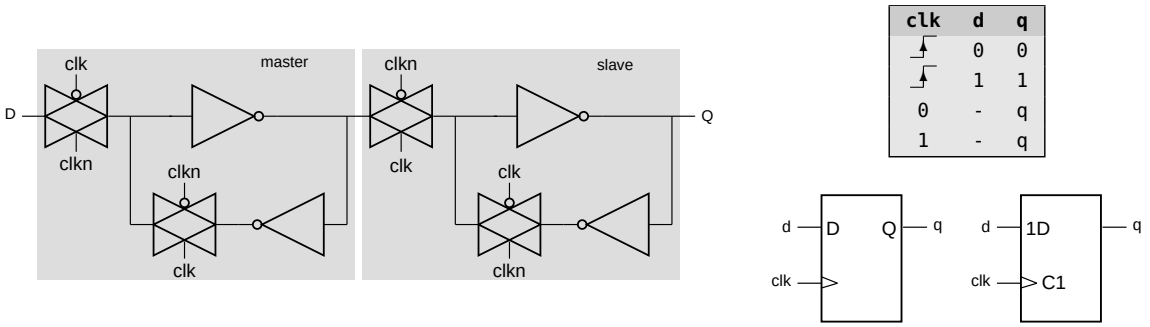
## D.5 De D-flipflop

De werking van de D-flipflop lijkt op die van de D-latch. In tegenstelling tot de D-latch is de D-flipflop niet niveaugevoelig (*level sensitive*), maar flankgevoelig (*edge triggered*). De uitgang neemt alleen bij de opgaande of neergaande klokflank de waarde van de ingang over. Hierdoor is deze component zeer geschikt als geheugenelement in een synchrone sequentiële schakeling.
Figuur D.12 geeft de waarheidstabel en de symbolen van een positieve flankgevoelige D-flipflop. De waarheidstabel en het tijdsdiagram uit figuur D.11 laten zien dat de uitgang `q` de waarde van `d` overneemt bij de positieve klokflank. In alle andere situaties behoudt de uitgang de waarde die deze al had.




**Figuur D.11 :** **Het tijdsdiagram van een D-flipflop.** De D-flipflop neemt alleen bij de actieve klokflank de ingangswaarde over. In alle andere situaties verandert de uitgang niet. De flipflop in dit voorbeeld is gevoelig voor de opgaande klokflank.

Het schema van de positieve flankgevoelige D-flipflop uit figuur D.12 bestaat uit twee in serie geschakelde D-latches. De eerste — de *master* — is aangesloten op het geïnverteerde kloksignaal `clkn` en de tweede — de *slave* — is aangesloten op het kloksignaal `clk`.

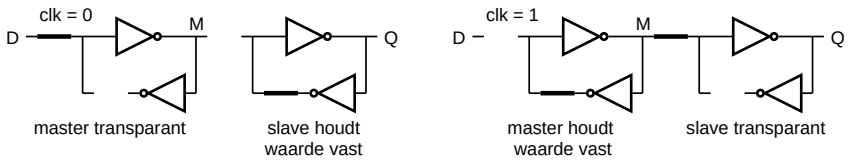



**Figuur D.12 :** **Een D-flipflop op basis van transmissiepoorten.**
Links staat het schema van een D-flipflop die is opgebouwd uit transmissiepoorten. In feite bestaat de flipflop uit twee in serie geschakelde D-latches: een *master* met een geïnverteerde klok (`clkn`) en een *slave* met een klok `clk`. Rechts bovenaan staat de waarheidstabel en daar onder staan het Amerikaanse en het IEC-symbool.

Sommige schrijvers noemen de D-latch en de D-flipflop respectievelijk D-flipflop en master-slave flipflop.

Vanwege de bouw spreekt men over een master-slave flipflop en vanwege het gedrag over een edge-triggered flipflop.

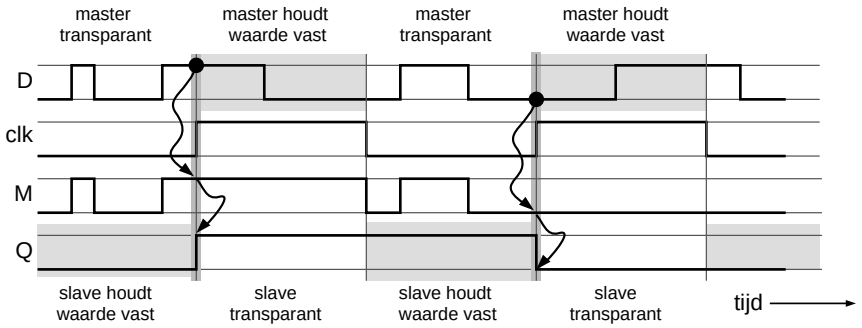
Figuur D.13 demonstreert het gedrag van de D-flipflop uit figuur D.12. Als het kloksignaal `clk` laag is, is de master transparant en houdt de slave de huidige uitgangswaarde vast. De uitgang `M` van de master volgt voortdurend het ingangssignaal `D`, maar de uitgang `Q` houdt voortdurend de huidige waarde vast.
Op het moment dat de klok hoog wordt, verandert de master naar de toestand waarin deze de waarde van knooppunt `M` vasthoudt. Veranderingen van het ingangssignaal `D` worden nu niet meer doorgegeven. De master onthoudt dus de waarde die `D` had bij de positieve, actieve klokflank. De slave is nu transparant en geeft de waarde van `M` door naar de uitgang `Q`.




**Figuur D.13** : **Het gedrag van een D-flipflop uit figuur D.12.** Links staat de situatie als de klok `clk` laag is. De master is dan transparant en de slave houdt de huidige uitgangswaarde vast. Links staat de situatie als de klok `clk` hoog is. De slave is dan transparant en geeft de waarde van knooppunt `M` door aan de uitgang.

Zolang de klok hoog blijft, verandert `M` niet en dus ook de uitgang niet. Alleen bij de actieve, opgaande klokflank kan de uitgang veranderen.
Het tijdsdiagram van figuur D.14 geeft ook het signaal van de uitgang `M` van de master. Als de klok laag is, volgt `M` ingang `D` en houdt `Q` de huidige waarde vast. Als de klok hoog is, verandert `M` niet meer. De slave geeft de laatste waarde van `M` door naar uitgang `Q`. Het totale effect is dat de uitgang van de flipflop alleen bij de actieve klokflank verandert.

D-flipfloppen hoeven niet altijd een master en slave te hebben. Wel zijn ze altijd flankgevoelig. Anders zijn het D-latches.




**Figuur D.14** : **Het tijdsdiagram van de D-flipflop inclusief de uitgang van de master.** Als de klok laag is, is de master transparant en volgt signaal `M` ingang `D`. Als de klok hoog is, verandert `M` niet meer en geeft de slave deze waarde door aan de uitgang. Bij de opgaande klokflank neemt uitgang `Q` de waarde van de ingang `D` over.

Een D-flipflop functioneert alleen goed als het datasignaal niet vlak bij de actieve klokflank verandert. De setuptijd $t_{\text{setup}}$ is de tijd dat het datasignaal stabiel moet zijn voor de klokflank. De holdtijd $t_{\text{hold}}$ is de tijd dat het datasignaal stabiel moet blijven na de klokflank. In figuur D.15 is een situatie getekend waar voldaan wordt aan de setuptijd en een situatie waarin dat niet het geval is. De uitgang van de flipflop wordt dan metastabiel en blijft dan gedurende een onbepaalde tijd hangen tussen de `0` en de 1.

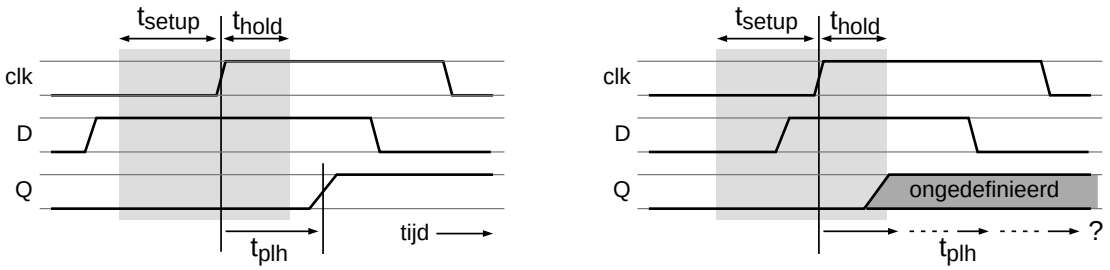
Een MTBF die heel kort is, ontdekt men — als het goed is — bij het testen.

Een MTBF die bijvoorbeeld een maand is, merkt men bij testen meestal niet op. De gebruiker van het product zal dit in de loop van de tijd wel merken.

Een MTBF van honderd jaar lijkt erg lang. Alleen als de productie een grote oplage heeft, zijn er statistisch gezien altijd gebruikers die er al veel eerder last van krijgen.

Naast de setup- en holdtijd moet er ook voor worden gezorgd dat de klokpuls voldoende breed is. Bovendien reageert een flipflop nooit direct. Er is altijd een bepaalde propagatietijd $t_{\text{p}}$. Deze kan anders zijn bij het hoog worden ($t_{\text{plh}}$) en bij het laag worden ($t_{\text{phl}}$).

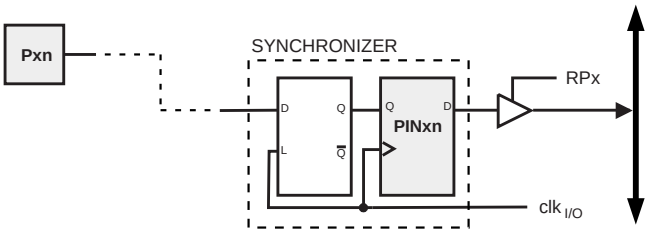
De kans dat het misgaat is te berekenen en hangt af van de bouw van de flipflop, van de klokfrequentie en van de frequentie waarmee het ingangssignaal verandert. Uit deze kans kan de tijd worden berekend tussen de twee momenten dat het misgaat. Deze tijd noemt men de MTBF (*mean time between failure*).




**Figuur D.15 :** **Het tijdsgedrag van een D-flipflop bij de actieve klokflank.** In het linker tijdsdiagram voldoet het ingangssignaal `D` aan de setup- en holdtijd. Het uitgangssignaal verandert na een propagatietijd. In het rechter tijdsdiagram voldoet het ingangssignaal `D` niet aan de setuptijd. Het uitgangssignaal komt in een ongedefinieerde metastabiele toestand.

Bij asynchrone ingangssignalen treedt het metastabiliteitsprobleem altijd op. Een oplossing hiervoor is om voor de flipflop een extra flipflop te plaatsen. Als de eerste flipflop metastabiel wordt, is de tweede flipflop nog steeds stabiel.

De generiek IO van de ATmega32, zie figuur 11.3, gebruikt een extra D-latch voor de synchronisatie. Deze latch, zie figuur D.16, staat voor flipflop `PINxn`.
Als de klok laag is, is de ingang van de D-latch niet doorverbonden. De latch houdt de huidige waarde vast en de uitgang van de D-latch is stabiel. Als de klok hoog wordt, leest de D-flipflop deze stabiele waarde. De flipflop wordt zo nooit metastabiel.

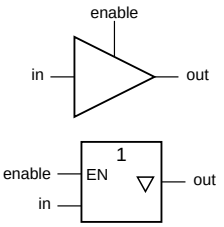



**Figuur D.16 :** **De synchronizer bij de generieke IO van de ATmega32.**
Tussen de aansluitpin `Pxn` en de D-flipflop `PINxn` is de extra D-latch aangebracht om het ingangssignaal te synchroniseren.

## D.6 De tristatebuffer en de tristate-inverter

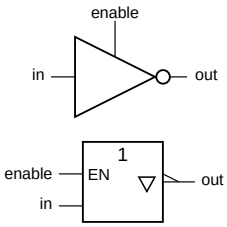
Een tristatebuffer of *three state buffer* heeft twee ingangen en een uitgang. De waarheidstabel, het Amerikaanse en het Europese symbool voor de tristatebuffer staan in figuur D.17.
De uitgang kent drie toestanden. Naast hoog (`1`) en laag (`0`) kan de uitgang ook hoogimpedant (`Z`) zijn. Als de ingang `enable` hoog is, werkt de buffer als een normale buffer: is de ingang `in` hoog, dan is de uitgang ook hoog en is `in` laag, dan is de uitgang ook laag. Als de ingang `enable` laag is, is de uitgang hoogimpedant. Dit betekent dat de uitgang niet meer aangestuurd wordt. Door lekstromen zal de uitgang laag worden.
Een tristate-inverter is exact hetzelfde als een tristatebuffer alleen inverteert deze het ingangssignaal. In figuur D.18 staan de symbolen en de waarheidstabel van de tristate-inverter.

| enable | in | out |
|---|---|---|
| 0 | – | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



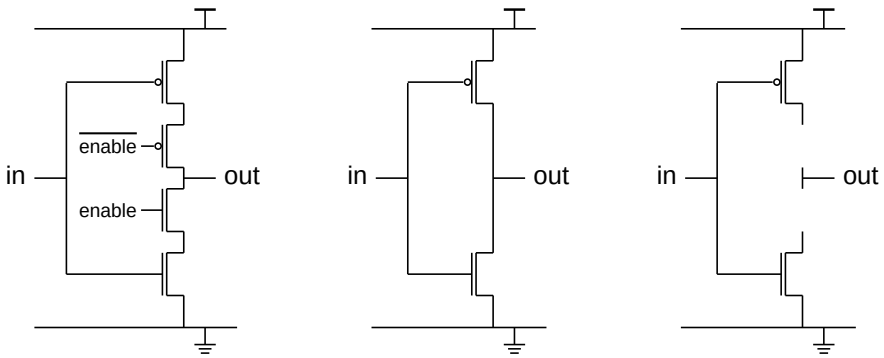

**Figuur D.17 :** **De waarheidstabel en de symbolen voor de tristate buffer.**
Links staat de waarheidstabel, rechtsboven het Amerikaanse en rechtsonder het IEC-symbool.

| enable | in | out |
|---|---|---|
| 0 | – | Z |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




**Figuur D.18 :** **De waarheidstabel en de symbolen voor de tristate inverter.**
Links staat de waarheidstabel, rechtsboven het Amerikaanse en rechtsonder het IEC-symbool.

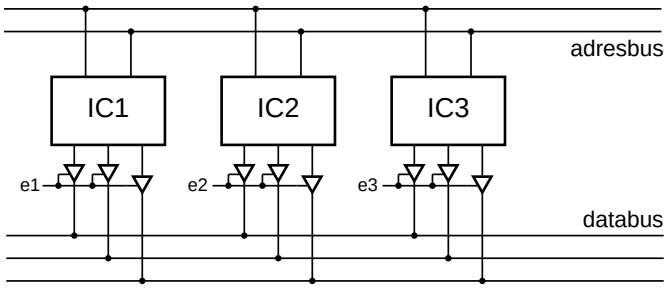
Het schema van de tristate-inverter uit figuur D.19 bestaat uit twee PMOS- en twee NMOS-transistoren. Het ingangssignaal `in` is aangesloten op de buitenste NMOS en PMOS-transistor. Het enable-signaal is aangesloten op de binnenste NMOS-transistor en het geïnverteerde enable-signaal op de binnenste PMOS. De inverter om `enable` te inverteren is niet getekend. De tristate-inverter bestaat dus uit zes transistoren.




