De taal C en de Xmega

tweede druk 2016

Wim Dolman

Informatie over dit boek en andere uitgaven is verkrijgbaar bij:
Wim Dolman
info@dolman-wim.nl
http://www.dolman-wim.nl/xmega/

De taal C en de Xmega is geschreven door Wim Dolman en in eigen beheer uitgegeven en gedrukt bij BoekenGilde drukkerij.
©2016 Wim Dolman, Culemborg
tweede druk 2016 (licht herzien in 2019)

Vormgeving en opmaak is verzorgd door de auteur.

ISBN: 978-94-632-3852-6
NUR: 173/959

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand, of openbaar gemaakt, in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen of enige andere manier, zonder voorafgaande schriftelijke toestemming van de auteur.

Hoewel aan de totstandkoming van deze uitgave de uiterste zorg is besteed, kunnen er in deze uitgave fouten en onvolledigheden staan. De auteur en de uitgever aanvaarden geen aansprakelijkheid voor de gevolgen van eventueel voorkomende fouten en onvolledigheden.

Voorwoord

Programmeren en Microcontrollers zijn vakken die gegeven worden in het eerste jaar bij de afdeling Elektrotechniek van de Hogeschool van Amsterdam. Deze vakken behandelen de programmeertaal C en de Xmega van Atmel. Studenten, die deze vakken volbracht hebben, kunnen niet-grafische applicaties maken voor een pc en applicaties maken met een Xmega. Hiervoor is kennis nodig van de taal C en kennis van microcontrollers in het algemeen. Bovendien moet de student weten hoe de Xmega is opgebouwd, op de hoogte zijn van de specifieke mogelijkheden van deze microcontroller en kennis hebben van de taal C voor de Xmega.

Dit boek heeft drieëntwintig hoofdstukken. Hoofdstuk 1 is een inleiding over embedded systemen. De microcontroller is de belangrijkste component van een embedded systeem. C is een veel gebruikte taal voor microcontrollers. Voor de ontwikkeling van embedded systemen is de taal C belangrijk.

De hoofdstukken 2 tot en met 13 behandelen de taal C zonder een microcontroller. In deze hoofdstukken worden alleen applicaties voor de pc gemaakt. Aan bod komen onder andere het compilatietraject, declaraties van variabelen, uitvoer naar het beeldscherm, invoer met het toetsenbord, gebruik van functies, voorwaardelijke opdrachten, herhalingsopdrachten, standaard datatypen, de opmaak van de code, arrays, pointers, strings, recursie, datastructuren en het lezen van en schrijven naar bestanden.

Hoofdstuk 14 bespreekt een aantal eigenschappen van de Xmega. Deels is dit een samenvatting van de datasheet van de Xmega, anderzijds is het juist een aanvulling daarop. Dit hoofdstuk bespreekt ook het ontwikkeltraject voor de Xmega.

De overige hoofdstukken behandelen de Xmega en de taal C voor de Xmega. Hoofdstuk 15 tot en met 17 bespreekt de knipperende led, de dotmatrix, het 7-segmentsdisplay, het gebruik van opzoektabellen, externe interrupts, het gebruik van flash en de timer/counter. Naast de uitleg van de code wordt de interne opbouw van Xmega verder verklaard en is er aandacht voor de hardware van de demonstratieschakelingen, zoals de aansturing van leds, het ontkoppelen van de Xmega en het bestrijden van contactdender.

De hoofdstukken 18 tot en met 22 behandelen de aansturing van een LCD en het gebruik van een groot aantal perifere blokken, zoals: de communicatie met de USART, de AD-converter, de SPI en de I^2C-interface en de toepassing van de timers bij pulsbreedtemodulatie. Deze hoofdstukken geven veel achtergrondinformatie en laten meerdere manieren zien waarop deze onderdelen gebruikt kunnen

worden. Hoofdstuk 23 behandelt in het kort: de DA-converter, de analoge comparator, de inputcapture, het kloksysteem van de Xmega, de realtime-counter, het lezen uit en het schijven naar het EEPROM, het gebruik van flash, DMA, slaapstanden, de watchdog en het atomisch blok.

Om dit boek zo concreet mogelijk te maken is het geschreven rond een bepaalde microcontroller. Er is gekozen voor de Xmega256a3u van Atmel. Dit is een microcontroller uit de Xmega-reeks. De Xmega is de opvolger van de populaire ATmega.

De software voor de Xmega is anders — meer gestructureerd — opgezet dan die voor de ATmega. Bovendien kent Atmel voor zijn microcontrollers het ASF, Atmel Software Framework. Dit is een uitgebreide softwarebibliotheek. De software van de application notes zijn onderdeel van deze bibliotheek. In de praktijk zijn er zodoende drie stijlen te onderscheiden: de ATmega-stijl, de Xmega-stijl en de ASF-stijl.
De ATmega-stijl is minder geschikt bij programma's voor de Xmega. De ASF-stijl schermt de registers af voor de gebruiker. De meeste bibliotheken bestaan uit een groot aantal functies waarmee de registers kunnen worden geconfigureerd. De Xmega-stijl staat daarentegen dicht bij de microcontroller. Om de opbouw en de organisatie van een microcontroller te leren kennen, is de Xmega-stijl het meest geschikt. De ASF-stijl wordt alleen gebruikt als een bibliotheek echt noodzakelijk is, zoals bij de UART en de TWI.

De belangrijkste wijziging bij deze tweede druk is, dat alle codes voor de Xmega zijn aangepast aan het nieuwe Xmega-bord, dat gebruikt wordt bij de afdeling Elektrotechniek van de Hogeschool van Amsterdam. Dit Xmega-bord heeft een extra Xmega, die gebruikt wordt als programmer en als RS232-USB-converter, een radiomodule op basis van de Nordic nRF24L01+ en een SD-kaarthouder. Er is een bijlage over het gebruik van de microSD-kaart en een bijlage over de draadloze module toegevoegd. Daarnaast is aan hoofdstuk 23 een paragraaf over flash en een paragraaf over DMA toegevoegd.

Het boek is geschreven met LaTeX. De gebruikte compiler is pdfTeX, 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 tekeningen zijn gemaakt met Mayura Draw 4.5 en als PDF-bestand toegevoegd.

Veel programmacodes zijn compleet en kunnen zonder enige aanpassing direct gecompileerd en uitgevoerd worden. De algemene C is getest met de GNU C-Compiler versie 4.8.2, die bij de Cygwin-omgeving versie 1.7.28 hoort. De C voor de Xmega is getest met Atmel Studio 7.0 met de AVR GNU C-Compiler versie 4.9.2.

De waarschuwingen en foutmeldingen in het boek wijken bij de Cygwin-voorbeelden af van versie 4.8.2. Deze horen nog bij versie 3.4.4. Dit is bewust gedaan, omdat de mededelingen bij versie 4.8.2 vaak zeer uitvoerig zijn.

Bij dit boek hoort de internetsite `http://www.dolman-wim.nl/xmega/` met een hulpprogramma voor het berekenen van de baudsnelheid, bibliotheken voor de LCD, de UART, de microSD-kaart, de nRF2424L01+ en alle programmacodes uit het boek.

Inhoud

1 De Microcontroller 1

1.1 Embedded Systemen . . . 1
1.2 De architectuur van de microprocessor en de microcontroller . 4
1.3 Geheugens en geheugenstructuur . . . 6
1.4 Harvard-architectuur . . . 7
1.5 RISC en CISC . . . 8
1.6 De keuze voor een microcontroller . . . 9

2 De taal C 11

2.1 Hello World . . . 12
2.2 Het compilatietraject . . . 14
2.3 Compilers . . . 16
2.4 Foutmeldingen . . . 17

3 C in het kort 19

3.1 Variabelen, declaraties en initialisatie . . . 20
3.2 Datatypen . . . 22
3.3 Samengestelde datatypen: arrays en strings . . . 23
3.4 Rekenkundige bewerkingen . . . 24
3.5 Afdrukken . . . 25
3.6 Voorwaardelijke opdrachten . . . 25
3.7 Herhalingsopdrachten . . . 27
3.8 Voorbeelden . . . 28

4 Functies 31

4.1 Verdeel en heers . . . 32
4.2 De opbouw van een functie . . . 34
4.3 Formele en actuele parameters . . . 38
4.4 De scope van functies en variabelen . . . 38
4.5 Call by reference . . . 40
4.6 Blokschema's, stroomdiagrammen, pseudocode en algoritmes . 41
4.7 Voorbeeld: cijferprogramma . . . 44
4.8 Het verschil tussen == en = . . . 47
4.9 Functies en programmeervaardigheden . . . 48

5 In- en uitvoer 49
5.1 Geformatteerde uitvoer . . . 50
5.2 Geformatteerde invoer . . . 52
5.3 Voorbeeld: invoer gegevens cijferprogramma . . . 54
5.4 Ongeformatteerde in- en uitvoer . . . 55
5.5 Alternatief voor het invoerprobleem . . . 58
5.6 Argumenten doorgeven aan een programma . . . 59
5.7 Declaratie en het gebruik van strings . . . 62

6 Voorwaardelijke opdrachten 67
6.1 Het if-statement: de if-vorm . . . 68
6.2 De bloktoewijzing . . . 69
6.3 Het if-statement: de if-else vorm . . . 70
6.4 Het nesten van if-statements . . . 70
6.5 Het if-statement: de if-else-if vorm . . . 72
6.6 Het switch-statement . . . 72
6.7 Definities en macro's . . . 77
6.8 De conditionele operator . . . 80

7 Herhalingsopdrachten 81
7.1 De for-lus . . . 81
7.2 De komma-operator . . . 84
7.3 De while-lus . . . 84
7.4 De do-while-lus of do-lus . . . 85
7.5 Het break-statement en het continue-statement . . . 86

8 Structuur en Opmaak 89
8.1 Commentaar . . . 90
8.2 Opmaak . . . 91
8.3 Naamgeving . . . 93

9 Datatypen en Operatoren 95
9.1 Gehele getallen . . . 96
9.2 Typecasting bij gehele getallen . . . 97
9.3 Gebroken getallen . . . 100
9.4 Typecasting bij gebroken getallen . . . 101
9.5 Constanten bij gebroken getallen . . . 103
9.6 Hexadecimaal, octaal en binair . . . 104
9.7 Rekenkundige operatoren . . . 105
9.8 Het karaktertype `char` en de speciale karakters . . . 106
9.9 Boolean . . . 107
9.10 De relationele bewerkingen . . . 107
9.11 Logische operatoren . . . 108
9.12 Bitbewerkingen . . . 108
9.13 Verkorte schrijfwijze bij toekenningen . . . 109
9.14 Bewerkingsvolgorde operatoren . . . 110
9.15 Voorbeeld: afdrukken binaire waarden . . . 112
9.16 Meer over operatoren, datatypen en declaraties . . . 113

10 Arrays 119

10.1 De getallen van Fibonacci en de Gulden Snede 119
10.2 Berekenen getallen van Fibonacci en de Gulden Snede 121
10.3 Declaraties van arrays 122
10.4 Toewijzingen bij arrays 123
10.5 Lezen buiten het bereik van een array 123
10.6 Schrijven buiten het bereik van een array 124
10.7 Meerdimensionale arrays 124
10.8 De driehoek van Pascal 127
10.9 Berekening driehoek van Pascal en getallen van Fibonacci . . . 127
10.10 Dynamische geheugenallocatie 130

11 Pointers 131

11.1 Declaraties van pointers 132
11.2 Toewijzingen met pointers 132
11.3 Rekenen met pointers 133
11.4 Fouten met pointers 134
11.5 Getallen van Fibonacci en Gulden Snede met pointers 135
11.6 Toepassingen pointers 138
11.7 Voorbeelden met pointers 139
11.8 Dynamische geheugenallocatie bij eendimensionale arrays . . . 141
11.9 VLA: *variable length array* 142
11.10 Dynamische geheugenallocatie bij tweedimensionale arrays . . 144

12 Strings 151

12.1 Declaratie van en toekenningen aan strings 152
12.2 Op veilige wijze strings gebruiken 154
12.3 Stringfuncties 155
12.4 Array van strings 157