**Figuur D.19 :** **De werking van een tristate-inverter.** Links is de tristate-inverter getekend, die bestaat uit twee PMOS- en twee NMOS-transistoren.
In het midden staat de situatie als het enable-signaal hoog is. De binnenste twee transistoren zijn geleidend. De tristate-inverter functioneert dan als een gewone inverter.
Rechts staat de situatie als het enable-signaal laag is. De binnenste twee transistoren geleiden niet. De uitgang is dan losgekoppeld van de buitenste transistoren. De tristate-inverter stuurt de uitgang niet meer aan.

Figuur D.19 toont de situaties als `enable` hoog en als `enable` laag is. Als `enable` hoog is, werkt de tristate-inverter als gewone inverter en als `enable` laag is, is de uitgang losgekoppeld.

De tristate-inverter en -buffer zijn heel belangrijke componenten. Hiermee is het mogelijk om met verschillende componenten naar één datalijn te schrijven. De generieke IO van de ATmega32 uit figuur 11.3 bevat bijvoorbeeld vier tristate-buffers.
Tristatebuffers worden veel gebruikt bij busstructuren. De IC's in figuur D.20 kunnen alle drie gelijktijdig 'luisteren' naar wat er — bijvoorbeeld door de adres-decoder van de microprocessor — op de adresbus is gezet. Daarentegen kunnen de drie IC's niet allemaal tegelijkertijd gegevens op de databus zetten. Met de

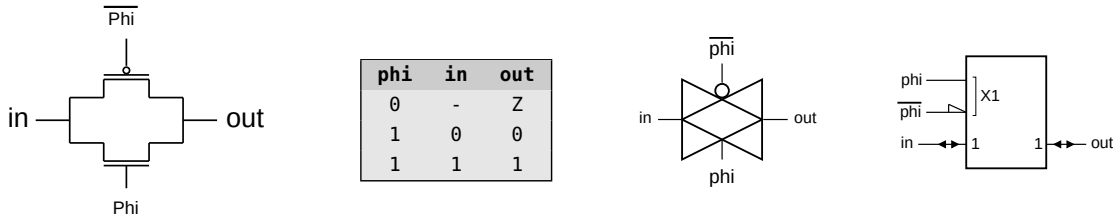



**Figuur D.20 : De toepassing van een tristatebuffer.** Alle drie IC's luisteren gelijktijdig naar de adresbus. De IC's zetten alleen informatie op de databus als het betreffende enable-signaal hoog is.

enable-signalen `e1`, `e2` en `e3` wordt geregeld wie er aan de beurt is. Er mag hooguit één van deze signalen actief zijn. Als bijvoorbeeld `e2` hoog is, zet IC2 zijn gegevens op de databus.

## D.7 De transmissiepoort

Een transmissiepoort (*transmission gate*) bestaat uit een NMOS- en een PMOS-transistor. Figuur D.21 geeft de schakeling, de waarheidstabel en twee symbolen.

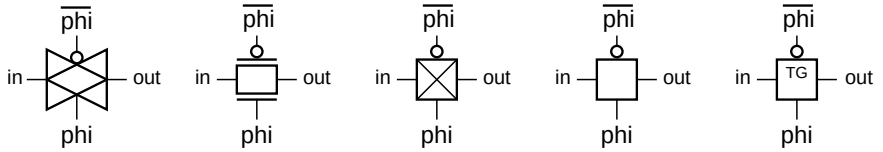



**Figuur D.21 : De transmissiepoort.**
Rechts staat de transmissiepoort. Deze bestaat uit een PMOS- en een NMOS-transistor. In het midden staat de waarheidstabel. Rechts staat het meest gangbare symbool en het IEC-symbool.

De transmissiepoort is een soort schakelaar. Indien het signaal `phi` laag is, sperren de beide transistoren. Als `phi` hoog is, geleidt de NMOS-, de PMOS-transistor of beide transistoren. De NMOS geleidt maximaal als de uitgang laag is en de PMOS geleidt maximaal als de uitgang hoog is.

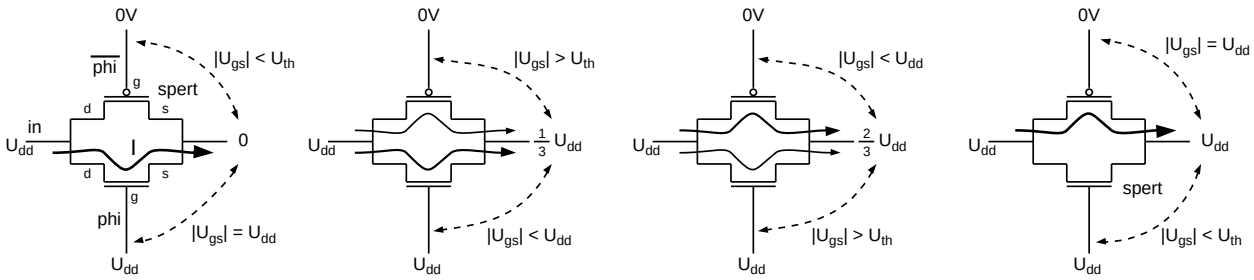
De transmissiepoort heeft geen richting. Beide zijde kunnen ingang en uitgang zijn. In de voorbeelden is de linker kant steeds hoog en loopt de stroom van links naar rechts. Als de linkerzijde laag is en de rechterzijde loopt de stroom van rechts naar links. Het IEC-symbool geeft dit bidirectionele gedrag duidelijk aan.

Er worden heel veel verschillende symbolen voor een transmissiepoort gebruikt. In figuur D.22 staan er een aantal.




**Figuur D.22 : Een aantal voorbeelden van symbolen die voor een transmissiepoort worden gebruikt.**

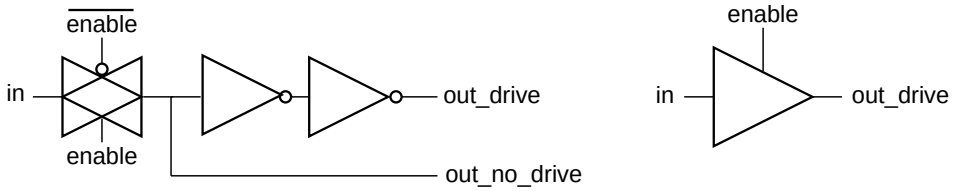
Figuur D.23 demonstreert de situatie dat de `phi` van laag naar hoog gaat terwijl de ingang hoog is. Aanvankelijk is de spanning $U_{gs}$ tussen de gate en de source van de NMOS-transistor gelijk aan de voedingsspanning $U_{dd}$. De stroom door de NMOS is dan maximaal. De grootte van de $U_{gs}$ van de PMOS is aanvankelijk nul; de PMOS spert. Naarmate de uitgangsspanning hoger wordt, wordt $|U_{gs}|$ van de NMOS kleiner en die van de PMOS groter. Uiteindelijk spert de NMOS-transistor en geleidt de PMOS maximaal en wordt de uitgang hoog ($U_{dd}$).




**Figuur D.23** : **De werking van de transmissiepoort.** De ingang `in` is hoog en `phi` wordt hoog. Aanvankelijk geleidt de NMOS-transistor maximaal en spert de PMOS. Naarmate de uitgangsspanning oploopt, gaat de NMOS steeds meer sperren en de PMOS beter geleiden. Uiteindelijk wordt de uitgang hoog ($U_{dd}$).

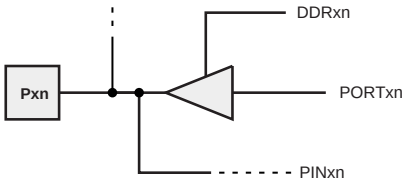
Met transmissiepoorten kunnen allerlei logische functies gerealiseerd worden. In figuur D.7 staat een multiplexer die opgebouwd is uit twee transmissiepoorten en een inverter. Figuur D.10 en figuur D.12 geven respectievelijke een D-latch en een D-flipflop met transmissiepoorten.
De kracht van componenten op basis van transmissiepoorten is dat er veel minder transistoren nodig zijn. Een nadeel is dat het elektrisch gedrag lastig is. De transmissiepoort is niet aangesloten op de voeding. Dus heeft de poort geen drijvende kracht. Het is niet meer dan een schakelaar, die ook nog een niet te verwaarlozen weerstand heeft en bovendien bidirectioneel is.

Functioneel is het gedrag van de transmissiepoort gelijk aan het gedrag van de tristatebuffer. De tristatebuffer is wel aangesloten op de voeding en de uitgang heeft dus een drijvende kracht. Het is verstandig om direct achter een transmissiepoort altijd een logische poort te plaatsen.

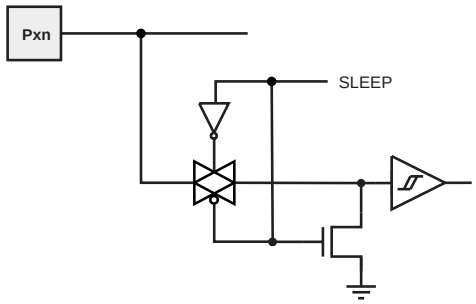



**Figuur D.24** : **De transmissiepoort met twee inverters en een tristatebuffer.** De uitgang `out_no_drive` van de transmissiepoort heeft geen drijvende kracht. De uitgang van de tristatebuffer heeft deze kracht wel.

De twee inverters in figuur D.24 achter de transmissiepoort zorgen er voor dat de drijvende kracht (*drive*) van de uitgang `out_drive` wel goed is.
Figuur D.25 toont de tristatebuffer bij de aansluitpin van de generieke IO van de ATmega32. Hier is een tristatebuffer nodig, want de stroom die geleverd moet




**Figuur D.25 :** **De tristatebuffer bij de aansluitpin van de ATmega32.**

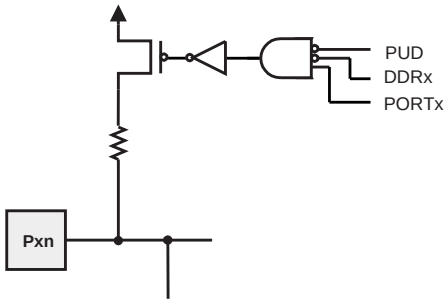



**Figuur D.26 :** **De transmissiepoort bij de generieke IO van de ATmega32.**

worden kan groot zijn. Figuur D.26 laat de ingangstak zien met de transmissiepoort, die aangesloten is op het signaal SLEEP. Achter de transmissiepoort zit een schmitttrigger, die de drijvende kracht levert. Daarom kan hier een transmissiepoort gebruikt worden.

## D.8 De pulluptransistor en de pulldowntransistor

Een PMOS-transistor wordt gebruikt om een datalijn hoog te maken. Een weerstand van enkele kΩ's maakt de drijvende kracht klein, zodat een andere component de spanning op de datalijn laag kan maken. Een PMOS-transistor, die op deze manier wordt gebruikt, wordt een pulluptransistor genoemd. Op dezelfde manier kan een NMOS-transistor gebruikt worden om een datalijn laag te maken.
Figuur D.27 geeft een detail van de generieke IO bij de ATmega32 met de pulluptransistor bij de aansluitpin. Als het signaal PORTxn hoog is en de signalen PUD en DDRxn laag zijn, is de gate van de pulluptransistor laag en geleidt de PMOS-transistor. De aansluitpin wordt dan hoog, mits er geen andere aansluiting is die het signaal omlaag trekt.

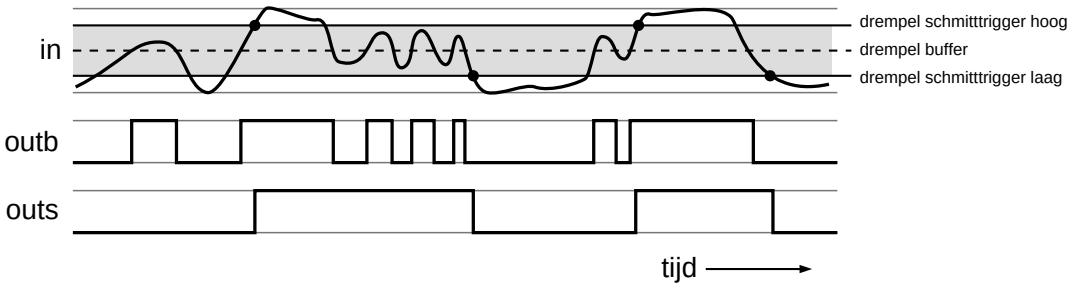



**Figuur D.27 :** **De pulluptransistor bij de aansluitpin van de ATmega32.** Als PORTxn hoog is en PUD en DDRxn laag zijn, is de gate van de pulluptransistor laag en geleidt de transistor. De spanning van de aansluitpin wordt in dat geval omhoog getrokken.

Het detail van de ingang van de ATmega32 uit figuur D.26 bevat een pulldowntransistor, die het signaal voor de schmitttrigger laag kan maken.

## D.9 De schmitttrigger

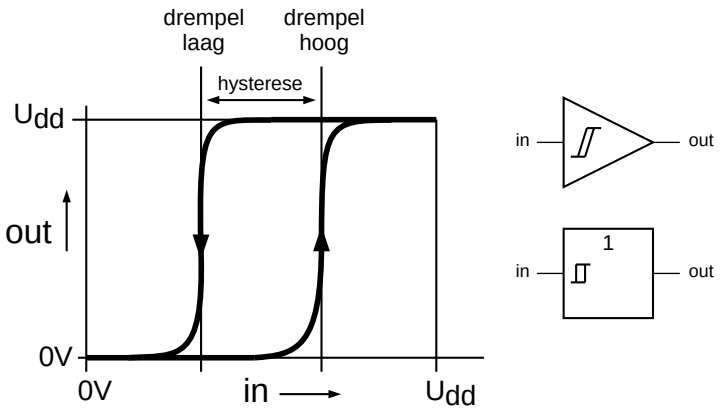
De schmitttrigger is een buffer of inverter met twee drempelwaarden. Als de ingang boven een bepaalde hoge drempelwaarde komt, wordt de uitgang hoog. Als de ingang beneden een andere lagere drempelwaarde komt, wordt de uitgang laag. Tussen de twee drempelwaarden behoudt de uitgang zijn waarde.




**Figuur D.28 : Het tijdsdiagram van een gewone buffer en een schmitttrigger.**
Het uitgangssignaal van de buffer verandert elke keer als het ingangssignaal de drempel van de buffer passeert. Het uitgangssignaal van de schmitttrigger wordt hoog als het ingangssignaal boven de hoge drempel van de schmitttrigger uitkomt en wordt laag als het onder de lage drempel van de schmitttrigger uitkomt.

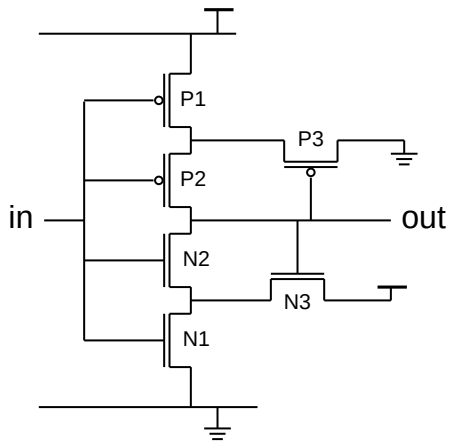
Het voordeel van de schmitttrigger ten opzichte van een gewone buffer of inverter is de grote stabiliteit en de lage ruisgevoeligheid. Een ingangssignaal met ruis veroorzaakt bij een gewone buffer meerdere overgangen. Bij een schmitttrigger is dat veel minder vaak het geval, zoals het tijdsdiagram van figuur D.28 laat zien. Alleen als het ingangssignaal boven de hoge drempelwaarde uitkomt of onder de lage drempelwaarde komt, verandert de uitgang.
Het gedrag met de twee drempelwaarden noemt men hysterese (*hysteresis*). In figuur D.29 is de uitgangsspanning als functie van de ingangsspanning getekend. In de symbolen voor de schmitttrigger staat een teken dat de hysterese representeert.