13 Advanced C 159

13.1 Lezen en schrijven naar bestanden 159
13.2 Recursie 168
13.3 Pointers naar functies 176
13.4 Samengestelde datatypes 177
13.5 Datastructuren 181
13.6 Functies met een variabele argumentenlijst 185
13.7 Preprocessoropdrachten of *compiler directives* 186

14 De Xmega 189

14.1 De opbouw van de Xmega 190
14.2 De behuizing van de Xmega 193
14.3 De geheugenorganisatie bij de Xmega 194
14.4 De systeemklok en klokopties 196
14.5 Het programmeren van de Xmega 197
14.6 De ontwikkelomgeving voor de Xmega 198

15 Generieke IO 199

15.1 De schakeling voor Led Blink 201
15.2 De software voor Led Blink 201
15.3 De generieke IO van de Xmega 202
15.4 De organisatie van de registers bij de Xmega: de Xmega-stijl . . 205
15.5 De registers bij de verouderde ATmega-stijl 207
15.6 Bitbewerkingen en de *read-write-modify*-methode 208
15.7 Vertragingstijden en de macrodefinitie `F_CPU` 212
15.8 De generieke IO als ingang gebruiken 213
15.9 Het aan- en uitzetten van een led met een drukknop 214
15.10 Contactdender 215
15.11 Hardwarematige antidendermaatregelen 216
15.12 Softwarematige antidendermaatregelen 217

16 Interrupts 221

16.1 Het interruptmechanisme 222
16.2 De interrupts en het interruptmechanisme bij de Xmega 223
16.3 Een voorbeeld met externe interrupt `0` 224
16.4 Timer/counters 229
16.5 Een tijdvertraging maken met een timer/counter 232
16.6 Een antidenderalgoritme met een externe interrupt en `TCE0` . . 237
16.7 Groepsconfiguratie, groepsmasker, groepspositie, bitmasker en bitpositie 239

17 Displays 243

17.1 De ledbar 244
17.2 Aansturing leds 250
17.3 Een tweedimensionale ledarray of dotmatrix 251
17.4 Cijfers afbeelden op een dotmatrix 253
17.5 Cijfers afbeelden op een dotmatrix met interrupt en timer . . . 255
17.6 Cijfers afbeelden op een dotmatrix met de gegevens in flash . . 258
17.7 Een 4-digit 7-segmentdisplay aansturen 260
17.8 Het uitlezen van zes drukknoppen 263
17.9 Conclusie 266

18 Liquid Crystal Display 267

18.1 Het aansluiten van een HD44780 op de Xmega 268
18.2 Het karaktergeoriënteerde display op basis van HD44780 270
18.3 Toepassing LCD met 8-bit modus en tijdvertraging 277
18.4 Toepassing met bewegende tekst 279
18.5 Toepassing LCD met 4-bit modus en busy flag 280
18.6 Toepassing met een LCD-bibliotheek 283
18.7 Geformatteerd afdrukken op een LCD 286
18.8 Het weergeven van gebroken getallen op een LCD 287

19 UART 291

19.1 Opbouw USART en het instellen van baudsnelheid 293
19.2 Instelling protocol 296
19.3 Ontvangen en verzenden van data 297
19.4 Het versturen van karakters via USART1 van poort D 298
19.5 Het ontvangen, converteren en versturen van karakters 300
19.6 Toepassing met gebruik van een interrupt 301
19.7 Het gebruik van een circulaire buffer 303
19.8 Circulaire buffers bij de communicatie met een UART 305
19.9 De USART-driver van Atmel en een bijbehorende wrapper . . 307
19.10 Het versturen van getallen via de UART 314
19.11 Het creëren van een stream voor printf en scanf 314
19.12 Een vaste stream voor USART0 van poort F 319

20 Analog-to-Digital Converter 323

20.1 Analoog-digitaalconversie 324
20.2 De opbouw van de ADC bij de Xmega 327
20.3 De conversiemethoden 333
20.4 Fouten bij AD-conversie 335
20.5 Toepassing: handmatige *unsigned single-ended* conversie 337
20.6 Toepassing: handmatige *signed single-ended* conversie 341
20.7 Toepassing: handmatige conversie met differentiële modus . . . 344
20.8 Toepassing: differentiële conversie met behulp van een interrupt 345
20.9 Toepassing: differentiële conversie op vaste tijdstippen 347
20.10 Toepassing: differentiële conversie in de freerunningmodus . . 351
20.11 Kalibratie van de ADC 351
20.12 Resumé ADC 352

21 Seriële communicatie 353

21.1 SPI 354
21.2 Toepassing: aansturing van een extern EEPROM via de SPI . . 357
21.3 Toepassing: aansturing van een schuifregister via een SPI 359
21.4 De USART als SPI 362
21.5 I^2C 364
21.6 I^2C of TWI voor de Xmega 366
21.7 Eenvoudige I2C-bibliotheek voor de Xmega in mastermodus . 367
21.8 Toepassing: eenvoudige I2C-bibliotheek bij een DS3232 370
21.9 Levelshifting voor I2C 374
21.10 De TWI-masterdriver van Atmel 375
21.11 De TWI-slavedriver van Atmel 378
21.12 Resumé TWI 382

22 Pulsbreedtemodulatie 383

22.1 De timer/counters van de Xmega 385
22.2 Bespreking PWM-mogelijkheden 387
22.3 De single-slope-modus: intensiteitsregeling voor een led 394
22.4 De single-slope-modus: intensiteitsregeling voor een rgb-led . . 395
22.5 De dual-slope-modus: een robotwagen met DC-motoren 397

22.6 De dual-slope-modus: aansturing servomotor 401
22.7 AWeX: advanced waveform extension 403
22.8 De frequentiemodus: het afspelen van muziek 404

23 Nog meer Xmega 409

23.1 Digitaal-analoogconverter 410
23.2 Direct Memory Access 415
23.3 Analoge comparator 420
23.4 Input capture 427
23.5 Het kloksysteem van de Xmega 435
23.6 De realtime-counter 439
23.7 Het EEPROM 441
23.8 Flash 445
23.9 De slaapstanden 448
23.10 De mogelijkheden om de Xmega256a3u te herstarten 451
23.11 Watchdog 452
23.12 Het atomic block 455

Bijlagen

A Stroomdiagrammen 457

B RS232 463

C JTAG 469

D SD-kaart 473

E Draadloze module 479

F CMOS 489

G Headerbestanden 503

H Application notes 511

I ASCII 513

J Xmega-bord 515

Index 519

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.

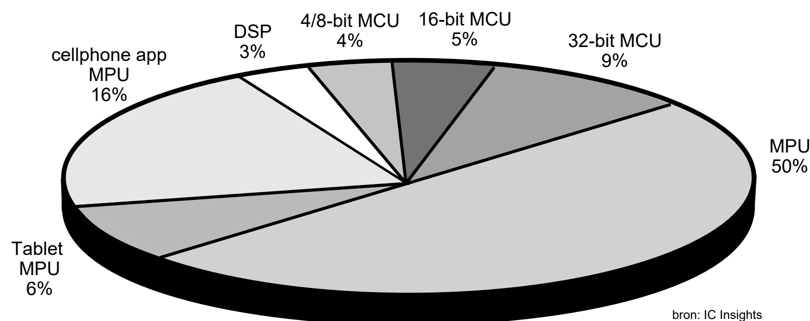
1.1 Embedded Systemen

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.

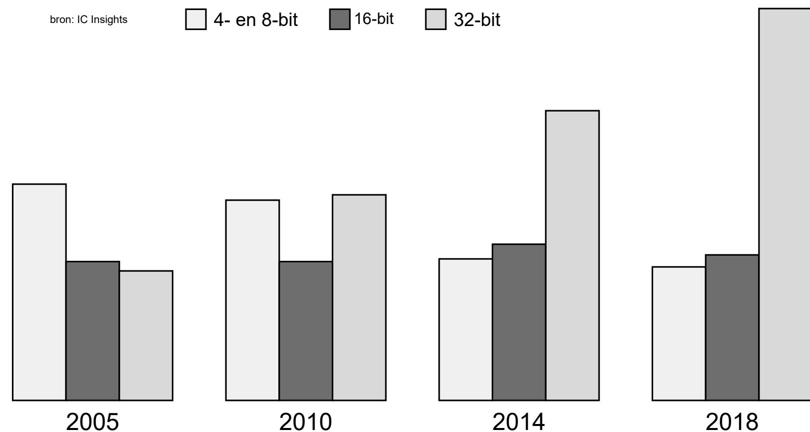
Er zijn tientallen verschillende definities van een embedded systeem. Een hele fraaie definitie is deze: ‘Een embedded systeem is een intelligent systeem dat er niet uit ziet als een computer! Dus zonder muis, beeldscherm en toetsenbord.’.

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 : Prognose wereldwijde omzet van processoren in 2014.
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 een



Figuur 1.3 : De wereldwijde omzet van microcontrollers.
De diagrammen geven de omzet van microcontrollers van 4/8-, 16-, en 32-bits in 2005 en 2010 en een voorspelling voor 2014 en 2018.

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.

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

[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 8-bit microcontroller. Tabel 1.2 vergelijkt een gewone notebook met een Xmega256a3u microcontroller van Atmel. 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 harde realtime-systemen.

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.

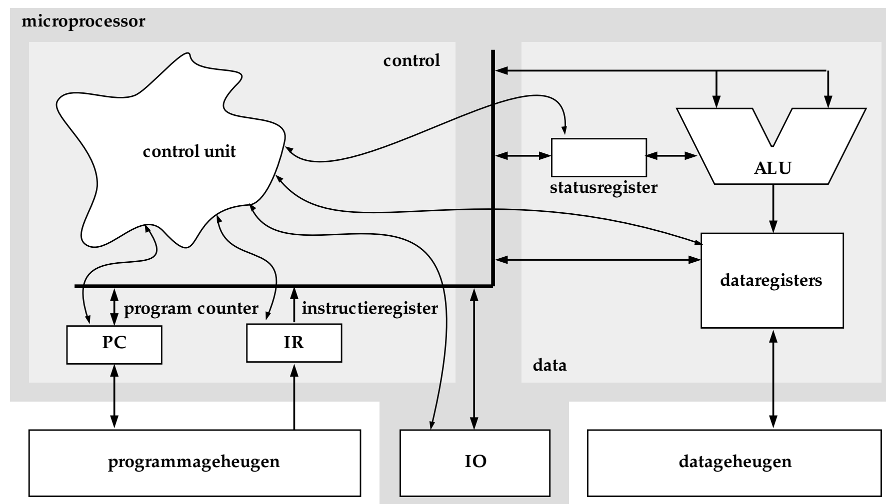
Tabel 1.2 : Een vergelijking tussen een pc en een microcontroller.

	notebook intel i5	**Xmega256a3u**
programmageheugen	8 GB instructies in cache, 500 GB harde schijf	256 KB bytes flash
kloksnelheid	2,8 GHz	32 MHz
rekengeheugen	8 GB RAM	16 KB RAM, 4 KB data EEPROM
processor	64 bits, quad, 2,6 GHz, 4 GMIPS, 7,5 GFLOPS	8 bits, 32 MHz, 32 MIPS, 50 kFLOPS [1]
randapparatuur	TFT, toetsenbord, muis, VGA, USB, HDMI, firewire, RS232, TCP/IP, SD-card, luidsprekers, microfoon	50 IO-pinnen
stroomverbruik	20 W	< 0,5 W [1]
omvang	37 × 25 × 3,5 cm	10 × 10 × 2 mm
gewicht	2,5 kg	< 1 g
prijs	€ 800	€ 3,5 [1]

[1] Dit zijn schattingen. Het aantal FLOPS is niet exact bekend. Het stroomverbruik is afhankelijk van het gebruik en van de slaapstanden. De prijs hangt sterk af van het aantal exemplaren.

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.

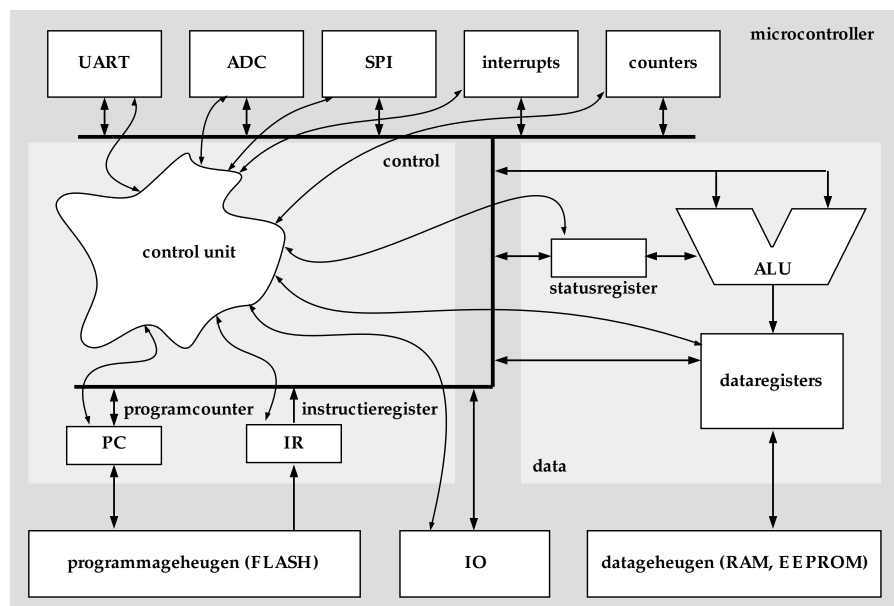
Figuur 1.4 toont de architectuur van een microprocessor. Het hart van de microprocessor is de centrale rekeneenheid — ALU (*Arithmetic Logic Unit*) — die

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.

RAM staat voor *Random Access Memory*.

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 in 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.



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.

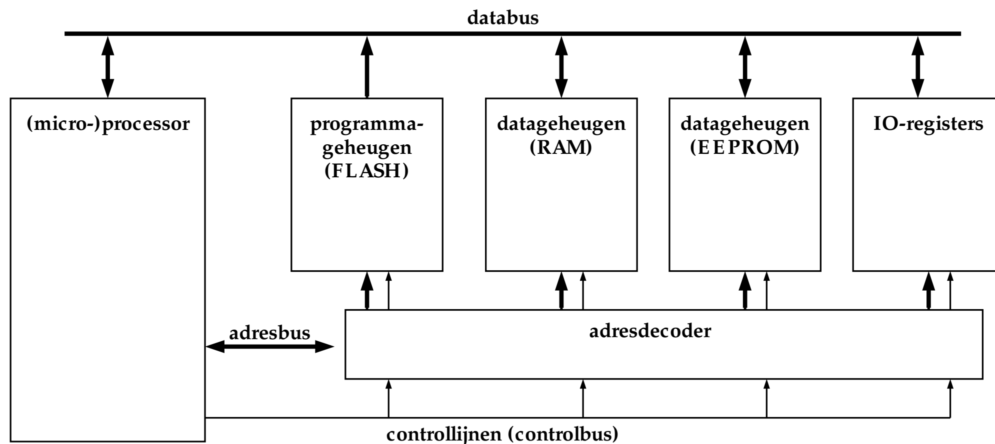
ROM staat voor *Read Only Memory*. Dat is geheugen dat alleen uitgelezen kan worden. De ontwikkelaar kan daar niet schrijven. De data worden in de chipfabriek er ingezet. Een PROM (*Programmable Read Only Memory* is een ROM die de productontwikkelaar met een programmer eenmalig kan programmeren. Een 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	PIC18F4682	Xmega256a3u
flash	programma	80 KB	256 KB
RAM	vluchtige data	3328 bytes	16384 bytes
EEPROM	niet-vluchtige data	1024 bytes	4096 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.

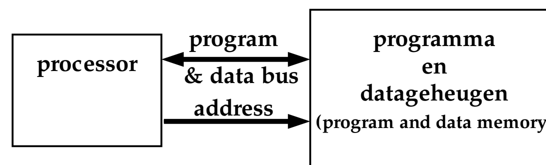


Figuur 1.6 : Een algemeen schema voor de adressering. Er zijn diverse geheugens en registers die geadresseerd worden via een adresbus. De adresdecoder zet het adres om naar een selectie van het juiste geheugendeel. De gegevens gaan via de databus van en naar de processor. De geheugens en de adresdecoder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

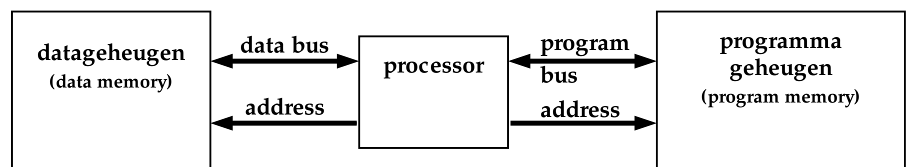
1.4 Harvard-architectuur

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.

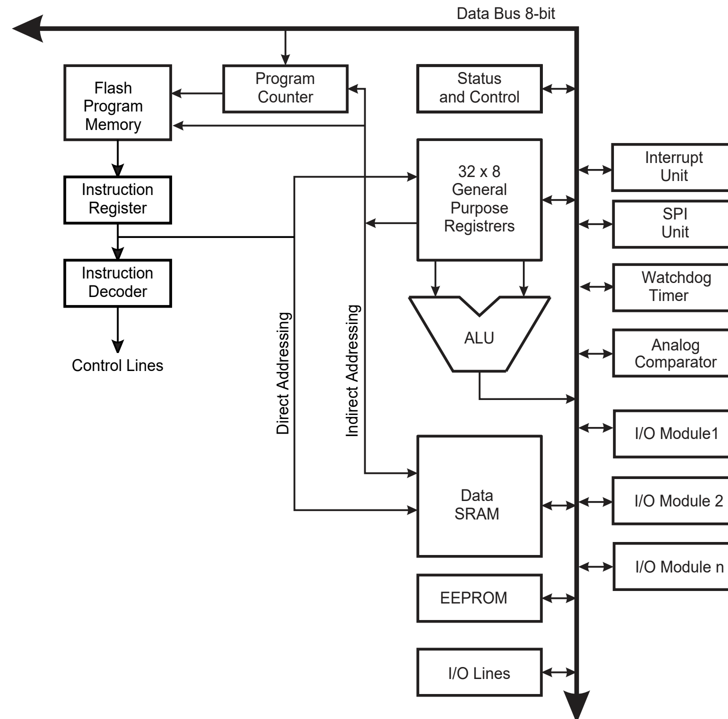
De Harvard-architectuur is getekend in figuur 1.8 en is ontwikkeld op de universiteit van Harvard. Deze architectuur heeft een aparte programmabus en aparte databus.



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 algemene architectuur van een AVR-microcontroller.

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 modi	Veel instructies en modi
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, zoals bijvoorbeeld de manier waarop de microcontroller geprogrammeerd wordt;
- 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.

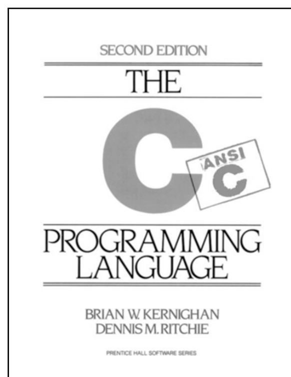
Dit boek is gebaseerd op de Xmega, dat is een AVR-microcontroller van Atmel. De architectuur van deze 8-bits microcontroller wordt aangeduid met AVR en is in 1992 bedacht door twee studenten van het Norwegian Institute of Technology (Norges Tekniske Høgskole). Deze studenten, Alf-Egil Bogen en Vegard Wollan, zijn na hun studie bij Atmel Norway gaan werken. Alf-Egil Bogen werkt inmiddels bij Novelda AS.

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. Atmel Studio van Atmel — in combinatie met de AVR GNU C-Compiler — is een gratis ontwikkelomgeving van hoog niveau, die zeer geschikt is voor het ontwikkelen van embedded systemen met de taal C.

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: ▪ Het belang van de taal C. ▪ Het voorbeeld 'Hello World' van Kernighan en Ritchie. ▪ Het compilatietraject. Voorbeelden van programma's in C zijn: ▪ Hello World. ▪ 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.
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.

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

Code 2.1 : Hello World.

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

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.

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 headerbestanden 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. Paragraaf 3.5 legt de functie `printf` kort uit en paragraaf 5.1 geeft een uitgebreide toelichting over de mogelijkheden.

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.

Code 2.1 is volgens de ISO C90 standaard opgeschreven. Dit kan nog eenvoudiger worden opgeschreven. Code 2.2 geeft de code van *Hello World!* uit de eerste druk van het boek van Kernighan en Ritchie.

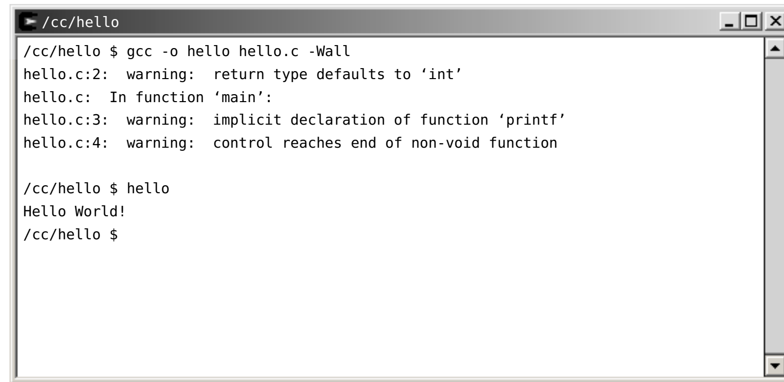
Code 2.2 : Oorspronkelijke niet-ANSI versie van Hello World.

```
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 2.2 wordt verder niet meer gebruikt. Bij de voorbeelden worden ook altijd alle include-bestanden genoemd.



Figuur 2.3 : Alle waarschuwingen bij compilatie van code 2.2 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 2.2. 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.

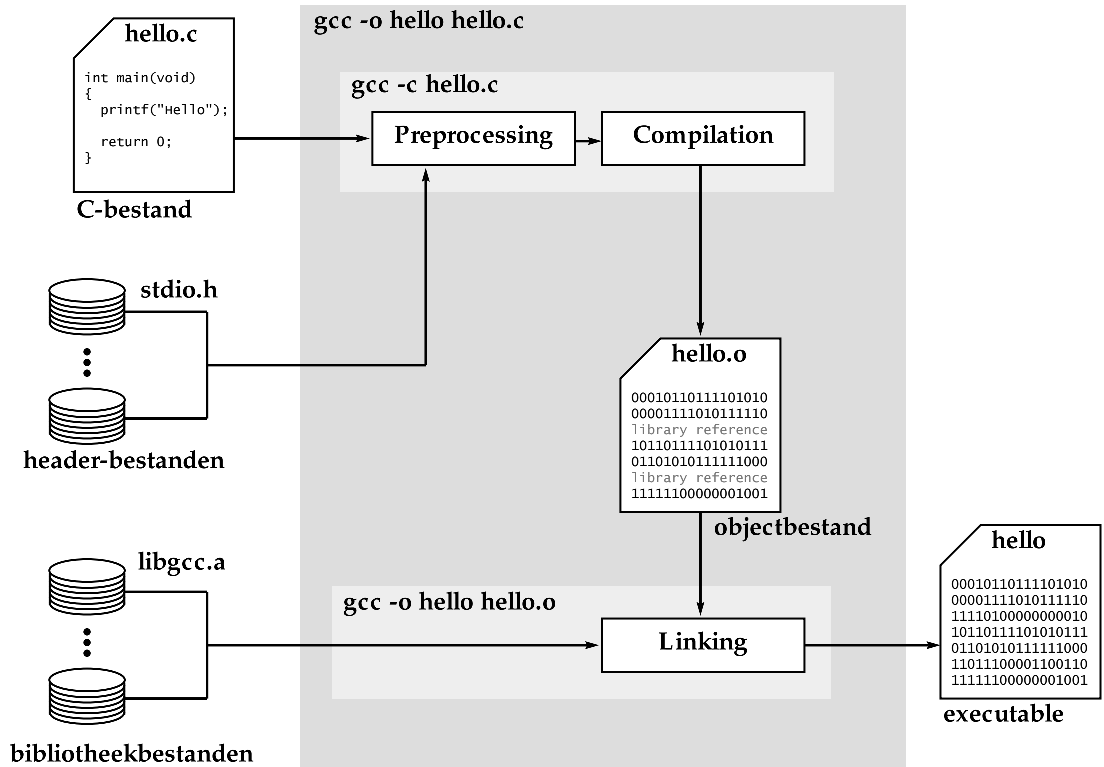
2.2 Het compilatietraject

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 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.

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.

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.



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.

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

De compilatie en het linken kunnen ook apart uitgevoerd worden. Met de 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.