Hysterese komt ook in de natuur voor. Bijvoorbeeld bij faseovergangen en bij de magnetisatie van ferromagneten.




**Figuur D.29 : De hysterese van de schmitttrigger.**
Het uitgangssignaal van de schmitttrigger wordt hoog als het ingangssignaal boven de hoge drempel van de schmitttrigger uitkomt en wordt laag als het onder de lage drempel van de schmitttrigger uitkomt.

Er zijn vele manieren om een schmitttrigger te maken, bijvoorbeeld met een operationele versterker of met een paar transistoren. In ieder geval is elke implementatie van een schmitttrigger gebaseerd op positieve terugkoppeling. In figuur D.30 staat een CMOS-implementatie.




**Figuur D.30 : Een schmitttrigger-inverter in de CMOS-technologie.** De NMOS- en PMOS-transistor N3 en P3 zorgen voor de positieve terugkoppeling.

Als de ingang laag is, geleiden de transistoren P1 een P2 en sperren de transistoren N1, N2 en P3. Omdat de uitgang hoog is, geleidt N3.
Als de ingang hoger wordt, gaat eerst N1 geleiden. Deze transistor vormt samen met N3 een spanningsdeler, die er voor zorgt dat de source van N2 ergens halverwege de voedingsspanning ligt. Pas als de ingang boven deze drempelwaarde komt, gaat N2 geleiden. De uitgang wordt omlaag getrokken en N3 spert. Ook als de ingangsspanning weer omlaag gaat blijft de uitgang laag. Het effect van de spanningsdeler is verdwenen omdat N3 gesperd is.
Het omgekeerde effect doet zich voor als de ingang van hoog naar laag gaat bij de transistoren P1, P2 en P3.

# E

# RTTTL

In paragraaf 22.6 is als toepassing van PWM-signalen het afspelen van een stuk muziek gebruikt. Voor het vastleggen van het muziekstuk is het RTTTL-formaat gebruikt. Deze bijlage behandelt dit formaat en definieert een bibliotheek met routines om dit formaat te kunnen afspelen. De bibliotheek wordt in deze bijlage getest bij een pc-applicatie met behulp van de GNU C-compiler. In paragraaf 22.6 wordt de bibliotheek toegepast bij de ATmega32 met de AVR GNU C-compiler.

## E.1 Specificatie RTTTL

RTTTL staat voor *Ring Tones Text Transfer Language*. Het is een eenvoudig tekstgeoriënteerd formaat voor beltonen van mobiele telefoons. Dit formaat is geschikt voor homofone, *monophonic*, melodieën en niet geschikt voor polyfone, *polyphonic*, melodieën.

Een voorbeeld van een RTTTL-beltoon is:

```
FurElise:d=8,o=5,b=125:
32p,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,32p,c,e,a,4b.,
32p,e,g#,b,4c.6,32p,e,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,
32p,c,e,a,4b.,32p,d,c6,b,2a
```

De beltoon bestaat uit drie delen: een naam, de zogenoemde standaardparameters en het feitelijke lied. De drie delen worden gescheiden door een dubbele punt. Officieel mag de naam maximaal tien karakters hebben, maar veel beltonen hebben een langere naam. In het voorbeeld is de naam `FurElise`. Het deel met de standaardparameters bevat de waarden voor drie grootheden: de toonduur (`d`), de octaaf (`o`) en het aantal *beats* per minuut (`b`). Als deze parameters ontbreken worden deze standaardwaarden gebruikt: `d=4,o=6,b=63`. In het voorbeeld zijn dit de standaardwaarden voor de parameters: `d=8,o=5,b=125`.

**Tabel E.1 : Mogelijke waarden voor de parameter toonduur.**

| toonduur (*duration*) | betekenis |
|---|---|
| 1 | hele noot |
| 2 | halve noot |
| 4 | kwart noot |
| 8 | 1/8 noot |
| 16 | 1/16 noot |
| 32 | 1/32 noot |

De melodie bestaat uit een of meer noten, die gescheiden worden door komma's. De syntax van de RTTTL-noot is:

```
[toonduur] noot [speciale toonduur] [octaaf]
```

De toonduur, de speciale toonduur en de octaaf zijn optioneel. Als de toonduur en de octaaf ontbreken worden de standaardwaarden gebruikt.

Tabel E.1 bevat de mogelijke waarden voor de parameter toonduur. Een kwart noot ($d = 4$) duurt bij een vierkwartsmaat één tel. Een achtste noot ($d = 8$) is twee zo kort en duurt een halve tel. De parameter $b$ geeft het aantal beats per minuut en geeft het tempo aan. Voor een normaal tempo is $b$ gelijk aan 60 bpm.

Een temposlag (*beat*) duurt $60000/b$ ms. De tijdsduur van een noot hangt af van de toonduur $d$ en van het tempo $b$. Als $d$ groter is, betreft het een kortere noot en zal de tijdsduur korter zijn. Als $b$ groot is, is het tempo hoog en zullen de noten kort duren. Voor de tijdsduur van een noot, $t_{\text{ms}}$, geldt:

$$t_{\text{ms}} = \frac{4}{d}\frac{60000}{b} = \frac{240000}{d\,b} \tag{E.1}$$

De speciale toonduur is optioneel. Dit wordt aangegeven met een punt. Het verlengt de duur van de noot met de helft. De noot duurt dan anderhalve keer zo lang.

De specificatie gebruikt bovenkast voor de noten. De beltonen gebruiken daarentegen meestal onderkast.

Tabel E.2 geeft de symbolen voor de noten en hun muzikale betekenis. De frequentie van een noot hangt ook af van de octaaf. In de RTTTL-specificatie zijn geldige octaven 4, 5, 6 en 7. Als de noot en de octaaf bekend zijn, is de frequentie in de tabel te vinden.

**Tabel E.2 : De noten met de frequentie in Hz voor verschillende octaven.**

| noot (*note*) | | betekenis | octaaf (*scale*) | | | | |
|---|---|---|---|---|---|---|---|
| | | | **4** | **5** | **6** | **7** | **8** |
| C | c | c | 262 | 523 | 1047 | 2093 | 4186 |
| C# | c# | cis | 277 | 554 | 1109 | 2217 | 4435 |
| D | d | d | 294 | 587 | 1177 | 2349 | 4697 |
| D# | d# | dis | 311 | 622 | 1245 | 2489 | 4978 |
| E | e | e | 330 | 659 | 1319 | 2637 | 5274 |
| F | f | f | 349 | 698 | 1397 | 2794 | 5588 |
| F# | f# | fis | 367 | 740 | 1480 | 2960 | 5920 |
| G | g | g | 392 | 784 | 1568 | 3136 | 6272 |
| G# | g# | gis | 415 | 831 | 1661 | 3322 | 6645 |
| A | a | a | 440 | 880 | 1760 | 3520 | 7040 |
| A# | a# | ais | 466 | 932 | 1865 | 3729 | 7459 |
| B | b | b | 494 | 988 | 1976 | 3951 | 7902 |
| P | p | rust | | | | | |

De beltonen gebruiken de octaven 4, 5, 6 en 7. Sommige RTTTL-specificaties noemen als geldige octaven 5, 6, 7 en 8. Daarom is ook octaaf 8 in tabel E.2 vermeld.

In sommige specificaties wordt in plaats van noot B noot H gebruikt. Sommige landen gebruiken H in plaats van B. In de beltonen komt alleen noot B voor.

In het *Für Elise*-voorbeeld zijn de standaardparameters `d=8,o=5,b=125`. De standaardtoonduur $d$ is 8, de standaardoctaaf $o$ is 5 en het tempo $b$ is 125. Hier staat een vijftal voorbeelden van RTTTL-noten:

`d#6` Er staat geen toonduur, dus is de $d$ gelijk aan 8. De noot is de dis en de octaaf is 6. De frequentie kan worden opgezocht in tabel E.2 en is 1245 Hz. Uit formule E.1 volgt met $b$ is 125 en $d$ is 8 dat de tijdsduur 240 ms is.

`2a` De toonduur $d$ is gelijk aan 2. De noot is de a en er staat geen octaaf. De octaaf is dus 5. De frequentie is dus 880 Hz. Met $b$ is 125 en $d$ is 2 is de tijdsduur 960 ms.

`4b.` De toonduur $d$ is gelijk aan 4. De noot is de b en er staat geen octaaf. De octaaf is dus 5. De frequentie is 988 Hz. Met $b$ is 125 en $d$ is 4 is de tijdsduur 480 ms. De punt verlengt de tijdsduur met een factor anderhalf. De verlengde tijdsduur is 720 ms.

`e` Er staat geen toonduur, dus is $d$ gelijk aan 8. De noot is de e en er staat geen octaaf bij. De octaaf is dus 5. De frequentie is 659 Hz. Met $b$ 125 en $d$ 8 is de tijdsduur 240 ms.

`32p` De toonduur $d$ is gelijk aan 32. De noot is $p$. Het is een rust en de frequentie is niet relevant. Met $b$ 125 en $d$ 32 wordt de tijdsduur 60 ms.

## E.2 Bibliotheekroutines voor het lezen van RTTTL

De beltoon is een string die karakter voor karakter geëvalueerd moet worden. Dat kan met een pointer die steeds één positie opgeschoven wordt. Voor het afspelen van de beltoon is de naam niet interessant. Deze kan worden overgeslagen. Nadat de standaardparameters zijn gelezen, kunnen de noten worden afgespeeld. Het algoritme voor het afspelen van een beltoon is:

```
sla naam over
lees de standaardparameters
zolang er nog noten zijn
  lees de noot en bepaal de frequentie en tijdsduur
  laat de noot horen
```

Dit algoritme leest eerst de noot voor de frequentie en de tijdsduur en speelt de noot daarna af. Er is gekozen om dit niet in een handeling te doen. Het voordeel is dat de methode om de noot af te spelen afhangt van de omgeving. Bij de microcontroller gaat dat anders dan bij een pc-applicatie. Code E.1 geeft een opzet voor een functie `playRTTTL`, die een beltoon afspeelt.

**Code E.1 : Een opzet voor de functie playRTTTL.**

```
void playRTTTL(char *p)
{
  unsigned int f, ms;

  p = readRTTTLdefaults(p);
  while (*p != '\0') {
    p = readRTTTLnote(p, &f, &ms);
    speel noot met frequentie f en tijdsduur ms
  }
}
```

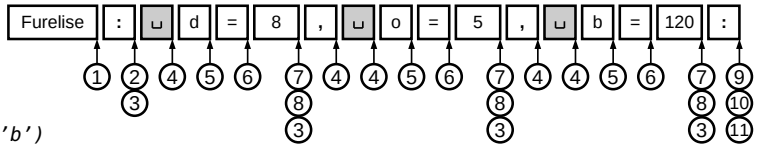
De functie `readRTTTLdefaults` skipt de naam, leest de standaardparameters en geeft een pointer naar de rest van de tekst met de melodie terug. In de herhalingslus leest de functie `readRTTTLnote` de eerstvolgende noot, bepaalt daaruit de frequentie `f` en de tijdsduur `ms`. Op een nog nader te definiëren wijze wordt de noot afgespeeld. De functie `readRTTTLnote` geeft een pointer naar de rest van de melodie terug. Alle noten zijn gelezen als de pointer naar de end-of-string wijst.

Naast de functies `readRTTTLdefaults` en `readRTTTLnote` bevat de `RTTTL`-bibliotheek vier globale variabelen — zie daarvoor ook het codefragment van code E.2 — voor de standaardtoonduur, de standaardoctaaf en het tempo en een standaardtijdsduur in milliseconden `defaultms`. Uit de standaardtoonduur en het tempo bepaalt `readRTTTLdefaults` de standaardtijdsduur. De functie `readRTTTLnote` gebruikt deze variabele om de juiste tijdsduur te berekenen.

**Code E.2 : Fragment van de `RTTTL`-bibliotheek met de globale variabelen.**

```
23 #define DEFAULT_DURATION   4
24 #define DEFAULT_OCTAVE     6
25 #define DEFAULT_BPM        63
26
27 unsigned int  defaultduration = DEFAULT_DURATION;
28 unsigned int  defaultoctave   = DEFAULT_OCTAVE;
29 unsigned int  defaultbpm      = DEFAULT_BPM;
30 unsigned int  defaultms;
```

```
1  skip naam
2  skip (eerste) dubbele punt
3  zolang nog geen (tweede) dubbele punt
4    skip komma en eventuele spaties
5    lees en onthoud parameter ('d','o','b')
6    skip gelijkteken
7    lees decimale waarde
8    ken waarde toe aan de bij parameter behorende standaardparameter
9  sla : over
10 bereken de standaardtijdsduur in ms
11 geef pointer naar rest van de string terug
```




**Figuur E.1 : Het lezen van de standaardparameters.** Links staat het algoritme en rechts staat een voorbeeld van een te lezen RTTTL-string. De nummers komen overeen met de betreffende regels uit het algoritme.

In figuur E.1 staat het algoritme voor de functie `readRTTTLdefaults`. Deze functie leest het begin van de beltoon met de standaardwaarden. Voor *Für Elise* is dat de tekst `Furelise:d=8,o=5,b=15:`. Dit voorbeeld is aan figuur E.1 toegevoegd. Het lezen van de parameters wordt gedaan met een herhalingslus. Elke keer wordt het karakter van de parameter onthouden, nadat de decimale waarde van de parameter is gelezen wordt deze waarde toegekend aan de betreffende standaardparameter. De volgorde waarin de parameters in de RTTTL-string staan, doet er bij deze methode niet toe.

**Code E.3 : Fragment van de `RTTTL`-bibliotheek met macro's voor het lezen.**

```
#if ( defined(__AVR__) && defined(__PGMSPACE_H_) )                    // flash
  #define  readRTTTLtoken(p)       (pgm_read_byte(p))
  #define  readnextRTTTLtoken(p)   (pgm_read_byte(++p))
#elif ( defined(__AVR__) && defined(_AVR_EEPROM_H_) )                 // EEPROM
  #define  readRTTTLtoken(p)       (eeprom_read_byte((const uint8_t *)p))
  #define  readnextRTTTLtoken(p)   (eeprom_read_byte((const uint8_t *)++p))
#else                                                                 // RAM
  #define  readRTTTLtoken(p)       (*p)
  #define  readnextRTTTLtoken(p)   (*++p)
#endif

char *readRTTTLdefaults(char *p);
char *readRTTTLnote(char *p, unsigned int *freq, unsigned int *ms);
```

De bibliotheek is zo gemaakt dat deze geschikt voor zowel een pc-applicatie als voor een applicatie voor de AVR-microcontroller. De twee bibliotheekroutines scannen de RTTTL-string en gebruiken hiervoor een pointer. Bij een ATmega32 gaat het lezen van een waarde anders als de RTTTL-string in RAM, flash of EEPROM staat. Daarom zijn er twee macro's `readRTTTLtoken` en `readnextRTTTLtoken` gedefinieerd. De eerste macro leest een waarde uit het geheugen. De tweede macro schuift de pointer eerst op en leest dan de waarde uit het geheugen. In het headerbestand `rtttl_lib.h` uit code E.3 staan de macrodefinities. Deze zijn verschillend voor flash, EEPROM en RAM. Bij een microcontroller-applicatie met de RTTTL-string in flash moet voor de aanroep van `rtttl_lib.h` het header-

bestand `pgmspace.h` ingesloten zijn en bij een microcontroller-applicatie met de RTTTL-string in EEPROM moet voor de aanroep van `rtttl_lib.h` het header-bestand `eeprom.h` ingesloten zijn. Zonder de aanroep van deze headerbestanden komt de string in RAM te staan. Bij een pc-applicatie staat RTTTL-string altijd in RAM.
In het headerbestand `rtttl_lib.h` staan verder nog de prototypen van de functies `readRTTTLdefaults` en `readRTTTLnote`.