```
/cc/hello
/cc/hello $ gcc -c hello.c

/cc/hello $ ls
hello.c hello.o

/cc/hello $ gcc -o hello hello.o

/cc/hello $ ls
hello.c hello.exe hello.o
```

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

2.3 Compilers

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.

Bij 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 iedere Unix- en Linux-distributie zit een C-compiler. Voor Windows bestaan heel veel C-compilers. Er zijn commerciële pakketten, zoals Microsoft Visual Studio, maar ook freeware oplossingen.
Een belangrijke C-compiler 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 paragraaf 13.2 behandeld.

Cygwin en MinGW

Er bestaan verschillende implementaties van de GNU C-Compiler voor Windows. Twee belangrijke zijn Cygwin en MinGW.

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

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

MinGW, Minimal GNU for Windows is een goed alternatief. De compiler kan vanuit het standaard commandovenster van Windows (*command shell window*) als vanuit een eigen commandovenster worden aangeroepen.

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.

Figuur 2.6 : De Msys-shell van MinGW.

Het belangrijkste verschil tussen Cygwin en MinGW is dat Cygwin een Unix-achtige omgeving is en MinGW een Windowsomgeving is. Programma's, die gemaakt zijn met Cygwin, kunnen niet direct in de Windowsomgeving gebruikt worden. Programma's, die gemaakt zijn met MinGW, kunnen dat wel.

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.

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 (Win-

dows) voor een andere systeem (ARM-processor) heet crosscompilen. De ontwikkelomgeving Atmel Studio voor het maken van programma's voor de Atmel AVR-microcontrollers gebruikt een aangepaste GNU C-Compiler als crosscompiler.
Dit boek gebruikt tot en met hoofdstuk 13 de Cygwin-omgeving voor het maken van Windows-toepassingen. In de overige hoofdstukken wordt Atmel Studio gebruikt voor het maken van applicaties voor de Atmel microcontroller.

2.4 Foutmeldingen

Bij het schrijven van een programma kunnen fouten worden gemaakt. Er zijn drie soorten fouten te onderscheiden:

- compilatiefouten,
- *linker*-fouten,
- runtime-fouten.

Compilatiefouten treden op wanneer de compiler de C-code niet begrijpt. Als bijvoorbeeld op regel 5 in code 2.1 de puntkomma ontbreekt, meldt de compiler:

```
hello.c: In function ‘main’:
hello.c:7:3: error: expected ‘;’ before ‘return’
   return 0;
   ^
```

De compiler zegt dan dat er in de functie `main` een fout is gevonden op regel 7. De compiler begrijpt de **`return`** op deze regel niet; hij had een puntkomma verwacht. Compilatiefouten staan vaak een of meer regels voor de regel, die genoemd wordt in de foutmelding.

Linker-fouten treden op als er met meerdere C-bestanden wordt gewerkt. Men vergeet bij het compileren een bestand mee te geven, waardoor bijvoorbeeld bij het linken de functie `print_age` ontbreekt:

```
/tmp/ccmAO6bk.o:main.c:(.text+0x17): undefined reference to ‘print_age’
collect2: error: ld returned 1 exit status
```

De mededelingen van de linker zijn te herkennen aan de teksten *undefined reference* en *error: ld returned 1 exit status*.

Nadat het programma gecompileerd en gelinkt is, kan het programma nog steeds fouten bevatten. Bij het runnen kan het programma dan crashen. Deze runtimefouten zijn vaak lastig te vinden. Bij goed leesbare en goed gestructureerde code is de kans op dit soort fouten veel kleiner.

3

C in het kort

Doelstelling

Dit hoofdstuk vertelt een aantal basisaspecten uit de taal C. Je maakt kennis met eenvoudige datatypen, bewerkingen en besturingsconstructies. Met deze informatie moet je in staat zijn zelf eenvoudige C-programma's te maken.

Onderwerpen

De behandelde onderwerpen zijn:

- Variabelen.
- Datatypen en typedeclaraties.
- Initialisatie.
- De basis datatypen `char`, `int`, `float`, `double`
- Eenvoudige rekenkundige bewerkingen.
- Array's.
- Het begrip string en de *end-of-string*.
- Voorwaardelijke opdrachten: `if`-statement, `else`-statement.
- Herhalingsopdrachten: `for`-lus, `while`-lus.

Voorbeelden van programma's in C zijn:

- De som van twee getallen
- Twee keer een som van twee getallen
- Hello World met strings
- De som van een aantal opeenvolgende even getallen
- De som en het gemiddelde van een array met getallen
- De som en het gemiddelde van een rij getallen met een afsluitteken

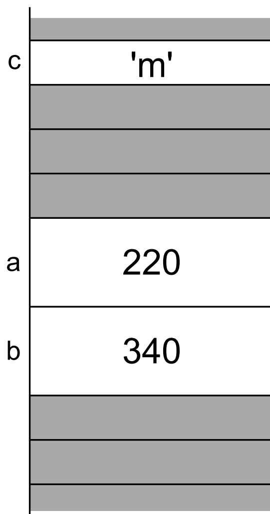
Hoewel C geen omvangrijke taal is, hoeft niet alles van deze taal bekend te zijn om toch een programma in C te kunnen maken. Dit hoofdstuk behandelt de belangrijkste aspecten van de taal C, die nodig zijn om zelfstandig een programma te kunnen maken. Dit hoofdstuk vertelt kort iets over eenvoudige datatypen, de bewerkingen die er met deze datatypen gedaan kunnen worden en de belangrijkste besturingsopdrachten.

In volgende hoofdstukken komen dezelfde onderwerpen opnieuw, maar dan uitgebreider en vollediger, aan bod — aangevuld met een groot aantal nieuwe onderwerpen.

3.1 Variabelen, declaraties en initialisatie

Net als de meeste andere programmeertalen kent C variabelen. Een variabele is de symbolische naam voor een stuk uit het computergeheugen waar het programma gegevens kan opslaan. In C bestaat een declaratie uit een datatype en de namen van één of meerdere variabelen.

```
char c;
int  a, b;
```



Figuur 3.1 : De indeling van het geheugen.

Variabele `c` is van het type **`char`** en de variabelen `a` en `b` zijn van het type **`int`**. Een **`int`** is bedoeld voor gehele getallen en een **`char`** voor het opslaan van karakters.

Aan iedere variabele kan een waarde worden toegekend. Tijdens de uitvoering van het programma kunnen deze waarden veranderen. Dat is dan ook de reden dat variabelen zo genoemd worden. Een toekenning of toewijzing wordt in C met het isgelijkteken gedaan.

```
c = 'm';
a = 220;
b = 340;
```

Na deze toekenningen heeft `c` de waarde `'m'` en variabelen `a` en `b` hebben dan respectievelijk de waarde 220 en 340.

In code 3.1 staat een programma dat de som berekent van twee getallen. Op regel 5 zijn twee variabelen `a` en `b` gedeclareerd. Op regel 6 staat de declaratie van de variabele `sum`. Vanaf regel 8 krijgt `a` de waarde 220, `b` de waarde 340 en aan `sum` wordt de som van de variabelen `a` en `b` toegekend. Op regel 12 wordt het resultaat van de berekening afgedrukt.

Code 3.1 : De som van twee getallen.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b;
6    int sum;
7
8    a = 220;
9    b = 340;
10   sum = a + b;
11
12   printf("De som van %d en %d is %d.", a, b, sum);
13
14   return 0;
15 }
```

Bij het afdrukken is een zogenoemde *format string* gebruikt met drie *format specifiers* of plaatsvervangers. Voordat `printf` de string afdrukt, wordt eerst op de plaatsen waar in de string `%d` staat, de waarden van de variabelen `a`, `b`, en `sum` ingevuld, zie ook figuur 3.2.

```
printf("De som van %d en %d is %d.", a, b, sum);
```

Figuur 3.2 : Het gebruik van *format specifiers* bij `printf`. Op de plaats van de *format specifiers* worden in de formatstring de waarden van de variabelen `a`, `b` en `sum` ingevuld.

De eerste variabele na de *format string* is `a`. De waarde van deze variabele komt op de plaats van de eerste `%d` in de string te staan. De waarde van variabele `b` komt op de plaats van de tweede plaatsvervanger en de waarde van `sum` op de plaats van de derde plaatsvervanger te staan. Als `a` gelijk is aan 220, `b` gelijk aan 340, is `sum` gelijk aan 560 en drukt het programma deze string af:

```
De som van 220 en 340 is 560.
```

In code 3.1 is aan de variabelen `a` en `b` eenmalig een waarde toegekend. Dit kan ook direct bij de declaratie van `a` en `b` gedaan worden.

```
int a = 220;
int b = 340;
```

In de meeste gevallen zullen de variabelen tijdens de uitvoering van het programma van waarde veranderen. In code 3.2 wordt twee keer een som van twee getallen afgedrukt. De waarde van de variabelen `a` en `b` krijgen nu na het afdrukken van de eerste som een andere waarde, waarna er weer een som wordt afgedrukt.

Code 3.2 : Twee keer een som van twee getallen.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b;
6    int sum;
7
8    a = 220;
9    b = 340;
10   sum = a + b;
11   printf("De som van %d en %d is %d.\n", a, b, sum);
12
13   a = 350;
14   b = 640;
15   sum = a + b;
16   printf("De som van %d en %d is %d.\n", a, b, sum);
17
18   return 0;
19 }
```

Op regel 11 is `\n` aan de *format string* van de functie `printf` toegevoegd. Het teken `\n` is een zogenoemde *escape sequence* en geeft het einde van de regel aan. Het resultaat van de tweede som komt dan op de volgende regel te staan:

```
De som van 220 en 340 is 560.
De som van 350 en 640 is 990.
```

Omdat er voor een einde van de regel, een tab en een backspace geen karakter bestaat, worden deze tekens gerepresenteerd door een *escape sequence* of *escape character*. Het einde van de regel wordt aangegeven met `\n` , de tab met `\t` en een backspace met `\b`.

In tabel 9.5 op pagina 106 staat een overzicht met alle *escape sequences*.

3.2 Datatypen

Zoals aan het begin van paragraaf 3.1 is vermeld, is een variabele een symbolische naam voor een stuk uit het computergeheugen. Een declaratie legt vast hoeveel ruimte de compiler voor een variabele in het geheugen moet reserveren en hoe de enen en nullen van de variabele worden geïnterpreteerd.

C kent een beperkt aantal standaard datatypen, namelijk: **`char`**, **`int`**, **`float`** en **`double`**. Een geheel getal `i`, een karakter `c` en de gebroken getallen `f`, `d1` en `d2` worden als volgt gedeclareerd:

```
int    i;
char   c;
float  f;
double d1, d2;
```

Aan deze variabelen worden daarna in een programma een geheel getal, een karakter en drie gebroken getallen toegekend:

```
i  = 2013;
c  = 'b';
f  = 4.2;
d1 = 4.2e+03;
d2 = 3.14;
```

De **`char`** wordt gebruikt voor het vastleggen van karakters. Het is een achtbits getal, dat een ASCII-waarde kan voorstellen. De waarde 65 is bijvoorbeeld het karakter `'A'` en 97 is een `'a'`. In C wordt het type **`char`** geïnterpreteerd als een 8-bits getal. Deze twee toekenningen zijn gelijkwaardig:

```
c = 'b';
c = 98;
```

Variabelen van het type **`char`** zijn 8-bits getallen en kunnen de ASCII-waarde van een karakter representeren. Tabel I.1 uit bijlage I geeft de verschillende karakters met de bijbehorende ASCII-waarden.

Een *byte* is een groep van acht bits.

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 Xmega is een **`int`** twee bytes groot en is het bereik −32768 tot 32767.

De standaardtypen **`float`** en **`double`** representeren gebroken getallen, zoals: 1,23; 50,6; 0,00338; 6,0 en $4{,}83\ 10^{13}$. Meestal is een **`float`** 32-bits en een **`double`** 64-bits. Als `x` een **`float`** of **`double`** is, zijn er verschillende notaties om aan `x` een gebroken getal toe te kennen:

```
x = 1.23;
x = 5034.6;
x = 0.00338;
x = 3.38e-3;
x = 4.83E+13;
```

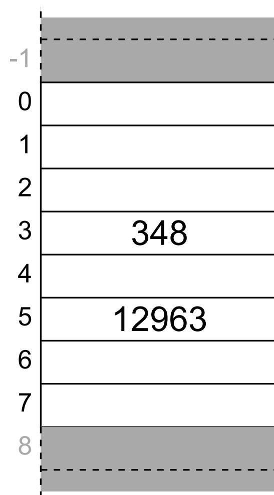
Naast de vier standaardtypen bestaan er een aantal varianten. Bij een `char` en een `int` geeft het sleutelwoord `unsigned` aan dat de waarden altijd positief zijn. Bij een `int` kan de afmeting worden gewijzigd met behulp van de sleutelwoorden `short`, `long` en `long long`. Een variabele van het type `long double` heeft twee keer zoveel bits als een `double`. In hoofdstuk 9 komen deze datatypen uitgebreider aan de orde.

3.3 Samengestelde datatypen: arrays en strings

De taal C kent ook samengestelde datatypen. Voorbeelden daarvan zijn de array en de string. Een array is een verzameling gegevens van hetzelfde type. Bij de arraydeclaratie wordt tussen rechte haken het aantal elementen vermeld:

```
int    x[8];
double d[30];
char   c[10];
```

Variabele `x` bestaat uit acht integers, variabele `d` uit 30 gebroken getallen en `c` uit tien karakters.



Figuur 3.3 : Het geheugen bij array `x`. Het getal 348 wordt in het hokje met index 3 geplaatst. Als `z` 12963 is, krijgt hokje 5 deze waarde.

De verschillende elementen worden gevonden met een index. Als variabelen `y` en `z` ook integers zijn, zijn dit geldige bewerkingen:

```
z    = 12963;
x[3] = 348;
y    = x[3];
x[5] = z;
```

Bij C begint de index altijd bij nul. De compiler controleert daar niet op. Er kan dus buiten de array worden geschreven en gelezen. Hoofdstuk 10 bespreekt uitgebreid de array's en licht dit aspect verder toe. Figuur 3.3 geeft de geheugenindeling van de array `x`.

De index mag ook een variabele zijn. In onderstaand voorbeeld krijgt het element met index 2 de waarde 300, die vervolgens wordt afgedrukt:

```
i = 2;
x[i] = 300;
printf("%d\n", x[i]);
```

Veel programmeertalen kennen een stringtype om tekst in op te slaan. Standaard kent C geen apart type string. 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'`. Dit sluitteken wordt de *end-of-string* genoemd. In C wordt een string gedeclareerd en geïnitialiseerd met:

```
char s1[6] = {'T', 'E', 'K', 'S', 'T', '\0'};
char s2[6] = "TEKST";
```

De strings `s1` en `s2` zijn helemaal identiek. Beide bevatten de vijf karakters van het woord *TEKST* met direct daarachter de *end-of-string*.

Als bij de declaratie een string direct een beginwaarde krijgt, is het niet nodig het aantal karakters te vermelden:

```
char s3[] = {'T', 'E', 'K', 'S', 'T', '\0'};
char s4[] = "TEKST";
```

De compiler weet dat er in beide gevallen zes karakters nodig zijn om deze variabelen in op te slaan.

3.4 Rekenkundige bewerkingen

Code 3.1 berekent de som van twee gehele getallen. Voor het bepalen van de som is de rekenkundige bewerking optellen gebruikt. In het totaal kent C vijf rekenkundige bewerkingen: optellen, aftrekken, vermenigvuldigen, delen en de modulus. Tabel 3.1 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.

Tabel 3.1 : 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 deze eenvoudige rekenkundige bewerkingen kent C ook een bibliotheek met de wiskundige functies zoals de sinus, de cosinus, de tangens, de logaritme en de exponentiële functie. Tabel 9.3 in paragraaf 9.7 geeft een overzicht.

De *modulus operator* berekent de modulus van twee getallen. Bij C is dat de rest van de deling. Als 32 geheeltallig door 5 gedeeld wordt, blijft er 2 over. Uit de bewerking `32 % 5`, oftewel 32 modulus 5, komt dus 2. Voor negatieve getallen wijkt de definitie van de modulus in C af van de wiskundige definitie.

Als de modulus nul is, is het deeltal deelbaar door de deler. Daarom is bijvoorbeeld een getal even als de modulus 2 van het getal, oftewel `x % 2`, gelijk is aan nul.

Code 3.3 : Hello World met strings.

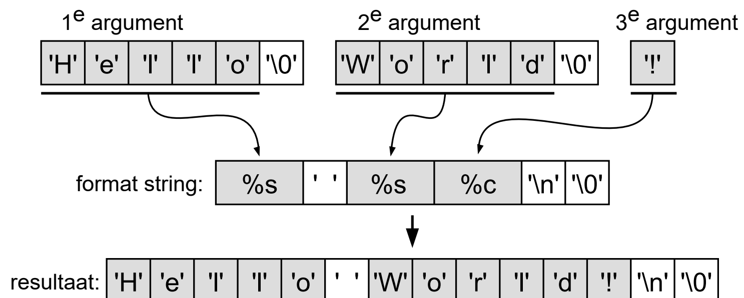
```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char s1[] = "Hello";
6    char s2[] = "World";
7    char c    = '!';
8
9    printf("%s %s%c\n", s1, s2, c);
10
11   return 0;
12 }
```

3.5 Afdrukken

Het eerste voorbeeld van dit boek, code 2.1 uit paragraaf 2.1, gebruikt de `printf`-functie om de tekst “Hello World!” af te drukken. Code 3.1 gebruikt deze functie om een tekst met daarin de waarde van de variabelen `a`, `b` en `sum` af te drukken. Het afdrukken van tekst en variabelen in een bepaald formaat wordt geformatteerd afdrukken genoemd.

De variabelen, die na de *format string* komen, worden ook wel de argumenten genoemd.

In code 3.3 staat een alternatieve versie van "Hello World!". Er zijn twee strings `s1` en `s2` gedeclareerd voor de woorden "Hello" en "World" en een **`char`** voor het uitroepteken.
De functie `printf` drukt de string `"%s %s%c\n"` af. Op de plaats van de `%s` wordt *Hello* ingevuld. Bij de tweede `%s` wordt *World* ingevuld en op de plaats van `%c` het uitroepteken. Figuur 3.4 laat zien hoe de argumenten in de *format string* worden ingevuld.



Figuur 3.4 : Het invoegen van de drie argumenten in de format string.

De `%s` en `%c` zijn *format specifiers* en worden ook wel plaatsvervangers genoemd. Omdat de betekenis van de enen en nullen bij ieder datatype anders is, kent ieder datatype een eigen plaatsvervanger.
In paragraaf 5.1 wordt het geformatteerd afdrukken met `printf` verder besproken. Tabel 5.1 in paragraaf 5.1 geeft de *format specifiers* voor de functie `printf`.

3.6 Voorwaardelijke opdrachten

Een programma bestaat uit een aantal opdrachten die na elkaar uitgevoerd worden. In veel gevallen zijn er handelingen, die — net als in dagelijks leven — soms wel en soms niet gedaan moeten worden. Een voorbeeld van een dergelijke beslissing is:

```
als de zon schijnt
  gaan we naar het strand
```

De voorwaarde om naar het strand te gaan, is dat de zon schijnt. In C zou dit er zo uit kunnen zien:

```
if (zon == 1) {
  printf("We gaan naar het strand\n");
}
```

Het **`if`**-statement is een voorwaardelijke opdracht. Tussen de ronde haken staat de voorwaarde en tussen de accolades staat wat er gedaan moet worden als aan de voorwaarde wordt voldaan. Het symbool == is de gelijkheidsoperator en het vergelijkt twee getallen met elkaar.

Een beslissing kan ook een alternatieve mogelijkheid hebben. Er kan aan toe worden gevoegd wat er gedaan wordt als de zon niet schijnt:

```
als de zon schijnt
  gaan we naar het strand
anders
  gaan we naar het museum
```

Het equivalent in C is:

```
if (zon == 1) {
  printf("We gaan naar het strand\n");
} else {
  printf("We gaan naar het museum\n");
}
```

Achter de **`else`** staat wat er gedaan wordt als de variabele `zon` ongelijk is aan 1.

Bij meerdere keuzes is een **`if`**-**`else`**-**`if`**-constructie nodig:

```
if (temperatuur > 30) {
  printf("We gaan buiten in schaduw zitten\n");
} else if (temperatuur >= 15 ) {
  printf("We gaan erop uit\n");
} else {
  printf("We blijven binnen en lezen we een boek\n");
}
```

De tekens `>` en `>=` vergelijken de temperatuur met respectievelijk `30` en 15 ^{o}C. Deze tekens zijn voorbeelden van relationele of vergelijkingsoperatoren. Tabel 3.2 geeft alle relationele operatoren. Deze operatoren kunnen alleen gebruikt worden bij datatype die een getal representeren, zoals **`char`**, **`int`**, **`float`** en **`double`**.

Tabel 3.2 : De zes relationele operatoren voor logische bewerkingen.

bewerking	wiskundig symbool	symbool in C
groter dan	$>$	`>`
groter dan of gelijk aan	$\geq$	`>=`
kleiner dan	$<$	`<`
kleiner dan of gelijk aan	$\leq$	`<=`
gelijk aan	$=$	`==`
ongelijk aan	$\neq$	`!=`

! Verschil tussen == en =

Verwar bij de taal C het symbool == niet met het symbool =. Het =-teken is altijd een toekenning. Het ==-teken is in C een vergelijking. Het resultaat is `0` als de bewerking niet waar is en ongelijk aan `0` als de bewerking waar is.

De compiler geeft *geen* foutmelding bij de uitdrukking **`if`** `( zon = 1 ) {}`. De conditie is dan altijd 1 en is dus altijd waar en het **`if`**-statement drukt altijd af dat we naar het strand gaan.

3.7 Herhalingsopdrachten

Naast de voorwaardelijke opdrachten zijn er handelingen, die — net als in dagelijks leven — herhaald uitgevoerd moeten worden. Een voorbeeld is:

```
zolang er nog klanten zijn
  krijgt de volgende klant een ijsje
```

In C kan dat er als volgt uit zien:

```
aantalklanten = 8;
while (aantalklanten > 0) {
  printf("%s\n", "Hier is een ijsje.\n");
  aantalklanten = aantalklanten - 1;
}
printf("Alle klanten hebben nu een ijsje.");
```

Het **`while`**-statement is een herhalingsopdracht. Tussen de ronde haken staat de voorwaarde en tussen de accolades staat wat er gedaan moet worden zolang er aan de voorwaarde voldaan wordt.

Een andere herhalingsopdracht is het **`for`**-statement. In dit voorbeeld worden alle cijfers van `0` tot en met 9 afgedrukt.