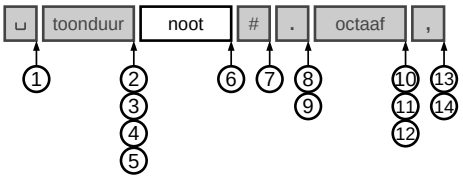
De uitwerking van het algoritme van de functie `readRTTTLdefaults` uit figuur E.1 staat in code E.4. De standaardtijdsduur `defaultms` wordt op regel 68 met formule E.1 berekend.

**Code E.4 : De functie `readRTTTLdefaults` uit de `RTTTL`-bibliotheek.**

```
32  char *readRTTTLdefaults(char *p)
33  {
34    char c, parameter;
35    unsigned int value;
36
37    while ( (c = readRTTTLtoken(p)) != ':' ) {         // skip name
38      p++;
39      if (c == '\0') return p;
40    }
41    c = readnextRTTTLtoken(p);                         // skip semicolon
42
43    while ( c != ':' ) {                               // read defaults
44      if ( isspace(c) || (c == ',') ) {                    // skip space or comma
45        c = readnextRTTTLtoken(p);                         // and continue
46        continue;
47      }
48      if (c == '\0') return p;                             // quit if end-of-string
49
50      parameter = c;                                       // remember parameter
51      p++;                                                 // skip parameter
52      c = readnextRTTTLtoken(p);                           // skip = and read next token
53
54      value = 0;
55      while (isdigit(c)) {                                 // read decimal value
56        value = value * 10 + (c - '0');
57        c = readnextRTTTLtoken(p);
58      }
59
60      switch (parameter) {                                 // assign value to default parameter
61        case 'd':  defaultduration = value; break;
62        case 'o':  defaultoctave   = value; break;
63        case 'b':  defaultbpm      = value; break;
64      }
65    }
66    p++;                                               // skip semicolon
67
68    defaultms = 240000/defaultduration/defaultbpm;     // calculate default ms
69
70    return (p);                                        // return pointer to rest of string
71  }
```

De functie `readRTTTLnote` leest een noot uit de RTTTL-string. In figuur E.2 staat het algoritme en de structuur van een RTTTL-noot. Achtereenvolgens leest en verwerkt de functie — mits aanwezig — de toonduur, de noot, de speciale toonduur en de octaaf.

```
1  skip eventuele spaties
2  tijdsduur is standaardtijdsduur
3  als toonduur aanwezig
4    lees toonduur
5    herbereken tijdsduur
6  lees noot
7  als # verhoog noot
8  als speciale toonduur aanwezig
9    verleng tijdsduur
10 als octaaf aanwezig
11   herbereken octaaf
12 bereken frequentie
13 skip komma
14 geef pointer naar rest van de string terug
```




**Figuur E.2** : **Het lezen van een RTTTL-noot.** Links staat het algoritme en rechts staat het formaat van een te lezen noot. De grijs gekleurde objecten zijn optioneel. De nummers komen overeen met de betreffende regels uit het algoritme.

De herberekening van de tijdsduur $t_{\text{ms}}$ volgt uit de standaardtijdsduur $t_{\text{ms,std}}$, de standaardtoonduur $d_{\text{std}}$ en de toonduur $d$:

$$t_{\text{ms}} = t_{\text{ms,std}} \frac{d_{\text{std}}}{d} \tag{E.2}$$

Een kortere toon heeft een hogere $d$ en duurt dus korter.

Een optie is om de frequenties van de noten in een grote opzoektabel te plaatsen. Per octaaf zijn daar twaalf integers voor nodig. In de hier gekozen oplossing staan alleen de frequenties van octaaf 8 in een array. De andere frequenties worden berekend uit de noot en de octaaf. Iedere lagere octaaf halveert de frequentie.
De frequentie $f$ van een noot hangt af van de frequentie van de noot in octaaf 8, $f_8$ en de octaaf $o$:

$$f = \frac{f_8}{2^{(8-o)}} \tag{E.3}$$

Bij een lagere octaaf hoort een kleinere $o$ en een lagere frequentie.

De uitwerking van het algoritme van de functie `readRTTTLnote` uit figuur E.2 staat in code E.5. De functie heeft drie ingangsparameters: een pointer naar de string met de te lezen noot en twee pointers naar de integers voor de frequentie `freq` en de tijdsduur `ms`.

De functie `readRTTTTLnote` gebruikt de standaardtoonduur $t_{\text{ms,std}}$ of berekent op regel 91, als de beschrijving van de noot een toonduur bevat, met formule E.2 de nieuwe tijdsduur $t_{\text{ms}}$. Op regel 110 wordt, als de speciale toonduur is gegeven, de tijdsduur anderhalf keer zo lang. De schuifoperator `>>` deelt de tijdsduur door 2 en met `+=` wordt dit resultaat bij de tijdsduur opgeteld.

**Code E.5 : De functie `readRTTTLnote` uit de `RTTTL`-bibliotheek.**

```
73  char *readRTTTLnote(char *p, unsigned int *freq, unsigned int *ms)
74  {
75    char c;
76    int i = -1;
77    unsigned int octave   = defaultoctave;
78    unsigned int duration = defaultduration;
79
80    c = readRTTTLtoken(p);
81    while ( isspace(c) ) c = readnextRTTTLtoken(p);    // skip white space
82    if ( c == '\0' ) return p;                         // quit if empty string
83
84    *ms = defaultms;                                   // time span is default time span
85    if ( isdigit(c) ) {                                // read duration if available
86      duration = 0;
87      while (isdigit(c)) {                                  // scan digits
88        duration = duration * 10 + (c - '0');
89        c = readnextRTTTLtoken(p);
90      }
91      *ms = defaultms * defaultduration / duration;         // recalculate time span
92    }
93
94    switch (c) {                                       // assign natural note
95      case 'C': case 'c':  i =  0; break;
96      case 'D': case 'd':  i =  2; break;
97      case 'E': case 'e':  i =  4; break;
98      case 'F': case 'f':  i =  5; break;
99      case 'G': case 'g':  i =  7; break;
100     case 'A': case 'a':  i =  9; break;
101     case 'B': case 'b':  i = 11; break;
102     case 'H': case 'h':  i = 11; break;
103     case 'P': case 'p':  i = -1; break;
104   }
105   if ( (c = readnextRTTTLtoken(p)) == '#') {         // next note if sharp note
106     i++;
107     c = readnextRTTTLtoken(p);
108   }
109
110   if (c == '.') {                                    // stretch time span if special duration
111     *ms += (*ms >> 1);
112     c = readnextRTTTLtoken(p);
113   }
114
115   if (isdigit(c) ) {                                 // read octave if available
116     octave = (c - '0');
117     c = readnextRTTTLtoken(p);
118   }
119
120   *freq = 0;                                         // calculate frequency
121   if (i >= 0) *freq = notes[i] >> (8 - octave);
122
123   if ( c == ',' )  p++;                              // skip comma
124
125   return (p);                                        // return pointer to rest of string
126 }
```

Code E.6 toont het fragment uit de RTTTL-bibliotheek met de definities van de noten en de declaratie van de array notes met de noten uit octaaf 8. De noten van de andere octaven worden hieruit berekend. Er is gekozen om octaaf 8 te gebruiken, omdat octaaf 8 in sommige RTTTL-specificaties ook een geldige octaaf is.

**Code E.6 : Het fragment van RTTTL-bibliotheek met definities van de noten.**

```
 8  #define C8    4186
 9  #define CIS8  4434
10  #define D8    4698
11  #define DIS8  4978
12  #define E8    5274
13  #define F8    5587
14  #define FIS8  5919
15  #define G8    6271
16  #define GIS8  6644
17  #define A8    7040
18  #define AIS8  7458
19  #define B8    7902
20
21  unsigned int notes[12] = {C8,CIS8,D8,DIS8,E8,F8,FIS8,G8,GIS8,A8,AIS8,B8};
```

Het switch-statement op regel 94 uit code E.5 bepaalt de index i van de noot. Regel 105 test of er een '#' achter de noot staat. Als dat het geval is, is het een verhoogde noot en wordt de index i met één opgehoogd.

Als de noot een octaaf bevat, wordt deze vanaf regel 115 gelezen. Regel 120 berekent met formule E.3 de frequentie van de noot.. In plaats van te delen door een macht van twee, gebruikt regel 120 de schuifoperator >>.

De RTTTL-bibliotheek bestaat uit het headerbestand rtttl_lib.h en het bestand rtttl_lib.c. Bij de ATmega32 is er nog extra header-bestand rtttl.h nodig met eventueel een include-opdracht naar pgmspace.h of naar eeprom.h. In code E.7 staat het begin van rtttl_lib.c. Als de definitie __AVR__ bestaat, wordt rtttl.h ook ingesloten.

**Code E.7 : Het begin van de RTTTL-bibliotheek rtttl_lib.c.**

```
1  #include <ctype.h>
2  #if defined(__AVR__)
3    #define __RTTTL_LIB__      1
4    #include "rtttl.h"
5  #endif
6  #include "rtttl_lib.h"
```

Het header-bestand rtttl_lib.h staat in code E.3 en het bestand rtttl_lib.c is de samenvoeging van achtereenvolgens de codes E.7, E.6, E.2, E.4 en E.5.

## E.3 Een pc-applicatie met RTTTL-bibliotheek

Een pc-applicatie om beltonen af te spelen staat in code E.9. De applicatie gebruikt de opzet van de functie playRTTTL uit code E.1. De zoemer of *buzzer* van de pc is bij de GNU C-compiler te benaderen met de functie Beep, die het header-bestand windows.h nodig heeft. De functie Beep genereert een toon en heeft twee

parameters: de frequentie en de tijdsduur in ms. Als de beltoon een rust bevat, moet er alleen gewacht worden. De functie `usleep`, die het headerbestand `unistd.h` nodig heeft, wacht 1 $\mu$s. De juiste wachttijd wordt verkregen door het aantal milliseconden met duizend te vermenigvuldigen.

**Code E.8 : De RTTTL-specificatie van het Wilhelmus.**

```
char Wilhelmus[] = "Wilhelmus:d=4,o=5,b=90:d,g,g,8a,8b,8c6,8a,b,8a,8b,  \
                    c6,b,8a,8g,a,2g,p,d,g,g,8a, 8b,8c6,8a,b,8a,8b,c6,b, \
                    8a,8g,a,2g,p,8b,8c6,2d6,e6,2d6,c6,b,8a,8b,c6,b,a,g, \
                    2a,p, d,8g,8f#,8g,8a,b,2a,g,f#,d,8e,8f#,g,g,f#,2g";
```

Het headerbestand `rtttl_lib.h`, uit code E.3, bevat de prototypen van de functies `readRTTTLdefaults` en `readRTTTLnote`. Het headerbestand `wilhelmus.h`, zie code E.8, bevat de string voor het Wilhelmus. De `\` aan het eind van de eerste drie regels zorgt ervoor dat de string doorloopt op de volgende regel.

**Code E.9 : Een GNU-applicatie om beltonen af te spelen.**

```
1  #include <windows.h>
2  #include <unistd.h>
3  #include "rtttl_lib.h"
4  #include "wilhelmus.h"
5
6  void playRTTTL(char *p)
7  {
8    unsigned int freq;
9    unsigned int ms;
10
11   p = readRTTTLdefaults(p);
12   while (readRTTTLtoken(p) != '\0') {
13     p = readRTTTLnote(p, &freq, &ms);
14     if (freq == 0) {
15       usleep(ms*1000);
16     } else {
17       Beep(freq, ms);
18     }
19   }
20 }
21
22 int main(void)
23 {
24    playRTTTL(Wilhelmus);
25
26    return 0;
27 }
```

De applicatie roept de functie `playRTTTL` met de string `wilhelmus` uit het headerbestand `wilhelmus,h` aan en speelt één keer de melodie van het Wilhelmus. In paragraaf 22 staat een applicatie voor de ATmega32, die eveneens gebruik maakt van de functies `readRTTTLdefaults` en `readRTTTLnote` uit de `RTTTL`-bibliotheek.

# F

# Headerbestanden

Deze bijlage bevat een overzicht van de belangrijkste headerbestanden, die horen bij de standaard C-bibliotheek. Van elk headerbestand worden de belangrijkste functies, macro's en typedefinities genoemd. Dit overzicht is zeker niet volledig. Sommige headerbestanden zijn uitgebreider en kennen meer functies die hier genoemd worden. Anderzijds zijn de meeste bestanden ook al besproken in de hoofdtekst van het boek. De toelichting op de verschillende functies, macro's en typedefinities is zeer summier. Dit hoofdstuk geeft alleen overzicht over welke mogelijkheden er standaard beschikbaar zijn.