```
printf("Dit zijn alle cijfers: ");
for (i=0; i<=9; i++) {
  printf("%d", i);
}
printf("\n");
```

De toekenning `i++` is een verkorte schrijfwijze van `i=i+1`.
`++` is de increment-operator.

De code tussen de accolades wordt herhaald uitgevoerd. Tussen de ronde haken staan drie statements. Het eerste statement is de beginconditie `i=0;`, in dit geval krijgt `i` de waarde `0`. Het tweede is de stopconditie. Zolang `i` kleiner of gelijk aan 9 is, wordt de code tussen de accolades uitgevoerd. Het derde statement is de iteratie. Iedere keer dat de code wordt uitgevoerd, wordt de variabele `i` met één opgehoogd.

`aantalklanten--` is een verkorte schrijfwijze voor: `aantalklanten = aantalklanten - 1`.
`--` is de decrement-operator.

Een **`for`**-lus kan altijd door een **`while`** en een **`while`**-lus kan vaak door een **`for`** worden vervangen. De **`while`** aan het begin van deze paragraaf kan ook met een **`for`** worden opgeschreven:

```
for ( aantalklanten = 8; aantalklanten > 0; aantalklanten-- ) {
  printf("%s\n", "Hier is een ijsje.\n");
}
printf("Alle klanten hebben nu een ijsje.");
```

De **`for`**-lus wordt vooral toegepast bij situaties waarbij het aantal iteraties vooraf bekend is. De **`while`**-lus is handig als het aantal iteraties vooraf onbekend is.

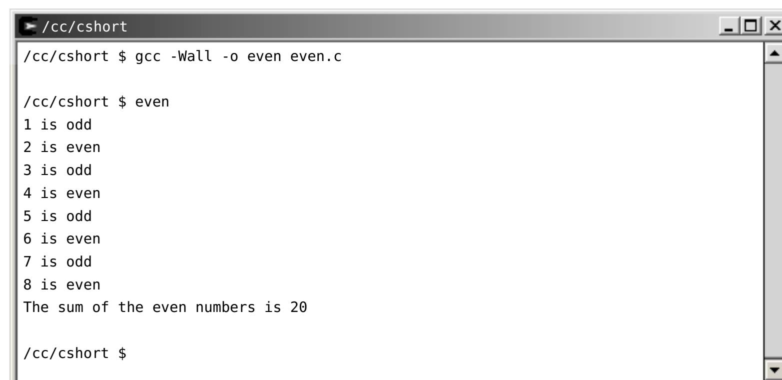
3.8 Voorbeelden

Voorwaardelijke en herhalingsopdrachten mogen ook door elkaar worden gebruikt. In code 3.4 staat een programma dat voor opeenvolgende getallen afdrukt of het getal even of oneven is. Bovendien drukt het programma de som van de even getallen af.

Code 3.4 : De som van een aantal opeenvolgende even getallen.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int i;
6    int sum_even  = 0;
7
8    for (i=1; i<=8; i++) {
9      if (i % 2) {
10       printf("%d is odd\n", i);
11     } else {
12       printf("%d is even\n", i);
13       sum_even = sum_even + i;
14     }
15   }
16   printf("The sum of the even numbers is %d\n", sum_even);
17
18   return 0;
19 }
```

Er wordt een `for`-lus gebruikt om alle getallen van 1 tot en met 8 te evalueren. Het `if`-statement met de modulus-operator test op regel 9 of `i` even of oneven is. De bewerking `i % 2` is 1 als `i` oneven en `0` als `i` even is. De uitvoer van het programma staat in figuur 3.5.



Figuur 3.5 : De uitvoer van code 3.4.

Een array is een verzameling gegevens van hetzelfde type. De elementen van een array kunnen bekeken worden door met een `for`-lus de index van de array te veranderen. In code 3.5 is een array met tien getallen gedeclareerd. De `for`-lus telt de waarden van alle getallen bij elkaar. De index `i` loopt van `0` tot en met 9 en voor iedere `i` wordt het betreffende getal uit de array bij de som opgeteld.

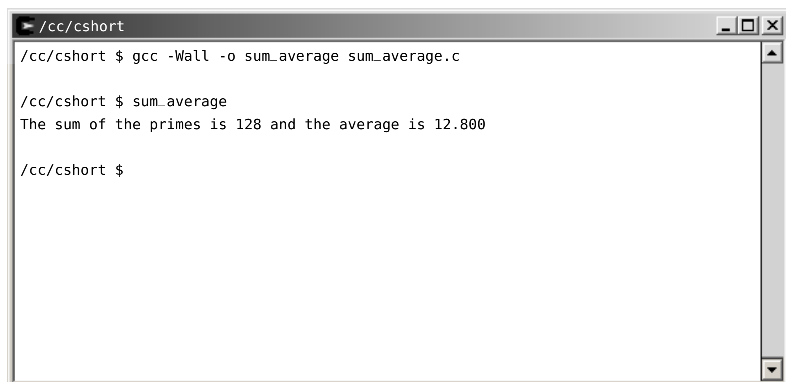
Code 3.5 : De som en het gemiddelde van een array met getallen.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int sum;
6    int i;
7    double average;
8    int array[] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29};
9
10   sum = 0;
11   for (i=0; i<10; i++) {
12     sum = sum + array[i];
13   }
14   average = (double) sum/i;
15
16   printf("The sum of the primes is %d ", sum);
17   printf("and the average is %.3f\n", average);
18
19   return 0;
20 }
```

Op regel 17 is na het procentteken `.3` aan de *format specifier* toegevoegd. Dit betekent dat het resultaat met drie cijfers achter de komma (punt) wordt afgedrukt. In paragraaf 5.1 wordt dit besproken.

Typecasting bij gebroken getallen wordt in paragraaf 9.4 besproken.

De variabele `average` is van het type **`double`** om ook de cijfers achter de komma te laten zien. Op regel 14 staat tussen haakjes het type **`double`**. Dit wordt een *typecasting* genoemd en is nodig omdat `sum` en `i` beide integers zijn. Het resultaat van `sum/i` is dan ook een integer en is in dit geval `12`. Met de *typecasting* wordt het een **`double`** en is het resultaat `12.800`, zoals figuur 3.6 laat zien.



Figuur 3.6 : De uitvoer van code 3.5.

Het nadeel van het programma uit code 3.5 is dat als er getallen aan de array worden toegevoegd, de stopconditie van de **`for`**-lus ook moet worden aangepast. Dat wordt dan gemakkelijk vergeten. Dit probleem kan worden opgelost door het aantal array-elementen te bepalen. Dat kan met behulp van de **`sizeof`**-operator, die in paragraaf 9.16 op bladzijde 116 aan de orde komt.

Code 3.6 : De som en het gemiddelde van een rij getallen met een afsluitteken.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int sum;
6    int i;
7    double average;
8    int array[] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29, -1};
9
10   sum = 0;
11   i   = 0;
12   while (array[i] != -1) {
13     sum = sum + array[i];
14     i   = i + 1;
15   }
16
17   if ( i==0 ) {
18     printf("The array is with primes is empty\n");
19     return 1;
20   }
21   average = (double) sum/i;
22
23   printf("The sum of the primes is %d ", sum);
24   printf("and the average is %.3f\n", average);
25
26   return 0;
27 }
```

De `-1` is een afsluitteken. Dit wordt in het Engels een *sentinel*, of schildwacht, genoemd.

Bij het aanleren van een programmeertaal zijn aan het begin veel mogelijkheden niet bekend. Een programmeur, die onbekend is met **`sizeof`**, kan dan op een heel andere oplossing komen. In code 3.6 staat een oplossing waarbij er aan de array een extra getal `-1` is toegevoegd. In plaats van een **`for`**-lus wordt in dit programma een **`while`** gebruikt. Het lezen van getallen uit de array stopt als de waarde `-1` gevonden is.

4

Functies

Doelstelling	In dit hoofdstuk leer je wat een functie is, waarom je functies moet gebruiken en hoe je in C een functie declareert en hoe je een functie aanroept.

Onderwerpen

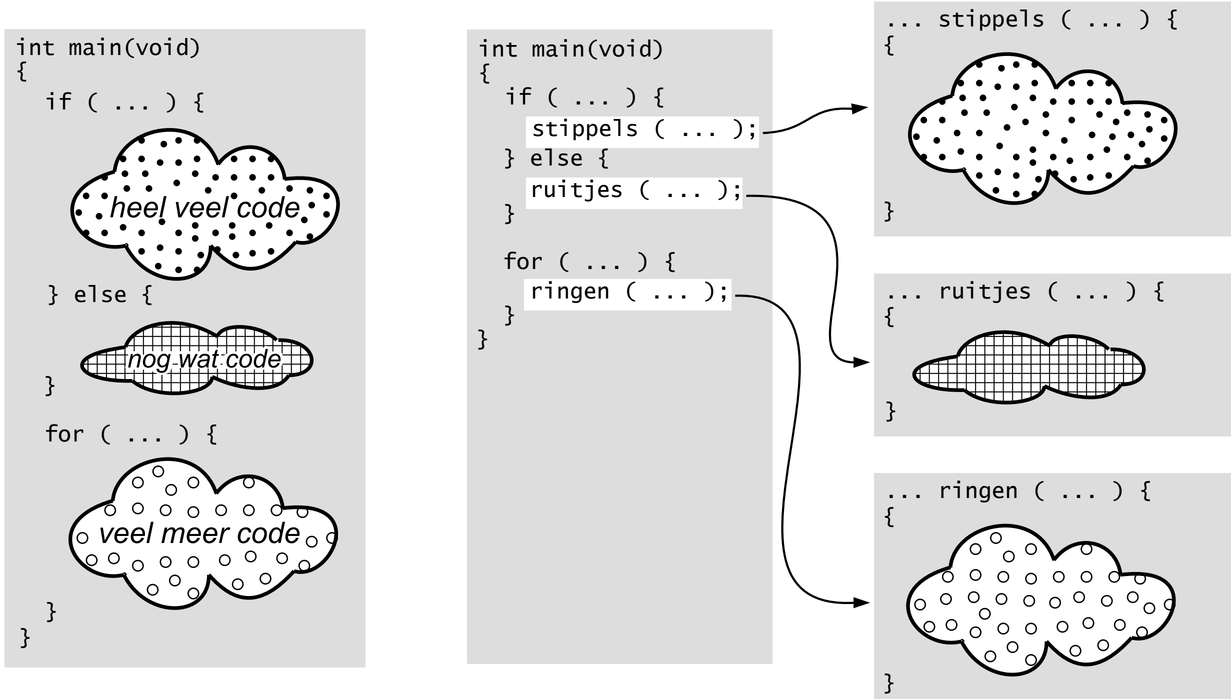
De behandelde onderwerpen zijn:

- Verdeel-en-heersstrategie.
- Hergebruik van code.
- Hiërarchisch ontwerpen.
- 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 `&`.
- Blokschema's, stroomdiagrammen, pseudocode en algoritmes.

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.
- Cijferprogramma met functies `getScore` en `showResult`.
- Het verschil tussen `==` en `=`.

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.



Figuur 4.1 : **Het opsplitsen van code in deelproblemen.** Links staat een voorbeeld van een hoofdprogramma dat een groot stuk code bestaat. In het midden staat het zelfde voorbeeld waarbij belangrijke delen zijn vervangen door drie functieaanroepen `stippels()`, `ruitjes()` en `ringen()`. Rechts staan deze drie functies, die ieder delen uit de originele code bevatten.

4.1 Verdeel en heers

Verdeel-en-heers is een strategie die bij programmeren wordt toegepast om de complexiteit van het ontwerp te beheersen. Deze strategie splitst het ontwerp in kleinere deelproblemen. Deze deelproblemen worden vervolgens apart opgelost. De deeloplossingen worden daarna gebruikt voor het oplossen van het originele probleem.

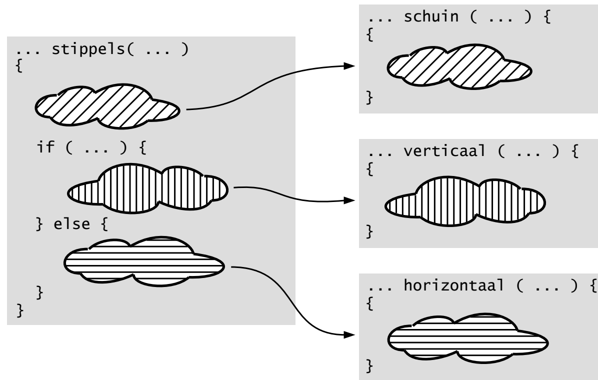
Verdeel-en-heers is een begrip dat uit de Oudheid komt: *divide et impera*. Het is de tactiek om tweedracht te zaaien tussen de vijanden.