**Tabel F.1 : De belangrijkste functies, macro's en typedefinities uit stdio.h.**

| **Functies** | |
|---|---|
| **toegang tot bestanden** | |
| `fclose` | sluit bestand |
| `feof` | test voor *end-of-file* |
| `fflush` | leegt uitvoerbuffer |
| `fopen` | opent bestand |
| `freopen` | heropent bestand |
| `setbuf` | stelt buffer in |
| `setvbuf` | verandert buffering |
| **geformatteerd schrijven** | |
| `scanf` | leest geformatteerd van `stdin` |
| `fscanf` | leest geformatteerd uit een bestand |
| `sscanf` | leest geformatteerd uit een string |
| `vscanf` | leest met een variabele argumentenlijst geformatteerd van `stdin` |
| `vscanf` | leest met een variabele argumentenlijst geformatteerd uit een bestand |
| `vfscanf` | leest met een variabele argumentenlijst geformatteerd uit een string |
| **geformatteerd lezen** | |
| `printf` | schrijft geformatteerd naar `stdin` |
| `fprintf` | schrijft geformatteerd naar een bestand |
| `sprintf` | schrijft geformatteerd naar een string |
| `vprintf` | schrijft met een variabele argumentenlijst geformatteerd naar `stdin` |
| `vsprintf` | schrijft met een variabele argumentenlijst geformatteerd naar een bestand |
| `vfprintf` | schrijft met een variabele argumentenlijst geformatteerd naar een string |

**Tabel F.1 (vervolg) : De belangrijkste functies, macro's en typedefinities uit stdio.h.**

| **Functies (vervolg)** | |
|---|---|
| **karakter/string uitvoer** | |
| `fgetc` | leest karakter uit bestand |
| `fgets` | leest string uit bestand |
| `getc` | leest karakter uit bestand |
| `getchar` | leest karakter van `stdin` |
| `getline` | leest string uit bestand |
| `gets` | leest string van `stdin` |
| `getw` | leest *word* (twee bytes) uit bestand |
| `ungetc` | zet laatst gelezen karakter terug |
| **karakter/string invoer** | |
| `fputc` | schrijft karakter naar bestand |
| `fputs` | schrijft string naar bestand |
| `putc` | schrijft karakter naar bestand |
| `putchar` | schrijft karakter naar `stdout` |
| `puts` | schrijft string naar `stdout` |
| `putw` | schrijft *word* (twee bytes) naar bestand |
| **directe in- en uitvoer** | |
| `fread` | leest blok met gegevens uit bestand |
| `fwrite` | schrijft blok met gegevens naar bestand |
| **bestandsposities** | |
| `fgetpos` | geeft huidige positie in bestand |
| `fseek` | zet filepointer naar bepaalde positie |
| `fsetpos` | zet filepointer naar bepaalde positie |
| `ftell` | geeft huidige positie in bestand |
| `rewind` | zet filepointer terug naar het begin bestand |
| **bestandsbewerkingen** | |
| `remove` | verwijdert bestand |
| `rename` | verandert naam van een bestand |
| `tmpfile` | opent tijdelijk bestand |
| `tmpnam` | creëert een nog niet gebruikte bestandsnaam |
| **foutafhandeling** | |
| `clearerr` | schoont de bestandsindicatoren foutmeldingen op |
| `ferror` | test of er lees- of schrijf fout was opgetreden |
| `perror` | drukt standaard foutmelding af |
| **Macro's** | |
| `EOF` | *end-of-file* |
| `FILENAME_MAX` | maximale lengte bestandsnamen |
| `NULL` | nullpointer |
| `TMP_MAX` | maximaal aantal tijdelijke bestanden |
| `stdin` | standaard invoer van toetsenbord |
| `stdout` | standaard uitvoer naar beeldscherm |
| `stderr` | standaard uitvoer voor foutmeldingen naar beeldscherm |
| **Typedefinities** | |
| `FILE` | datastructuur voor gegevens van bestanden |
| `fpos_t` | is een **`long`** voor positie in bestanden |
| `size_t` | is een **`unsigned long`** om afmetingen mee aan te geven |

**Tabel F.2 : De belangrijkste functies, macro's en typedefinities uit stdlib.h.**

| **Functies** | |
|---|---|
| **stringconversies** | |
| `atof` | converteert string naar een **`double`** |
| `atoi` | converteert string naar een **`int`** |
| `atol` | converteert string naar een **`long`** `integer` |
| `strtod` | converteert string naar een **`double`** |
| `strtol` | converteert string naar een **`long int`** |
| `strtoul` | converteert string naar een **`unsigned long int`** |
| **bewerkingen met integers** | |
| `abs` | bepaald absolute waarde van een **`int`** |
| `div` | geheeltallige deling met **`int`**'s |
| `labs` | bepaald absolute waarde van een **`long int`** |
| `ldiv` | geheeltallige deling met **`long int`**'s |
| **dynamisch geheugenbeheer** | |
| `calloc` | reserveert geheugenruimte voor een array |
| `malloc` | reserveert een blok geheugenruimte |
| `realloc` | verandert de hoeveelheid gereserveerde geheugenruimte |
| `free` | geeft eerder gereserveerde geheugenruimte vrij |
| **zoeken en sorteren** | |
| `bsearch` | doet een *binary search* in een array |
| `qsort` | sorteert de elementen van een array |
| **systeem en omgeving** | |
| `abort` | stopt huidige proces |
| `atexit` | stopt huidige proces |
| `exit` | beëindigt programma |
| `getenv` | haalt omgevingsvariabele op |
| `system` | voert systeemcommando uit |
| **Multibyte characters/strings** | |
| `mblen` | geeft lengte van een *multibyte character* |
| `mbtowc` | zet een *multibyte character* om in een *wide character* |
| `wctomc` | zet een *wide character* om in een *multibyte character* |
| `mbstowcs` | zet een *multibyte string* naar een *wide character string* |
| `wcstomb` | zet een *wide character string* naar een *multibyte string* |
| **Macro's** | |
| `EXIT_FAILURE` | beëindigingscode bij falen |
| `EXIT_SUCCES` | beëindigingscode bij succes |
| `MB_CUR_MAX` | maximale grootte van een *multibyte* |
| `NULL` | nullpointer |
| `RAND_MAX` | maximum waarde getal random generator |
| **Typedefinities** | |
| `div_t` | datastructuur met **`int`**'s quotiënt en rest |
| `ldiv_t` | datastructuur met **`long int`**'s quotiënt en rest |
| `size_t` | is een **`unsigned long`** om afmetingen mee aan te geven |

**Tabel F.3 : De belangrijkste functies en typedefinities uit stdarg.h.**

| **Functies** | |
|---|---|
| `va_start` | Opent variabele argumentenlijst |
| `va_arg` | Haalt eerstvolgende argument op |
| `va_end` | Sluit het gebruik van een argumentenlijst |
| **Macro's** | |
| `va_list` | is een structuur voor het gebruik van een variabele argumentenlijst |

**Tabel F.4 : De belangrijkste functies, macro's en typedefinities uit string.h.**

| **Functies** | |
|---|---|
| **kopiëren** | |
| `strcpy` | kopieert string |
| `strlcpy` | kopieert string tot n karakters en voegt `'\0'` toe |
| `strncpy` | kopieert string tot n karakters |
| **concatenatie** | |
| `strcat` | voegt string toe aan string |
| `strlcat` | voegt string toe aan string tot n karakters en voegt `'\0'` toe |
| `strncat` | voegt string toe aan string tot n karakters |
| **vergelijken** | |
| `strcmp` | vergelijkt twee strings |
| `strcoll` | vergelijkt twee strings met *locale* |
| `strncmp` | vergelijkt n karakters van twee strings |
| `strxfrm` | vertaalt string met gebruikmaking van locale |
| **zoeken** | |
| `strchr` | zoekt in een string van links af naar een karakter |
| `strcspn` | geeft lengte van string tot waar bepaalde karakters niet voorkomen |
| `strpbrk` | zoekt karakter in string |
| `strrchr` | zoekt in een string van rechts af naar een karakter |
| `strspn` | geeft lengte van string tot waar bepaalde karakters voorkomen |
| `strstr` | zoekt substring in string |
| `strtok` | splitst string in tekens |
| **omzetten** | |
| `strlwr` | zet string om naar onderkast (kleine letters) |
| `strupr` | zet string om naar bovenkast (hoofdletters) |
| **overig** | |
| `strerror` | geeft een foutmelding terug |
| `strlen` | bepaalt de lengte van de string zonder de `'\0'` |
| **geheugen** | |
| `memchr` | zoekt een karakter in een geheugenblok |
| `memcmp` | vergelijkt twee geheugenblokken |
| `memcpy` | kopieert een geheugenblok |
| `memmove` | verplaatst de inhoud een geheugenblok |
| `memset` | vult een deel van een geheugenblok met een bepaalde waarde |
| **Macro's** | |
| `NULL` | nullpointer |
| **Typedefinities** | |
| `size_t` | is een **`unsigned long`** om afmetingen mee aan te geven |

**Tabel F.5 : De belangrijkste functies, macro's en typedefinities uit stddef.h.**

| **Functies** | |
|---|---|
| `offsetof` | geeft de positie van een veld in een datastructuur |
| **Typedefinities** | |
| `ptrdiff_t` | is een **`long`** die het verschil tussen twee pointers weergeeft |
| `size_t` | is een **`unsigned long`** om afmetingen mee aan te geven |
| `wchar_t` | *wide character* (twee bytes groot) |
| **Macro's** | |
| `NULL` | nullpointer |

**Tabel F.6 : De belangrijkste functies uit ctype.h.**

| Functies | |
|---|---|
| **testfuncties** | |
| `isalnum` | test of karakter alfanumeriek is |
| `isalpha` | test of karakter een hoofd- of kleine letter is |
| `islower` | test of karakter een kleine letter is |
| `isupper` | test of karakter een hoofdletter is |
| `isdigit` | test of karakter een cijfer is |
| `isxdigit` | test of karakter een hexadecimaal cijfer is |
| `iscntrl` | test of karakter een control-karakter is |
| `ispunct` | test of karakter interpunctie is |
| `isgraph` | test of karakter *printable* en geen *white space* is |
| `isprint` | test of karakter *printable* is |
| `isspace` | test of karakter een *white space* is |
| `isblank` | test of karakter een spatie of een tab is |
| `isascii` | test of karakter een ASCII-waarde is |
| **conversie** | |
| `tolower` | converteert karakter naar een kleine letter |
| `toupper` | converteert karakter naar een hoofdletter |
| `toascii` | geeft de ASCII-waarde van karakter |

**Tabel F.7 : De belangrijkste macro's uit limits.h.**

| Macro's | |
|---|---|
| `CHAR_BIT` | aantal bits van een **`char`** |
| `SCHAR_MIN` | minimum waarde **`signed char`** |
| `SCHAR_MAX` | maximum waarde **`signed char`** |
| `UCHAR_MAX` | maximum waarde **`unsigned char`** |
| `CHAR_MIN` | minimum waarde `SCHAR_MIN` or `0` |
| `CHAR_MAX` | maximum waarde `SCHAR_MAX` or `UCHAR_MAX` |
| `MB_LEN_MAX` | maximum waarde aantal bytes van een *multibyte character* |
| `SHRT_MIN` | minimum waarde **`short int`** |
| `SHRT_MAX` | maximum waarde **`short int`** |
| `USHRT_MAX` | maximum waarde **`unsigned short int`** |
| `INT_MIN` | minimum waarde **`int`** |
| `INT_MAX` | maximum waarde **`int`** |
| `UINT_MAX` | maximum waarde **`unsigned short int`** |
| `LONG_MIN` | minimum waarde **`long int`** |
| `LONG_MAX` | maximum waarde **`long int`** |
| `ULONG_MAX` | maximum waarde **`unsigned int`** |

**Tabel F.8 : De belangrijkste macro's uit floats.h.**

| Macro's | |
|---|---|
| `FLT_RADIX` | 2 RADIX Base for all floating-point types |
| `FLT_MANT_DIG` | aantal cijfers van de mantissa van **`float`** |
| `DBL_MANT_DIG` | aantal cijfers van de mantissa van **`double`** |
| `LDBL_MANT_DIG` | aantal cijfers van de mantissa van **`long double`** |
| `FLT_DIG` | aantal significante cijfers bij **`float`** |
| `DBL_DIG` | aantal significante cijfers bij **`double`** |
| `LDBL_DIG` | aantal significante cijfers bij **`long double`** |
| `FLT_MIN_EXP` | kleinste exponent **`float`** |
| `DBL_MIN_EXP` | kleinste exponent **`double`** |
| `LDBL_MIN_EXP` | kleinste exponent **`long double`** |

**Tabel F.8 (vervolg) : De belangrijkste functies, macro's en typedefinities uit floats.h.**

| **Macro's (vervolg)** | |
|---|---|
| `FLT_MIN_10_EXP` | kleinste exponent **`float`** met grondtal 10 |
| `DBL_MIN_10_EXP` | kleinste exponent **`double`** met grondtal 10 |
| `LDBL_MIN_10_EXP` | kleinste exponent **`long double`** met grondtal 10 |
| `FLT_MAX_EXP` | grootste exponent **`float`** |
| `DBL_MAX_EXP` | grootste exponent **`double`** |
| `LDBL_MAX_EXP` | grootste exponent **`long double`** |
| `FLT_MAX_10_EXP` | grootste exponent **`float`** met grondtal 10 |
| `DBL_MAX_10_EXP` | grootste exponent **`double`** met grondtal 10 |
| `LDBL_MAX_10_EXP` | grootste exponent **`long double`** met grondtal 10 |
| `FLT_MAX` | grootste waarde **`float`** |
| `DBL_MAX` | grootste waarde **`double`** |
| `LDBL_MAX` | grootste waarde **`long double`** |
| `FLT_EPSILON` | verschil van `1` en de eerstvolgende **`float`** waarde |
| `DBL_EPSILON` | verschil van `1` en de eerstvolgende **`double`** waarde |
| `LDBL_EPSILON` | verschil van `1` en de eerstvolgende **`long double`** waarde |
| `FLT_MIN` | kleinste waarde **`float`** |
| `DBL_MIN` | kleinste waarde **`double`** |
| `LDBL_MIN` | kleinste waarde **`long double`** |
| `FLT_ROUNDS` | gebruikte afrondingsmethode |

**Tabel F.9 : De belangrijkste functies, macro's en typedefinities uit time.h.**

| **Functies** | |
|---|---|
| **tijdfuncties** | |
| `clock` | geeft het aantal klokslagen sinds het programma is gestart |
| `difftime` | geeft het verschil tussen twee tijden |
| `mktime` | zet de datastructuur `tm` in `time_t` |
| `time` | geeft de huidige tijd |
| **conversies** | |
| `asctime` | zet de structuur `tm` om naar een string |
| `ctime` | zet `time_t` om naar een string |
| `gmtime` | zet `time_t` om naar `tm` op basis van de UTC-tijd |
| `localtime` | zet `time_t` om naar `tm` op basis van de lokale tijd |
| `strftime` | zet de structuur `tm` geformatteerd om naar een string |
| **Macro's** | |
| `CLOCKS_PER_SEC` | klokslagen per seconde |
| `NULL` | nullpointer |
| **Typedefinities** | |
| `clock_t` | is structuur voor de klokslagen |
| `size_t` | is een **`unsigned long`** om afmetingen mee aan te geven |
| `time_t` | is een **`long int`** voor het vast leggen van de tijd |
| **`struct`** `tm` | `tm` is een datastructuur voor het vastleggen tijd en datum |

**Tabel F.10 : De belangrijkste functies uit locale.h.**

| **Functies** | |
|---|---|
| `setlocale` | definieert gebruikte formaten voor teksten en naamgeving |
| `localeconv` | haalt de huidige formaten op |
| **Typedefinities** | |
| `lconv` | is datastructuur voor teksten en namen |

**Tabel F.11 : De belangrijkste macro's en typedefinities uit errno.h.**

| **Macro's** | |
|---|---|
| `EDOM` | is de foutcode bij een domeinfout bij berekeningen |
| `ERANGE` | is de foutcode bij een fout in het bereikfout bij berekeningen |
| **Typedefinities** | |
| `errno` | is een **`int`** en bevat de foutcode bij het optreden van een fout |

**Tabel F.12 : De belangrijkste functies uit assert.h.**

| **Functies** | |
|---|---|
| `assert` | detecteert runtimefouten en geeft dan een foutmelding |

**Tabel F.13 : De belangrijkste functies uit math.h.**

| **Functies** | |
|---|---|
| **trigonometrische functies** | |
| `cos` | berekent cosinus |
| `sin` | berekent sinus |
| `tan` | berekent tangens |
| `acos` | berekent arccosinus |
| `asin` | berekent arcsinus |
| `atan` | berekent arctangens |
| `atan2` | berekent arctangens met twee parameters |
| **hyperbolische functies** | |
| `cosh` | berekent cosinus hyperbolicus |
| `sinh` | berekent sinus hyperbolicus |
| `tanh` | berekent tangens hyperbolicus |
| **exponentiële en logaritmische functies** | |
| `exp` | berekent de exponentiële functie |
| `frexp` | splitst een gebroken getal in een macht van twee en getal tussen `0` en `1` |
| `ldexp` | berekent getal uit een macht van twee en getal tussen `0` en `1` |
| `log` | berekent natuurlijke logaritme |
| `log10` | berekent logaritme met grondtal `10` |
| `modf` | splitst gebroken getal in een geheel getal en gebroken getal tussen `0` en `1` |
| **machten** | |
| `pow` | berekent de macht |
| `sqrt` | berekent de wortel |
| **afronden** | |
| `ceil` | rondt gebroken getal naar boven af |
| `fabs` | berekent de absolute waarde |
| `floor` | rondt gebroken getal naar beneden af |
| `fmod` | berekent de rest van deling bij gebroken getallen |
| `round` | rondt gebroken getal af |

# G

# Timer instellingen PWM

## G.1 De bits uit register TCCR0 van timer 0

Timer 0 gebruikt register TCCR0 voor het instellen, zie figuur G.1. Een overzicht van de belangrijkste bits uit TCCR0 staat in de tabellen G.1, G.2 en G.3.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 |

**Figuur G.1 : De bits van register TCCR0 voor het instellen van timer 0.**

**Tabel G.1 : De CS0-bits van timer 0.**

| CS02..0 | klokconfiguratie |
|---|---|
| 000 | Timer 0 is uit |
| 001 | clk |
| 010 | clk/8 |
| 011 | clk/64 |
| 100 | clk/256 |
| 101 | clk/1024 |
| 110 | Externe klok (stijgende flank) |
| 111 | Externe klok (dalende flank) |

**Tabel G.2 : De WGM0-bits van timer 0.**

| WGM01..0 | Mode | TOP | Update OCR0 | Update TOV0 |
|---|---|---|---|---|
| 00 | Normal | MAX | Immediate | MAX |
| 01 | Phase correct PWM | MAX | TOP | BOTTOM |
| 10 | CTC | OCR0 | Immediate | MAX |
| 11 | Fast PWM | MAX | TOP | MAX |

De maximale waarde MAX van de 8-bits teller is 0xFF. De waarde van BOTTOM is altijd 0x00 De teller telt, afhankelijk van de WGM0-bits van BOTTOM tot de waarde TOP.

**Tabel G.3 : De COM0-bits van timer 0.**

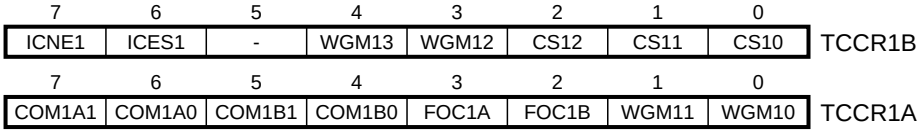
| Mode | COM01..0 | Beschrijving |
|---|---|---|
| **Normal, CTC** | 00 | OC0 niet aangesloten |
| | 01 | OC0 inverteren (toggelen) bij OCF0 |
| | 10 | OC0 laag maken bij OCF0 |
| | 11 | OC0 hoog maken bij OCF0 |
| **Phase correct PWM** | 00 | OC0 niet aangesloten |
| | 01 | niet gebruikt |
| | 10 | OC0 laag bij OCF0 tijdens optellen, OC0 hoog bij OCF0 tijdens aftellen |
| | 11 | OC0 hoog bij OCF0 tijdens optellen, OC0 laag bij OCF0 tijdens aftellen |
| **Fast PWM** | 00 | OC0 niet aangesloten |
| | 01 | niet gebruikt |
| | 10 | OC0 laag bij OCF0, OC0 hoog bij TOV0 |
| | 11 | OC0 hoog bij OCF0, OC0 laag bij TOV0 |

### Overige bits

Bit FOC0, *Force Output Compare 0*, forceert een directe actie van de waveformgenerator.

## G.2 De bits uit de registers TCCR1A en TCCR1B van timer 1

Timer 1 heeft meer mogelijkheden en gebruikt voor de instellingen twee registers: TCCR1A en TCCR1B.




**Figuur G.2 : De bits van register TCCR1A en TCCR1B voor het instellen van timer 1.**

De belangrijkste bits staan in de in de tabellen G.4, G.5 en G.6.

**Tabel G.4 : De CS1-bits van timer 1.**

| CS12..0 | klokconfiguratie |
|---|---|
| 000 | Timer 1 is uit |
| 001 | clk |
| 010 | clk/8 |
| 011 | clk/64 |
| 100 | clk/256 |
| 101 | clk/1024 |
| 110 | Externe klok (stijgende flank) |
| 111 | Externe klok (dalende flank) |

**Tabel G.5 : De WGM1-bits van timer 1.**

| Modusnummer | WGM13..0 | Mode | TOP | Update OCR1x | Update TOV1x | AVRstudio bug |
|---|---|---|---|---|---|---|
| 0 | 0000 | Normal | MAX | Immediate | MAX | werkt correct |
| 1 | 0001 | Phase correct PWM, 8-bit | 0x00FF | TOP | BOTTOM | werkt correct |
| 2 | 0010 | Phase correct PWM, 9-bit | 0x01FF | TOP | BOTTOM | werkt correct |
| 3 | 0011 | Phase correct PWM, 10-bit | 0x03FF | TOP | BOTTOM | werkt correct |
| 4 | 0100 | CTC | OCR1A | Immediate | MAX | werkt correct |
| 5 | 0101 | Fast PWM, 8-bit | 0x00FF | TOP | TOP | werkt als 1 |
| 6 | 0110 | Fast PWM, 9-bit | 0x01FF | TOP | TOP | werkt als 2 |
| 7 | 0111 | Fast PWM, 10-bit | 0x03FF | TOP | TOP | werkt als 3 |
| 8 | 1000 | Phase and freq. correct PWM | ICR1 | BOTTOM | BOTTOM | werkt als 0 |
| 9 | 1001 | Phase and freq. correct PWM | OCR1A | BOTTOM | BOTTOM | werkt als 1 |
| 10 | 1010 | Phase correct PWM | ICR1 | TOP | BOTTOM | werkt als 2 |
| 11 | 1011 | Phase correct PWM | OCR1A | TOP | BOTTOM | werkt als 3 |
| 12 | 1100 | CTC | ICR1 | Immediate | MAX | werkt als 4 |
| 13 | 1101 | niet gebruikt | | | | |
| 14 | 1110 | Fast PWM | ICR1 | TOP | TOP | werkt als 2 |
| 15 | 1111 | Fast PWM | OCR1A | TOP | TOP | werkt als 3 |

De maximale waarde MAX van de 16-bits teller is 0xFFFF. De waarde van BOTTOM is altijd 0x0000. De teller telt van BOTTOM tot de waarde TOP. De waarde van TOP hangt af van de WGM-bits.
De implementatie van timer 1 bij de simulator van AVRstudio is bij ATmega32 bij sommige modi niet correct. De laatste kolom van tabel G.5 geeft een overzicht.

Timer 1 heeft twee *compare*-uitgangen OC1A en OC1B met allebei een waveformgenerator met twee bits voor de instellingen. In het totaal dus vier COM-bits: COM1A1, COM1A0, COM1B1 en COM1B0. Met deze bits kunnen de *compare*-uitgangen OC1A en OC1B onafhankelijk van elkaar ingesteld worden. De betekenis van de bits van beide uitgangen is identiek. In tabel G.6 is dit aangegeven met een x, in plaats van een A of B, in de namen van de bits, de uitgang en de vlag.

**Tabel G.6 : De COM1A- en COM1B-bits van timer 1.**

| Mode | COM1x1..0 | Beschrijving |
|---|---|---|
| **Normal, CTC** | 00 | OC1x niet aangesloten |
| | 01 | OC1x toggelt bij OCF1x |
| | 10 | OC1x laag bij OCF1x |
| | 11 | OC1x hoog bij OCF1x |
| **Phase correct PWM** | 00 | OC1x niet aangesloten |
| | 01 | 9,11: OC1A toggelt bij OCF1A, OC1B niet aangesloten<br>anders: OC1x niet aangesloten |
| | 10 | OC1x laag bij OCF1x tijdens optellen,<br>OC1x hoog bij OCF1x tijdens aftellen |
| | 11 | OC1x hoog bij OCF1x tijdens optellen,<br>OC1x laag bij OCF1x tijdens aftellen |
| **Phase, freq. correct PWM** | 00 | OC1x niet aangesloten |
| | 01 | 9,11: OC1A toggelt bij OCF1A, OC1B niet aangesloten<br>anders: OC1x niet aangesloten |
| | 10 | OC1x laag bij OCF1x tijdens optellen,<br>OC1x hoog bij OCF1x tijdens aftellen |
| | 11 | OC1x hoog bij OCF1x tijdens optellen,<br>OC1x laag bij OCF1x tijdens aftellen |
| **Fast PWM** | 00 | OC1x niet aangesloten |
| | 01 | 15: OC1A toggelt bij OCF1A, OC1B niet aangesloten<br>anders: OC1x niet aangesloten |
| | 10 | OC1x laag bij OCF1x, OC1x hoog bij TOV1x |
| | 11 | OC1x hoog bij OCF1x, OC1x laag bij TOV1x |

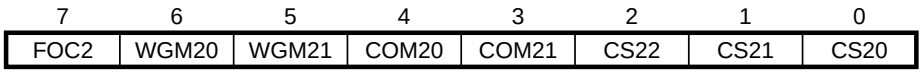
### Overige bits

Door bit FOC1A of FOC1B *Force Output Compare 1A/B* hoog te maken, voert de waveformgenerator van respectievelijk uitgang OC1A of OC1B de bewerking direct uit.

De bits ICNC1 en ICES1 worden gebruikt bij de *input capture* modus. ICNC1, *Input Capture Noise Canceler*, schakelt de ruisonderdrukking in. ICES1, *Input Capture Edge Select*, bepaalt de flankgevoeligheid van de *input capture*. Als dit bit hoog is, wordt de opgaande flank gebruikt, anders is deze ingang gevoelig voor de neergaande flank.

## G.3 De bits uit register TCCR2 van timer 2

Timer 2 gebruikt register TCCR2 voor het instellen, zie figuur G.3. Een overzicht van de belangrijkste bits uit TCCR2 staat in de tabellen G.7, G.8 en G.9.




**Figuur G.3 : De bits van register TCCR2 voor het instellen van timer 2.**

**Tabel G.7 : De CS2-bits van timer 2.**

| CS22..0 | klokconfiguratie |
|---|---|
| 000 | Timer 2 is uit |
| 001 | clk |
| 010 | clk/8 |
| 011 | clk/32 |
| 100 | clk/64 |
| 101 | clk/128 |
| 110 | clk/256 |
| 111 | clk/1024 |

**Tabel G.8 : De WGM2-bits van timer 2.**

| WGM21..0 | Mode | TOP | Update OCR2 | Update TOV2 |
|---|---|---|---|---|
| 00 | Normal | MAX | Immediate | MAX |
| 01 | Phase correct PWM | MAX | TOP | BOTTOM |
| 10 | CTC | OCR2 | Immediate | MAX |
| 11 | Fast PWM | MAX | TOP | MAX |

De maximale waarde MAX van de 8-bits teller is 0xFF. De waarde van BOTTOM is altijd 0x00 De teller telt, afhankelijk van de WGM2-bits van BOTTOM tot de waarde TOP.

**Tabel G.9 : De COM2-bits van timer 2.**

| Mode | COM21..0 | Beschrijving |
|---|---|---|
| **Normal, CTC** | 00 | OC2 niet aangesloten |
| | 01 | OC2 inverteren (toggelen) bij OCF2 |
| | 10 | OC2 laag maken bij OCF2 |
| | 11 | OC2 hoog maken bij OCF2 |
| **Phase correct PWM** | 00 | OC2 niet aangesloten |
| | 01 | niet gebruikt |
| | 10 | OC2 laag bij OCF2 tijdens optellen, OC2 hoog bij OCF2 tijdens aftellen |
| | 11 | OC2 hoog bij OCF2 tijdens optellen, OC2 laag bij OCF2 tijdens aftellen |
| **Fast PWM** | 00 | OC2 niet aangesloten |
| | 01 | niet gebruikt |
| | 10 | OC2 laag bij OCF2, OC2 hoog bij TOV2 |
| | 11 | OC2 hoog bij OCF2, OC2 laag bij TOV2 |

### Overige bits

Bit FOC2, *Force Output Compare 2*, forceert een directe actie van de waveformgenerator.

# H

# Compilers voor AVR

## H.1 C bij andere compilers voor AVR

Ondanks dat C een programmeertaal is, die dicht bij de machine staat, is de ANSI-standaard zo veel mogelijk machine onafhankelijk gemaakt. Omdat microcontrollers zich juist onderscheiden door een grote verscheidenheid aan mogelijkheden, is de ANSI-standaard niet direct geschikt voor microcontrollers. Er moeten aparte oplossingen worden bedacht voor het in- en uitlezen van signalen; voor de interruptafhandeling en voor het geheugenbeheer.
Elke compilerbouwer kiest daar zijn eigen oplossingen voor. Dat bevordert de overdraagbaarheid van programma's niet. Dat er verschillen zijn tussen compilers voor een microcontroller van Microchip en voor Atmel is zodoende niet vreemd. De architectuur van deze microcontrollers is immers anders. Doordat de compilerbouwers zelf oplossingen zoeken, wijken zelfs compilers voor een zelfde familie van microcontrollers van elkaar af.

**Tabel H.1 : De compilers voor AVR Atmel microcontrollers.**

| naam compiler | GNU | OS | naam leverancier |
|---|---|---|---|
| GNU avr-gcc | ja | Linux | open source |
| WinAVR | ja | Windows | open source |
| AtmanAvr C IDE | ja | Windows | Atman Electronics |
| IAR Embedded Workbench for Atmel AVR | nee | Windows | IAR Systems |
| Imagecraft C-Compiler for AVR | nee | Windows | Imagecraft |
| CodeVision | nee | Windows | HP InfoTech |

Voor de AVR-microcontroller van Atmel bestaan er verschillende varianten van de GNU C-Compiler (*avr-gcc*). Daarnaast zijn er een aantal andere compilers die niet op GNU gebaseerd zijn. Tabel H.1 geeft een overzicht.
De code, die geschreven is voor de ene GNU-compiler, is direct te gebruiken bij een andere GNU C-Compiler, mits er geen verouderde standaarden zijn gebruikt, zie paragraaf H.2

Code geschreven voor IAR, Imagecraft, CodeVision en de GNU C-Compilers is onderling niet uitwisselbaar. In code H.1 tot en met code H.4 staan de vier varianten van code 12.1, waarmee een led met behulp van een interrupt wordt aan- en uitgezet. Code H.1 is identiek aan code 12.1 en werkt alleen bij de GNU-compilers. In code H.2 tot en met H.4 staan de versies van deze code voor CodeVision, IAR en Imagecraft. Al deze vier versies zijn verschillend.

Ten eerste worden er andere headerbestanden gebruikt. Bij CodeVision, IAR en Imagecraft bevat de naam van het headerbestand voor de macro's `DDRA` en `PORTA` het typenummer van de component. De code moet aangepast worden als er overgestapt wordt naar een ATmega128. Bij Imagecraft staan in `macros.h` een aantal speciale macro's, zoals die voor `SEI()`. WinAVR en IAR hebben bij interrupts een extra headerbestand nodig.

Ten tweede zijn allerlei macro- en functienamen anders. De globale *interrupt enable* verschilt bij alle compilers. Bij CodeVision is het zelfs een assemblerinstructie. De code van IAR en Imagecraft kan eventueel met macrodefinities geschikt worden gemaakt voor WinAVR:

```
#define __enable_interrupt() sei()  // enab. glob. int. van IAR
#define SEI()                sei()  // enab. glob. int. van Imagecraft
```

Dit gaat niet voor alle instructies. Van de assemblerinstructie van CodeVision is bijvoorbeeld geen macro te maken.