Bij programmeren heeft verdeel-en-heers een iets andere betekenis. Het gaat hierbij om het opdelen van een ontwerp in kleine beheersbare onderdelen.

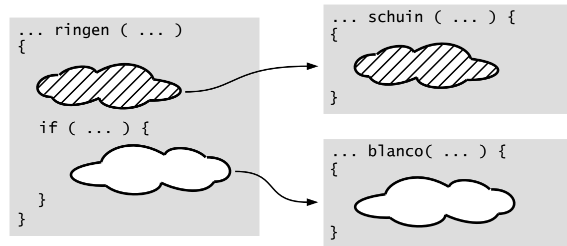
In figuur 4.1 staat links een schematisch voorbeeld van een complex programma. Alle code staat in de hoofdroutine `main`. Er zijn drie grote stukken code te onderscheiden. Deze codefragmenten kunnen ook vervangen worden door de functieaanroepen, `stippels()`, `ruitjes()` en `ringen()`. De drie functies bevatten dan de betreffende codefragmenten uit de originele, complexe beschrijving.

Het codefragment in de functie `stippels` bestaat nog steeds uit heel veel code. Figuur 4.2 toont een situatie waarbij de functie `stippels` gebruik maakt van drie functies. Mits de opdeling zinvol is, zullen de deelfuncties `stippels`, `ruitjes`, `ringen`, `schuin`, `horizontaal` en `verticaal` veel eenvoudiger te begrijpen zijn dan dezelfde code in het complexe hoofdprogramma.

Door de code te splitsen in deelproblemen wordt de code beter leesbaar, de complexiteit kleiner en beter beheersbaar. Een ander voordeel is dat de functies apart getest kunnen worden. Fouten worden dan eenvoudiger gevonden en de code kan



Figuur 4.2 : De functie `stippels` is opgesplitst in deelproblemen. De linker figuur laat zien dat de functie `stippels` uit drie delen bestaat. Rechts staan de drie bijbehorende functies `schuin`, `horizontaal` en `verticaal`.



Figuur 4.3 : De functie `ringen` is opgesplitst in deelproblemen. De linker figuur laat zien dat de functie `ringen` uit twee delen bestaat. Rechts staan de bijbehorende functies `schuin` en `blanco`.

systematisch worden getest. Bovendien wordt het eenvoudiger de code te documenteren. Bij iedere functie kan een commentaarblok worden toegevoegd met een korte beschrijving over wat de functie doet en hoe de functie gebruikt moet worden.

Hergebruik van code

Op een gelijke manier, zoals gedaan is bij de functie `stippels`, maakt de functie `ringen` in figuur 4.3 gebruik van twee functies `schuin` en `blanco`. De functie `schuin` wordt gebruikt in zowel de functie `stippels` als in de functie `ringen`. De functie `schuin` wordt dus opnieuw gebruikt.

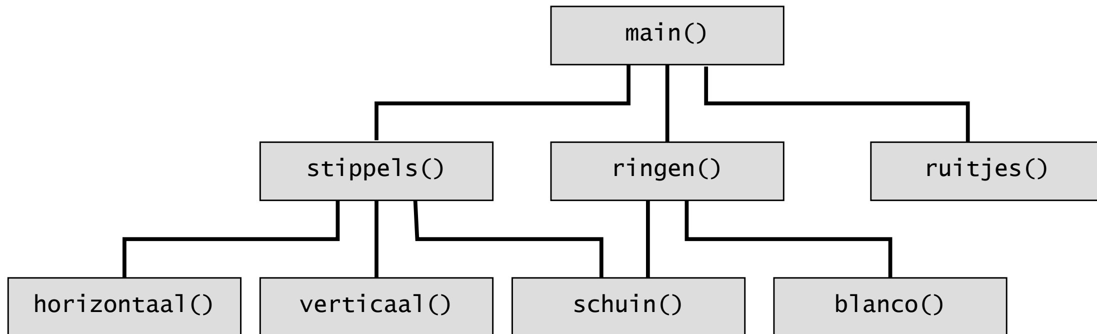
Naast de verbeterde beheersbaarheid en de leesbaarheid is hergebruik een andere belangrijke reden om de code op te splitsen in aparte functies. Hergebruik is heel belangrijk. De programmeur hoeft de functionaliteit van de functie `schuin` maar één keer op te schrijven.
In de originele, complexe beschrijving staat de functionaliteit van `schuin` op twee plaatsen: één keer bij het codefragment met de 'stippels' en één keer bij het deel met de 'ringen'.
Tijdens het ontwikkelen van de code is de kans groot dat deze codefragmenten van elkaar gaan afwijken. Een verbetering wordt bij het deel met de 'stippels' wel ingevoerd, maar bij de 'ringen' wordt het vergeten. Bij de gestructureerde aanpak met functies hoeven aanpassingen maar op één plaats te worden doorgevoerd, namelijk bij de functie.

Men spreekt in dit verband ook van het voorkomen van redundantie.
Redundant betekent overtollig. Er is meer gegeven dan strikt genomen nodig is.
Soms is redundantie nuttig, zo kan een beveiligingssysteem bewust dubbel uitgevoerd zijn. Maar bij software is redundante informatie vaak een bron van fouten.

Een ander voordeel is dat functies, die bij elkaar horen, samen kunnen worden geplaatst in aparte bibliotheken. Een voorbeeld is de standaardbibliotheek met onder andere de functies `printf`, `scanf` en `getchar`. Iedere programmeur kan deze functies gebruiken. Hij hoeft niet te weten hoe deze functies opgebouwd zijn. Hij hoeft alleen te weten hoe de functies gebruikt moeten worden.

Hiërarchie

Door het opsplitsen van het programma in verschillende functies ontstaat er een rangorde in het ontwerp. De hoofdroutine `main` gebruikt de functie `stippels`, `ruitjes` en `ringen`. De functies `stippels` en `ringen` maken op hun beurt weer gebruik van de functies `schuin`, `verticaal`, `horizontaal` en `blanco`.
Een ontwerp met een dergelijke rangorde noemt men een hiërarchisch ontwerp. Hiërarchie ontstaat automatisch bij de verdeel-en-heersstrategie. Een hiërarchie kan worden weergegeven met een structuur die op een boom lijkt. In figuur 4.4 staat de hiërarchie van het ontwerp.



Figuur 4.4 : De schematische weergave van de hiërarchie van het ontwerp.

Een ander voordeel van het opdelen van een ontwerp in functies is dat de implementatie van het ontwerp gedaan kan worden door meerdere ontwerpers. Programmeur A maakt bijvoorbeeld de functie `stippels` en de functies `horizontaal`, `verticaal` en `schuin`. Programmeur B maakt de functie `blanco` en gebruikt bij het maken van de functie `ringen` de functie `schuin`, die door programmeur A is gemaakt. Programmeur C ontwerpt functie `ruitjes` en gebruikt de functies `stippels` en `ringen` voor het hoofdprogramma.

Verdeel-en-heersstrategie

Het voorafgaande zou kunnen suggereren dat eerst het complexe programma wordt gemaakt en het daarna wordt gesplitst in functies. In het algemeen is dat niet de juiste aanpak.

Verdeel-en-heersstrategie is er op gebaseerd dat de ontwerper het ontwerp direct opdeelt in deelproblemen. De juiste werkwijze is:

1. splits het probleem in deelproblemen
2. los de deelproblemen afzonderlijk op
3. maak de totale oplossing met behulp van de deeloplossingen

Soms is het lastig om de juiste indeling te maken. Tijdens het ontwerpen blijkt een functie dan toch nog te complex en onbeheersbaar te worden. Het is daarom zeker niet verboden om op latere momenten de code opnieuw in te delen.

4.2 De opbouw van een functie

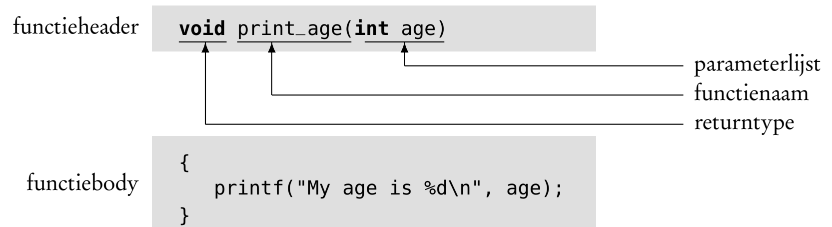
Een voorbeeld van een functie staat staat in code 4.1. De functie `print_age` drukt een tekst met de leeftijd af. Op regel 3 tot en met 6 staat de functiedeclaratie en

op regel 12 staat de functieaanroep. Het hoofdprogramma voert de functie uit en drukt, mits de waarde van variabele `age` gelijk aan 24 is, de tekst `"My age is 24"` af.

Code 4.1 : Afdrukken leeftijd met aparte afdrukfunctie.

```
 1  #include <stdio.h>
 2
 3  void print_age(int age)
 4  {
 5    printf("My age is %d\n", age);
 6  }
 7
 8  int  age=24;
 9
10  int main(void)
11  {
12    print_age(age);
13
14    return 0;
15  }
```

Een functiedefinitie, zie figuur 4.5, 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 `}`.



Figuur 4.5 : 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 `}`.

Tussen deze twee accolades staan de lokale declaraties en de statements van de functie. 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 12 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.

In code 4.2 komt de functiedeclaratie na `main`. Bij de aanroep op regel 8 zou de functie dan onbekend zijn en een waarschuwing geven. Met regel 3 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 4.2 : Afdrukken leeftijd met prototype.

```
1  #include <stdio.h>
2
3  void print_age(int age);
4  int  age=24;
5
6  int main(void)
7  {
8    print_age(age);
9    return 0;
10 }
11
12 void print_age(int age)
13 {
14   printf("My age is %d", age);
15 }
```

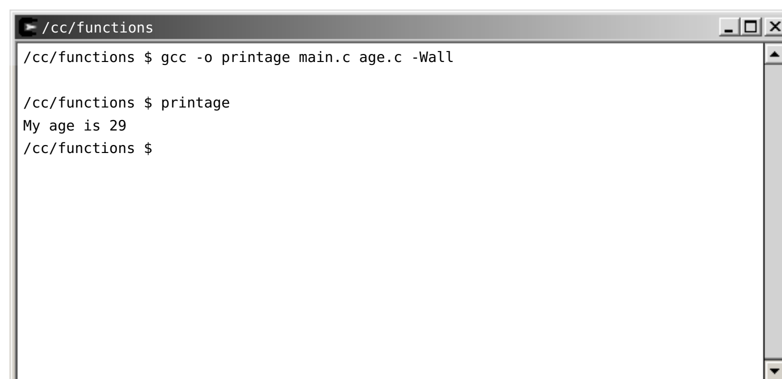
Code 4.3 : Code bestand main.c.

```
1  void print_age(int age);
2  int  age=29;
3
4  int main(void)
5  {
6    print_age(age);
7    return 0;
8  }
```

Code 4.4 : Code bestand age.c.

```
1  #include <stdio.h>
2
3  void print_age(int age)
4  {
5    printf("Your age is %d", age);
6  }
```

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 4.6 is gedaan.



Figuur 4.6 : 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.

```
1 #include "age.h"
2 int  age=29;
3
4 int main(void)
5 {
6   print_age(age);
7   return 0;
8 }
```

Code 4.6 : Code bestand age.h.

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

Code 4.7 : Code bestand age.c.

```
1 #include <stdio.h>
2
3 void print_age(int age)
4 {
5   printf("Your age is %d", age);
6 }
```

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 4.7 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 $ gcc -o printage main.o age.o -Wall

/cc/functions $ printage
My age is 29
/cc/functions $
```

Figuur 4.7 : De afzonderlijke compilatie van main.c en age.c.

4.3 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   vol = volume(length, w, 12);
15
16   printf("The volume is: %d", vol);
17   return 0;
18 }
```

formele parameters (l, w, h op regel 4)

actuele parameters (length, w, 12 op regel 14)

4.4 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

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
```

Annotaties in de marge: lokaal in functie print_age1() (regels 5–8); lokaal in prototype (regel 15); lokaal in block (regels 22–24); lokaal in functie main() (regels 19–33); lokaal in block (regels 39–41); lokaal in functie print_age3() (regels 35–43); globaal in bestand (regels 3–45).

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.
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\n", age);
20   get_age2(&age);
21   printf("My age is %d\n", age);
22
23   return 0;
24 }
```

4.5 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*.