Het is verwarrend dat de compilers van Imagecraft en IAR dezelfde naam hebben, namelijk `iccavr`.
Bij Imagecraft staat `icc` voor *Imagecraft C-Compiler* en bij IAR staat dit voor *IAR C-Compiler*.

Ten derde zijn de headers van de vier interruptroutines verschillend. IAR en Imagecraft gebruiken allebei een pragma (**`#pragma`**) voor een prototype van de interruptfunctie. Bij IAR is een prototype verplicht en bij Imagecraft is het optioneel. IAR gebruikt bij het prototype een pragma. Imagecraft gebruikt een pragma voor de interruptvector. WinAVR en CodeVision gebruiken geen pragma en hebben geen prototype nodig. De headers van de vier interruptroutines zijn totaal anders. De interruptvector staat bij CodeVision tussen rechte haken en bij WinAVR tussen ronde haken. Bij IAR en Imagecraft staan de namen van interruptvectoren bij de pragma's. Bovendien zijn de namen van de interruptvector bij alle vier de compilers anders: `INT0_vect`, `EXT_INT0`, `INT0_Int` of `iv_INT0`.

Naast de verschillen tussen de codes H.1 tot en met H.4 zijn er nog veel meer punten waar deze compilers van elkaar afwijken. Een belangrijk verschil tussen CodeVision en de anderen is het toepassen van bitvelden bij in- en uitgangsregisters. In CodeVision is met een punt en een cijfer achter de naam van een register het betreffende bit benaderbaar:

```
PORTA.0 = 1;
PORTB.2 = 0;
DDRC.4  = 1;
DDRC.5  = 1;
DDRC.6  = 1;
DDRC.7  = 1;
PORTD.0 = 0;
PORTD.3 = 0;
```

Dit boek gebruikt WinAVR. Hiervoor is gekozen omdat de compiler *open source* is en omdat deze volledig geïntegreerd kan worden in AVRstudio.
Daarnaast geeft WinAVR (`avr-gcc`) een compacte en snelle code en zijn er op internet veel voorbeelden te vinden.

Met WinAVR en alle andere `avr-gcc`'s wordt dat zo geschreven:

```
PORTA |=  _BV(0);
PORTB &= ~(_BV(2));
DDRC  |=  (_BV(4)|_BV(5)|_BV(6|_BV(7));
PORTD &= ~(_BV(0)|_BV(3));
```

Een van de nadelen van de methode van CodeVision is dat er bij de toekenning van de vier bits aan `DDRC` vier bewerkingen nodig zijn. Bij WinAVR is dat een bewerking. De uitdrukking `_BV(4)|_BV(5)|_BV(6|_BV(7)` is een getal dat in een keer met een *bitwise or* aan `DDRC` wordt toegekend.

**Code H.1 : De code met WinAVR (avr-gcc).**

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

ISR(INT0_vect)
{
  PORTA ^= 0x01;
}

int main(void) {
  DDRA  = 0x01;

  GICR  = 0x40;
  MCUCR = 0x02;

  sei();

  while(1);
}
```

**Code H.2 : De code met CodeVision (cvavr).**

```
#include <mega32.h>


interrupt [EXT_INT0] void ext_int0_isr(void)
{
  PORTA ^= 0x1;
}

int main(void) {
  DDRA  = 0x01;

  GICR  = 0x40;
  MCUCR = 0x02;

  #asm("sei");

  while(1) ;
}
```

**Code H.3 : De code met IAR (iccavr).**

```
#include <iom32.h>
#include <inavr.h>

#pragma vector = INT0_vect
__interrupt void INT0_Int(void);

__interrupt void INT0_Int(void)
{
  PORTA ^= 0x01;
}

int main(void) {
  DDRA  = 0x01;

  GICR  = 0x40;
  MCUCR = 0x02;

  __enable_interrupt();

  while(1);
}
```

**Code H.4 : De code met Imagecraft (iccavr).**

```
#include <iom32v.h>
#include <macros.h>

#pragma interrupt_handler int0_isr:iv_INT0


void int0_isr(void)
{
  PORTA ^= 0x01;
}

int main(void) {
  DDRA  = 0x01;

  GICR  = 0x40;
  MCUCR = 0x02;

  SEI();

  while(1);
}
```

Het lezen van een ingangspin gaat bij CodeVision zo:

```
if (PINC.2 == 1) {
   x = 1;
}
```

en bij WinAVR wordt bitmanipulatie toegepast:

```
if ( PINC & _BV(2) ) {
   x = 1;
}
```

**verouderde notatie**

```
outp(0x02, TCNT0);
x = inp(PORTC);
```

**verouderde notatie**

```
outb(TCNT0, 0x02);
x = inb(PORTC);
sbi(DDRA, 3);
cbi(DDRA, 4);
```

**aanbevolen notatie**

```
TCNT0 = 0x02;
x     = PORTC;
DDRA |= _BV(3);
DDRA &= ~(_BV(4));
```

**Figuur H.1 :** **De verouderde schrijfwijze `outp`, `inp`, `outb`, `inb`, `sbi` en `cbi`.** De linker en de middelste code geven de verouderde notaties. In de rechter code staat de aanbevolen notatie. Op de regel 1 en 2 staat het schrijven en lezen van een byte. Op de regel 3 en 4 staat het zetten en clearen van een bit.

## H.2 Verouderde notatie bij GNU C-compiler

De eerste uitgaven van `avr-gcc` gebruikte macro's om de bits en bytes van registers te setten, te clearen en te testen. Deze definities zijn verouderd (*obsolete*). Figuur H.1 toont de verouderde codes en de aanbevolen schrijfwijzes. Op het internet staan natuurlijk nog wel voorbeelden met de verouderde notaties.
De interrupts werden vroeger ook anders geschreven. Er werd onderscheid gemaakt tussen twee soorten interrupts: `SIGNAL` en `INTERRUPT`. Toen waren er ook twee include-bestanden nodig: `signal.h` en `interrupt.h`. De eerste bevatte de definitie van de typen `SIGNAL` en `INTERRUPT`. De tweede wordt nog steeds gebruikt en bevat onder andere de definitie van de `sei()` en bevat vanaf `avr-libc` versie 1.4.0 ook de definitie voor de nieuwe interrupt service routine `ISR`. Figuur H.2 toont de verouderde en de aanbevolen schrijfwijzes.

**verouderde notatie**

```
SIGNAL (SIG_INTERRUPT0)
{
  PORTA ^= 0x01;
}
```

**verouderde notatie**

```
INTERRUPT (SIG_INTERRUPT0)
{
  PORTA ^= 0x01;
}
```

**aanbevolen notatie**

```
ISR (INT0_vect)
{
  PORTA ^= 0x01;
}
```

**Figuur H.2 :** **De verouderde schrijfwijze voor de interrupt service routine.** De linker code is de verouderde `SIGNAL`-interrupt. De code in het midden is de verouderde `INTERRUPT`-interrupt. Rechts staat de aanbevolen notatie.

Het verschil tussen de twee verouderde interrupts is dat de `SIGNAL`-interrupt niet geïnterrumpeerd kan worden en dat een `INTERRUPT`-interrupt wel geïnterrumpeerd kan worden. De `SIGNAL`-interrupt maakt de interruptroutine helemaal af. De `INTERRUPT`-interrupt wordt pas afgemaakt nadat eerst de interruptroutine, waarmee deze is geïnterrumpeerd, is afgemaakt. Het interrumperen van een interrupt kan heel complex zijn. Het kan makkelijk een stack-overflow geven. Daarom werd de `INTERRUPT`-interrupt weinig gebruikt en werd het gebruik afgeraden.
In plaats van twee interrupts `SIGNAL`- en `INTERRUPT` kent `avr-gcc` nu een interrupt `ISR`. Deze is identiek aan de oude `SIGNAL`-interrupt. Om een interrumpeerbare interrupt te krijgen, moet de betreffende interruptvector worden geherdefinieerd:

```
void EXT_INT0_vect(void) __attribute__((interrupt));
void EXT_INT0_vect(void) {
  PORTA ^= 0x01;
}
```

Overigens zijn interrumpeerbare interrupts bijna nooit nodig. Gebruik altijd de gewone `ISR`.

# I

# Make

## I.1 De Makefile

In paragraaf 4 staat een voorbeeld voor het afdrukken van een leeftijd, waarbij de beschrijving is gesplitst over drie bestanden `main.c`, `age.h` en `age.c` die weergegeven zijn in code 4.5, 4.6 en 4.7. Om het programma `printage` te maken, werd de compiler drie aangeroepen. De compileropdrachten zouden ook in een *batch*-bestand worden gezet.

```
gcc -Wall -c main.c
gcc -Wall -c age.c
gcc -Wall -c main.o age.o
```

Vervolgens kan dit *batch*-bestand worden uitgevoerd.

Een betere aanpak is om gebruik te maken van `make`. Dat is een programma voor het automatisch bouwen van applicaties. Dit programma ziet welke bronbestanden gewijzigd zijn en zorgt ervoor dat deze opnieuw gecompileerd worden en dat er een nieuw uitvoerbaar programma wordt gemaakt. In code I.1 staat een makefile voor het programma `printage`.

**Code I.1 : Makefile 1.**

```
1 printage : main.o age.o
2 ←——TAB——→gcc -Wall -o printage main.o age.o
3 main.o   : main.c
4 ←——TAB——→gcc -Wall -c main.c
5 age.o    : age.c
6 ←——TAB——→gcc -Wall -c age.c
```

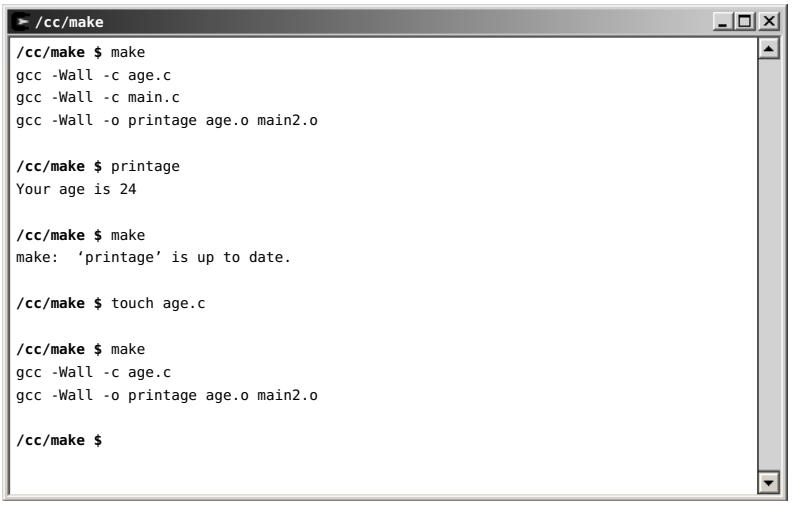
De opmaak van de makefile is strikt. Een commandoregel moet met een tabulatorteken beginnen. Voor `gcc` moet zodoende een <tab> staan. In de code I.1 is dat met ←——TAB——→ expliciet gemaakt. Voor de commandoregel staat een afhankelijkheidsregel. Deze bestaat uit een doel (*target*), een dubbele punt en de vereisten (*prerequisites*). De syntax luidt dus:

```
target : prerequisite1 prerequisite2 ... prerequisiteN
←——TAB——→command1
←——TAB——→command2
←——TAB——→...
←——TAB——→commandN
```

De vereisten zijn in het algemeen de bronbestanden die nodig zijn om het doel te maken. In de volgende voorbeelden van dit hoofdstuk worden de tabs niet meer expliciet met ←——TAB——→ weergegeven.

Een makefile wordt uitgevoerd met het commando `make`. Bij een aanroep zonder argumenten gaat het programma automatisch de makefile `Makefile` uitvoeren. Figuur I.1 laat zien dat bij de eerste aanroep alle regels worden uitgevoerd en dat er een applicatie `printage` is gemaakt. Bij de tweede aanroep is, mits `age.c` en `main.c` niet gewijzigd zijn, `printage` jonger dan deze twee bestanden en is het niet zinvol om de applicatie opnieuw te bouwen. `make` meldt dat de applicatie `up-to-date` is. Met het Unix-commando `touch` wordt de zogenoemde *time stamp* van een bestand gewijzigd. Bij de derde aanroep is `age.c` daardoor jonger dan de applicatie en zou dus gewijzigd kunnen zijn. `make` compileert `age.c` opnieuw en omdat `age.o` hierdoor ook gewijzigd is, zal er ook een nieuwe `printage` worden gemaakt.

Een makefile hoeft niet per se `Makefile` te heten. De optie `-f` voegt een bestand met een willekeurige naam toe. De makefile `makefile.cygwin` wordt gemaakt met het Unix-commando `make -f makefile.cygwin`. Het Unix-commando `make -f Makefile` is identiek aan het commando `make`.




**Figuur I.1 : De toepassing van make met de makefile voor printage.** De eerste aanroep `make` bestaat `printage` nog niet en wordt de applicatie gebouwd. Bij de tweede aanroep is, mits `age.c` en `main.c` niet gewijzigd zijn, het niet zinvol om de applicatie opnieuw te bouwen en geeft `make` alleen de mededeling dat `printage` up-to-date is. Bij de aanroep is `age.c` gewijzigd en wordt `age.o` en dus ook `printage` opnieuw gemaakt.

Code I.2 bevat drie definities: `TARGET`, `CC` en `CFLAGS` voor respectievelijk de naam van de applicatie, de naam van de compiler en een string met opties (`CFLAGS`).

**Code I.2 : Makefile 2.**

```
1  TARGET = printage
2  CC     = gcc
3  CFLAGS = -Wall
4
5  $(TARGET) : main.o age.o
6          $(CC) $(CFLAGS) -o $(TARGET) main.o age.o
7
8  age.o   : age.c
9          $(CC) $(CFLAGS) -c age.c
10
11 main.o  : main.c
12         $(CC) $(CFLAGS) -c main.c
```

Met deze macrodefinities is de code veel eenvoudiger te onderhouden, omdat deze aan het begin van de code staan en de namen en opties hoeven nu maar op een plaats te worden gewijzigd.

In code I.2 staat op de regels 11 en 12 in feite hetzelfde als op de regels 8 en 9. Naast de expliciete commandoregels zijn er ook impliciete commando's, zoals code I.3 laat zien.

**Code I.3 : Makefile 3.**

```
1  TARGET = printage
2  CC     = gcc
3  CFLAGS = -Wall
4
5  .SUFFIXES: .o .c
6
7  $(TARGET): main.o age.o
8          $(CC) $(CFLAGS) -o $(TARGET) main.o age.o
9
10 %.o: %.c
11         $(CC) $(CFLAGS) -c $<
```

In regel 10 van code I.3 wordt de *pattern rule* `%.o: %.c` gebruikt. Voor elke `.o`-bestand is er een `.c`-bestand vereist met dezelfde basisnaam. Bij dit commando verwijst `$<` naar de namen van de bronbestanden uit de afhankelijkheidsregel.
Door het headerbestand `age.h` toe te voegen aan de afhankelijkheidsregel zal er ook een nieuw uitvoerbaar programma gemaakt worden. Verder zijn in code I.4 de objectbestanden met een macrodefinitie vooraf gedefinieerd. De speciale macro `$@` verwijst naar de gewenste naam van het doel (*target*).

**Code I.4 : Makefile 4.**

```
1  TARGET = printage
2  CC     = gcc
3  CFLAGS = -Wall
4  # the application printage needs two objectfiles
5  OFILES = main.o age.o
6
7  .SUFFIXES: .o .c .h
8
9  $(TARGET): $(OFILES)
10         $(CC) $(CFLAGS) -o $@ $(OFILES)
11
12 %.o: %.c age.h
13         $(CC) $(CFLAGS) -c $<
```

Regels die met een `#` beginnen worden door `make` genegeerd en kunnen dus commentaar bevatten.
De makefile kan met *wildcards* helemaal worden ontdaan van bestandsnamen. Mits in de betreffende werkfolder geen andere c-bestanden staan, geeft in code I.5 de *wildcard* op regel 4 alle benodigde c-bestanden. In dit voorbeeld zijn dat `main.c` en `age.c`. Op dezelfde manier geeft de *wildcard* op regel 5 alle headerbestanden. Hier is dat alleen `age.h`.
De speciale macro `$^` expandeert alle vereisten uit de afhankelijkheidsregel. Het target `clean` ruimt alle objectbestanden op. Het commando `make clean` verwijdert al deze bestanden.

## I.2 De Makefile bij AVRstudio

AVRstudio gebruikt de GNU-compiler `avr-gcc` als C-compiler. De commando's `Build` en `Rebuild All` uit het menu `Build` voeren een `make` uit. De bijbehorende ma-

**Code I.5 : Makefile 5.**

```
1  TARGET = printage
2  CC     = gcc
3  CFLAGS = -Wall
4  CFILES = $(wildcard *.c)
5  HFILES = $(wildcard *.h)
6  OFILES = $(CFILES:.c=.o)
7
8  .SUFFIXES: .o .c .h
9
10 $(TARGET): $(OFILES)
11         $(CC) $(CFLAGS) -o $@ $^
12
13 %.o: %.c ${HFILES}
14         $(CC) $(CFLAGS) -c $<
15
16 clean:
17         rm -f *.o
```

kefile kan met `Export Makefile` uit het menu `Build` worden geëxporteerd. Code I.2 toont deze makefile. Bij het uitvoeren (`Build`) van deze makefile verschijnt in het uitvoervenster van AVRstudio de tekst van figuur I.2.

```
1  Build started 26.9.2007 at 21:10:47
2  avr-gcc.exe -mmcu=atmega32 -Wall -gdwarf-2 -O0 -fsigned-char -MD -MP -MT led_int.o
                                                              -MF dep/led_int.o.d -c ../led_int.c
3  avr-gcc.exe -mmcu=atmega32  led_int.o -o led_int.elf
4  avr-objcopy -O ihex -R .eeprom  led_int.elf led_int.hex
5  avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0
                                  --no-change-warnings -O ihex led_int.elf led_int.eep || exit 0
6
7  AVR Memory Usage
8  ----------------
9  Device: atmega32
10
11 Program:     256 bytes (0.8% Full)
12 (.text + .data + .bootloader)
13
14 Data:          0 bytes (0.0% Full)
15 (.data + .bss + .noinit)
16
17 Build succeeded with 0 Warnings...
```

**Figuur I.2 : De uitvoer bij het bouwen van een ontwerp in AVRStudio.**

De regels 2, 3, 4 en 5 van deze uitvoer zijn de commando's van de regels 39, 43, 46 en 49 uit de makefile voor respectievelijk het compileren, het linken, het creëren van het hex-bestand en het maken van `eeprom`-bestand. Regel 53 geeft een samenvatting van het geheugengebruik. Deze staat in het uitvoervenster tussen de regels 7 en 15. In de makefile staan tussen regel 1 en regel 30 alle definities die de commando's gebruiken. Op regel 56 staat een afhankelijkheidsregel die bepaalt welke headerbestanden gebruikt worden.

Deze Makefile is gemaakt met versie 4.12 van AVRstudio en gebruikt de optimalisatie `-O0`.

**Code I.6 : De Makefile voor avr-gcc bij AVRStudio.**

```
## General Flags
PROJECT = led_int
MCU = atmega32
TARGET = led_int.elf
CC = avr-gcc.exe

## Options common to compile, link and assembly rules
COMMON = -mmcu=$(MCU)

## Compile options common for all C compilation units.
CFLAGS = $(COMMON)
CFLAGS += -Wall -gdwarf-2 -O0
CFLAGS += -MD -MP -MT $(*F).o -MF dep/$(F).d

## Assembly specific flags
ASMFLAGS = $(COMMON)
ASMFLAGS += $(CFLAGS)
ASMFLAGS += -x assembler-with-cpp -Wa,-gdwarf2

## Linker flags
LDFLAGS = $(COMMON)
LDFLAGS +=

## Intel Hex file production flags
HEX_FLASH_FLAGS = -R .eeprom

HEX_EEPROM_FLAGS = -j .eeprom
HEX_EEPROM_FLAGS += --set-section-flags=.eeprom="alloc,load"
HEX_EEPROM_FLAGS += --change-section-lma .eeprom=0 --no-change-warnings

## Objects that must be built in order to link
OBJECTS = led_int.o

## Build
all: $(TARGET) led_int.hex led_int.eep size

## Compile
led_int.o: ../led_int.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

## Link
$(TARGET): $(OBJECTS)
	 $(CC) $(LDFLAGS) $(OBJECTS) $(LIBDIRS) $(LIBS) -o $(TARGET)

%.hex: $(TARGET)
	avr-objcopy -O ihex $(HEX_FLASH_FLAGS)  $< $

%.eep: $(TARGET)
	-avr-objcopy $(HEX_EEPROM_FLAGS) -O ihex $< $ || exit 0

size: ${TARGET}
	echo
	avr-size -C --mcu=${MCU} ${TARGET}

## Other dependencies
-include $(shell mkdir dep 2>/dev/null) $(wildcard dep/*)
```

**Tabel J.1 : Tabel met ASCII-waarden.** De eerste kolom geeft de decimale waarde, de volgende drie kolommen geven de octale, hexadecimale en binaire waarde. De laatste kolom geeft de omschrijving of het afdrukbare karakter. Bij sommige symbolen staat ook de *escape sequence*.

| Dec | Oct | Hex | Bin | Symbool |
|---|---|---|---|---|
| 0 | 000 | 00 | 0000 0000 | NUL, Null, '\0' |
| 1 | 001 | 01 | 0000 0001 | SOH |
| 2 | 002 | 02 | 0000 0010 | STX |
| 3 | 003 | 03 | 0000 0011 | ETX |
| 4 | 004 | 04 | 0000 0100 | EOT |
| 5 | 005 | 05 | 0000 0101 | ENQ |
| 6 | 006 | 06 | 0000 0110 | ACK |
| 7 | 007 | 07 | 0000 0111 | BEL, Bell, '\a' |
| 8 | 010 | 08 | 0000 1000 | BS, Backspace, '\b' |
| 9 | 011 | 09 | 0000 1001 | HT, Horizontal Tab, '\t' |
| 10 | 012 | 0A | 0000 1010 | LF, Line Feed, '\n' |
| 11 | 013 | 0B | 0000 1011 | VT, Vertical Tab, '\v' |
| 12 | 014 | 0C | 0000 1100 | FF, Form Feed, '\f' |
| 13 | 015 | 0D | 0000 1101 | CR, Carriage Return, '\r' |
| 14 | 016 | 0E | 0000 1110 | SO |
| 15 | 017 | 0F | 0000 1111 | SI |
| 16 | 020 | 10 | 0001 0000 | DLE |
| 17 | 021 | 11 | 0001 0001 | DC1 |
| 18 | 022 | 12 | 0001 0010 | DC2 |
| 19 | 023 | 13 | 0001 0011 | DC3 |
| 20 | 024 | 14 | 0001 0100 | DC4 |
| 21 | 025 | 15 | 0001 0101 | NAK |
| 22 | 026 | 16 | 0001 0110 | SYN |
| 23 | 027 | 17 | 0001 0111 | ETB |
| 24 | 030 | 18 | 0001 1000 | CAN |
| 25 | 031 | 19 | 0001 1001 | EM |
| 26 | 032 | 1A | 0001 1010 | SUB |
| 27 | 033 | 1B | 0001 1011 | ESC, Escape, '\e' |
| 28 | 034 | 1C | 0001 1100 | FS |
| 29 | 035 | 1D | 0001 1101 | GS |
| 30 | 036 | 1E | 0001 1110 | RS |
| 31 | 037 | 1F | 0001 1111 | US |
| 32 | 040 | 20 | 0010 0000 | SP Space, '\ ' |
| 33 | 041 | 21 | 0010 0001 | ! |
| 34 | 042 | 22 | 0010 0010 | " |
| 35 | 043 | 23 | 0010 0011 | # |
| 36 | 044 | 24 | 0010 0100 | $ |
| 37 | 045 | 25 | 0010 0101 | % |
| 38 | 046 | 26 | 0010 0110 | & |
| 39 | 047 | 27 | 0010 0111 | ' |
| 40 | 050 | 28 | 0010 1000 | ( |
| 41 | 051 | 29 | 0010 1001 | ) |
| 42 | 052 | 2A | 0010 1010 | * |
| 43 | 053 | 2B | 0010 1011 | + |
| 44 | 054 | 2C | 0010 1100 | , |
| 45 | 055 | 2D | 0010 1101 | - |
| 46 | 056 | 2E | 0010 1110 | . |
| 47 | 057 | 2F | 0010 1111 | / |
| 48 | 060 | 30 | 0011 0000 | 0 |
| 49 | 061 | 31 | 0011 0001 | 1 |
| 50 | 062 | 32 | 0011 0010 | 2 |
| 51 | 063 | 33 | 0011 0011 | 3 |
| 52 | 064 | 34 | 0011 0100 | 4 |
| 53 | 065 | 35 | 0011 0101 | 5 |
| 54 | 066 | 36 | 0011 0110 | 6 |
| 55 | 067 | 37 | 0011 0111 | 7 |
| 56 | 070 | 38 | 0011 1000 | 8 |
| 57 | 071 | 39 | 0011 1001 | 9 |
| 58 | 072 | 3A | 0011 1010 | : |
| 59 | 073 | 3B | 0011 1011 | ; |
| 60 | 074 | 3C | 0011 1100 | < |
| 61 | 075 | 3D | 0011 1101 | = |
| 62 | 076 | 3E | 0011 1110 | > |
| 63 | 077 | 3F | 0011 1111 | ? |

| Dec | Oct | Hex | Bin | Symbool |
|---|---|---|---|---|
| 64 | 100 | 40 | 0100 0000 | @ |
| 65 | 101 | 41 | 0100 0001 | A |
| 66 | 102 | 42 | 0100 0010 | B |
| 67 | 103 | 43 | 0100 0011 | C |
| 68 | 104 | 44 | 0100 0100 | D |
| 69 | 105 | 45 | 0100 0101 | E |
| 70 | 106 | 46 | 0100 0110 | F |
| 71 | 107 | 47 | 0100 0111 | G |
| 72 | 110 | 48 | 0100 1000 | H |
| 73 | 111 | 49 | 0100 1001 | I |
| 74 | 112 | 4A | 0100 1010 | J |
| 75 | 113 | 4B | 0100 1011 | K |
| 76 | 114 | 4C | 0100 1100 | L |
| 77 | 115 | 4D | 0100 1101 | M |
| 78 | 116 | 4E | 0100 1110 | N |
| 79 | 117 | 4F | 0100 1111 | O |
| 80 | 120 | 50 | 0101 0000 | P |
| 81 | 121 | 51 | 0101 0001 | Q |
| 82 | 122 | 52 | 0101 0010 | R |
| 83 | 123 | 53 | 0101 0011 | S |
| 84 | 124 | 54 | 0101 0100 | T |
| 85 | 125 | 55 | 0101 0101 | U |
| 86 | 126 | 56 | 0101 0110 | V |
| 87 | 127 | 57 | 0101 0111 | W |
| 88 | 130 | 58 | 0101 1000 | X |
| 89 | 131 | 59 | 0101 1001 | Y |
| 90 | 132 | 5A | 0101 1010 | Z |
| 91 | 133 | 5B | 0101 1011 | [ |
| 92 | 134 | 5C | 0101 1100 | \ |
| 93 | 135 | 5D | 0101 1101 | ] |
| 94 | 136 | 5E | 0101 1110 | ^ |
| 95 | 137 | 5F | 0101 1111 | _ |
| 96 | 140 | 60 | 0110 0000 | ` |
| 97 | 141 | 61 | 0110 0001 | a |
| 98 | 142 | 62 | 0110 0010 | b |
| 99 | 143 | 63 | 0110 0011 | c |
| 100 | 144 | 64 | 0110 0100 | d |
| 101 | 145 | 65 | 0110 0101 | e |
| 102 | 146 | 66 | 0110 0110 | f |
| 103 | 147 | 67 | 0110 0111 | g |
| 104 | 150 | 68 | 0110 1000 | h |
| 105 | 151 | 69 | 0110 1001 | i |
| 106 | 152 | 6A | 0110 1010 | j |
| 107 | 153 | 6B | 0110 1011 | k |
| 108 | 154 | 6C | 0110 1100 | l |
| 109 | 155 | 6D | 0110 1101 | m |
| 110 | 156 | 6E | 0110 1110 | n |
| 111 | 157 | 6F | 0110 1111 | o |
| 112 | 160 | 70 | 0111 0000 | p |
| 113 | 161 | 71 | 0111 0001 | q |
| 114 | 162 | 72 | 0111 0010 | r |
| 115 | 163 | 73 | 0111 0011 | s |
| 116 | 164 | 74 | 0111 0100 | t |
| 117 | 165 | 75 | 0111 0101 | u |
| 118 | 166 | 76 | 0111 0110 | v |
| 119 | 167 | 77 | 0111 0111 | w |
| 120 | 170 | 78 | 0111 1000 | x |
| 121 | 171 | 79 | 0111 1001 | y |
| 122 | 172 | 7A | 0111 1010 | z |
| 123 | 173 | 7B | 0111 1011 | { |
| 124 | 174 | 7C | 0111 1100 | \| |
| 125 | 175 | 7D | 0111 1101 | } |
| 126 | 176 | 7E | 0111 1110 | ~ |
| 127 | 177 | 7F | 0111 1111 | DEL |

# J

# ASCII

ASCII staat voor American Standard Code for Information Interchange en is een standaard om letters, cijfers, leestekens en andere symbolen te representeren. Aan ieder teken is een geheel getal gekoppeld, waarmee het teken wordt aangeduid.

De ASCII-waarden zijn 7-bits breed en representeren 128 verschillende symbolen. In tabel J.1 zijn deze waarden en de bijbehorende betekenissen vermeld. De eerste 32 ASCII-waardes 0 tot en met 31 en de laatste ASCII-waarde 127 zijn besturingscommando's en zijn niet afdrukbaar. Deze ASCII-waarden waren bedoeld als meta-informatie bij bijvoorbeeld de communicaties met een printer of een telex. De andere 95 waarden (32 tot en met 126) zijn wel afdrukbaar. Sommige niet-afdrukbare symbolen hebben nog steeds een functie in de programmeertaal C. Voor deze symbolen, zoals de horizontale tabulator, is ook de *escape sequence* vermeld.

Later zijn er verschillende uitbreidingen van de ASCII-tabel gemaakt tot 256 karakters. Een voorbeeld hiervan is de ISO/IEC 8859-1. Omdat er veel meer karakters dan 256 bestaan, zijn er universele karaktersets ontwikkeld waarmee de karakters van vele talen beschreven worden, zoals Unicode, UTF-8, UTF-16 en UTF-32.

# Index

## F



## G

## N



## O



## P

## T



## U

## V



## W



## Z