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.

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.

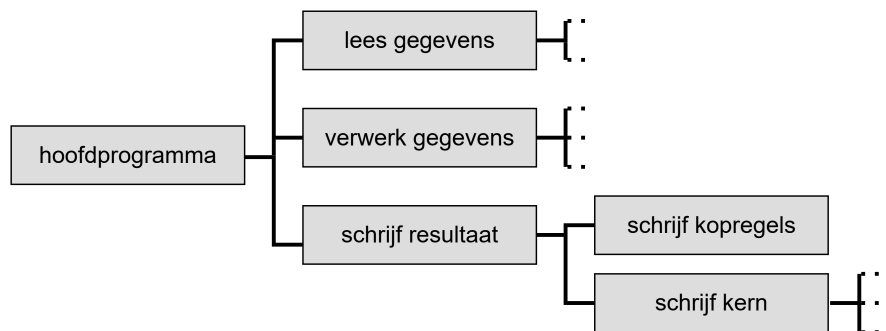
Op regel 9 staat in de parameterlijst van de functie `get_age2` een `int *ptr_age`. Dit is een pointer die naar een variabele van het type `int` wijst. Een pointer is een geheugenplaats waar het adres van een andere geheugenplaats kan staan. Bij de aanroep van de functie op regel 20 staat voor `age` 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 het effect dat `a` niet toegekend wordt aan `ptr_age`, maar ingevuld wordt op de geheugenplaats waar `ptr_age` naar wijst.
Pointers en pointerberekeningen zijn in C een belangrijk en krachtig mechanisme. In hoofdstuk 11 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.

4.6 Blokschema's, stroomdiagrammen, pseudocode en algoritmes

In het algemeen bestaat een computerprogramma uit meerdere onderdelen die allemaal een specifieke taak uitvoeren. Deze taken zijn meestal hiërarchisch opgebouwd. Een programma of programma-onderdeel dat bijvoorbeeld een bestand met meetgegevens verwerkt bestaat uit een deel dat de gegevens leest, een deel dat de gegevens converteert en een deel dat het resultaat naar een bestand schrijft. Het schrijven naar het bestand kent weer een aantal deeltaken, namelijk het schrijven van de kopregels en de kern.

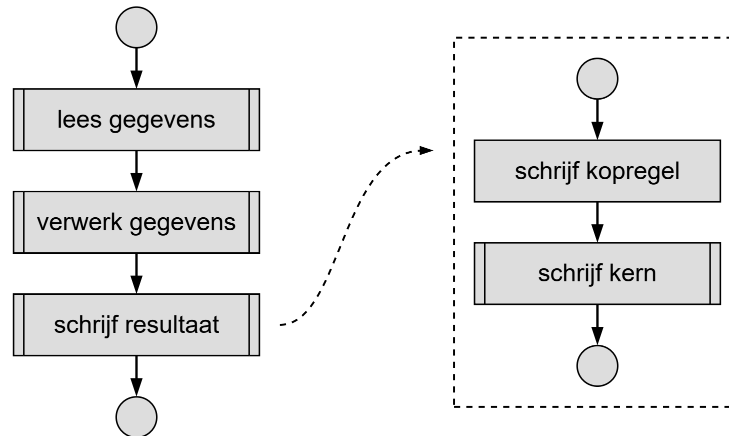


Figuur 4.8 : De hiërarchische opbouw van een programma dat meetgegevens verwerkt.

In figuur 4.8 staat de hiërarchische opbouw van het meetprogramma. Ieder blok uit het figuur is een functie, die weer uit meerdere functies kan bestaan.

Een blokschema geeft alleen de hiërarchische opbouw. Het zegt in principe niets over de volgorde waarin de subprogramma's of functies worden uitgevoerd. In figuur 4.9 staan de stroomdiagrammen voor het hoofdprogramma en een van de deelprogramma's.

De betekenis van de symbolen voor een stroomdiagram of een *flow chart* staat in bijlage A over stroomdiagrammen.



Figuur 4.9 : Stroomdiagrammen voor het programma dat meetgegevens verwerkt.

Een stroomdiagram is een grafische weergave van het recept hoe een bepaalde taak moet worden uitgevoerd. Een voorbeeld van een recept is het zetten van koffie met behulp van een koffiezetapparaat. Afhankelijk van het type apparaat zijn er verschillende procedures mogelijk. Dit is een mogelijk recept:

- spoel de koffiekan om en zet deze in het apparaat;
- pak een koffiefilterzakje en plaats deze in het koffiefilter;
- pak de koffie en schep de koffie in het filter;
- vul het waterreservoir met water;
- zet het koffiezetapparaat aan;
- wacht tot de koffie klaar is.

In de wiskunde en ook bij programmeren wordt een recept of een voorschrift om een bepaald probleem op te lossen een algoritme genoemd.
Algoritmes hangen vaak af van bepaalde gegevens. Bij het koffiezetten is dat bijvoorbeeld het aantal kopjes dat gezet moet worden. Het algoritme bevat dan herhalingsopdrachten als: doe acht keer een schepje koffie in het koffiefilter.

Pseudocode

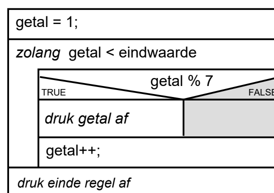
Het recept voor het koffie zetten is in gewone spreektaal opgeschreven. In principe kan dit ook met een stroomdiagram beschreven worden, maar dat levert geen belangrijke voordelen op. Ook bij het ontwikkelen van een algoritme van een computerprogramma is het vaak handig om het eerst in spreektaal op te schrijven.

Stel dat er een programma nodig is dat voor 1 tot en met een bepaalde eindwaarde alle getallen afdrukt die niet deelbaar zijn door 7. In spreektaal luidt het algoritme:

```
geef getal de beginwaarde
zolang de eindwaarde niet is bereikt doe
    als getal niet deelbaar door 7
        druk het getal af
    neem het volgende getal
```

Door de tekst in te laten springen wordt de hiërarchie in de code zichtbaar. In dit geval zijn er twee niveaus te onderscheiden:

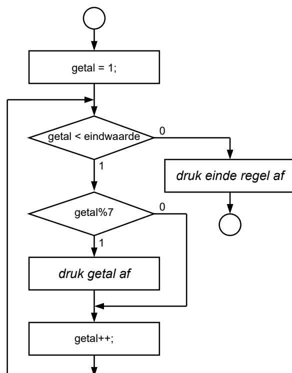
```
geef getal de beginwaarde
zolang de eindwaarde niet is bereikt doe
    als getal niet deelbaar door 7
        druk het getal af
    neem het volgende getal
```



Figuur 4.10 : Een Nassi-Shneiderman diagram voor het afdrukken van getallen.

Nadat er in spreektaal is opgeschreven wat het programma moet doen, kan stapsgewijs de spreektaal door code vervangen worden. De toewijzingen worden ingevuld, de '*zolang*' wordt een **`while`** en de '*als*' een **`if`**-statement. Bovendien wordt er na de **`while`** nog een nieuwe regel afgedrukt:

```
getal = 1;
while (de eindwaarde niet is bereikt) {
  if (getal niet deelbaar door 7) {
    printf("%d ", getal);
  }
  getal++;
}
printf("\n");
```



Figuur 4.11 : Het stroomdiagram voor het afdrukken van getallen.

Tenslotte wordt de rest van de code ingevuld. Voor de test of het getal deelbaar is door zeven wordt de modulus-operator gebruikt:

```
getal = 1;
while ( getal < eindwaarde ) {
  if ( getal % 7 ) {
    printf("%d ", getal);
  }
  getal++;
}
printf("\n");
```

In de jaren 70 van de vorige eeuw waren PSD's populair. De schrijver was tijdens zijn studie verplicht om een PSD in te leveren. Nadat deze was goedgekeurd mocht de code op een mainframe gecompileerd worden.
In de praktijk werd door de studenten eerst de code geschreven en daarna een bijbehorend PSD getekend.

Er bestaan veel grafische methoden om een programmastructuur te visualiseren. In figuur 4.10 staat het zogenoemde Nassi-Shneiderman diagram en in figuur 4.11 de flow-chart of stroomdiagram van het programma dat alle cijfers die niet deelbaar door 7 zijn afdrukt.

Een Nassi-Shneiderman diagram geeft de structuur van het programma weer met behulp van rechthoeken, driehoeken en tekst en wordt ook wel een PSD, *program structure diagram*, genoemd. Het voordeel van deze diagrammen is dat deze direct omgezet kunnen worden naar code. Het nadeel is dat het samenstellen veel werk is en dat het in veel gevallen eenvoudiger is om de code direct te schrijven.

4.7 Voorbeeld: cijferprogramma

Schrijf een programma dat steeds een geheel getal aan de gebruiker vraagt en vervolgens afdrukt of het resultaat goed is (> 7), of het voldoende (> 5) is, of dat het onvoldoende is. Als een nul wordt ingevoerd stopt het programma. Alle andere invoer dan 1 tot en met 10 levert een foutmelding op.

Het programma moet voortdurend twee dingen doen: de gebruiker om een cijfer vragen en daarna de waarde van het cijfer afdrukken. In pseudocode kan het ontwerp er zo uitzien:

```
vraag gebruiker om cijfer
zolang het cijfer ongelijk aan nul is doe
    druk de waarde van het cijfer af
    vraag gebruiker om cijfer
```

Eerst wordt de gebruiker om een getal gevraagd. Zolang dat getal ongelijk aan nul is, wordt de waarde van het cijfer afgedrukt en wordt de gebruiker weer om een getal gevraagd.
De twee subtaken die het programma moet doen, worden met twee functies gerealiseerd. De functie `getScore` levert het cijfer dat de gebruiker invoert en de functie `showResult` drukt de waarde van het cijfer af.

```
score = getScore();
zolang score ongelijk aan nul is doe
    showResult(score);
    score = getScore();
```

Dit leidt tot het hoofdprogramma dat in code 4.11 staat. De variabele `score` bevat het cijfer dat door de gebruiker is ingevoerd. De functies `getScore` en `showResult` moeten nog verder uitgewerkt worden.

Code 4.11 : Het hoofdprogramma voor het afdrukken van de cijfers.

```
1  #include <stdio.h>
2
3  int  getScore();
4  void showResult(int score);
5
6  int main(void)
7  {
8    int score;
9
10   score = getScore();
11   while ( score != 0 ) {
12     showResult(score);
13     score = getScore();
14   }
15
16   return 0;
17 }
```

De functie `showResult` kan met een **`if-else-if`** worden gerealiseerd. Bij de opdeling van een bereik in verschillende categorieën kan het beste aan een zijde van het

bereik worden begonnen. In code 4.12 wordt getest of het cijfer groter is dan 10, daarna wordt getest of het cijfer groter is dan 7. Dit wordt herhaald tot er geen geldige cijfers meer zijn.

Code 4.12 : De functie `showResult` voor afdrukken van de waarde van het cijfer.

```
19  void showResult(int score)
20  {
21    if (score > 10) {
22      printf("The score %d is not valid.\n", score);
23    } else if (score > 7) {
24      printf("The score %d is good.\n", score);
25    } else if (score > 5) {
26      printf("The score %d is sufficient.\n", score);
27    } else if (score > 0) {
28      printf("The score %d is insufficient.\n", score);
29    } else {
30      printf("The score %d is not valid.\n", score);
31    }
32  }
```

De functie `getScore` moet een tekst afdrukken, die de gebruiker stimuleert om een getal in te voeren. Met de `scanf` die in paragraaf 5.2 behandeld wordt, kan de door de gebruiker ingevoerde waarde gelezen worden. In code 4.13 staat een functie `getScore`, die dit doet.

Code 4.13 : De functie `getScore` die de gebruiker om een cijfer vraagt.

```
34  int getScore(void)
35  {
36    int c;
37
38    printf("\nGive score : ");
39    scanf("%d", &c);
40
41    return c;
42  }
```

Een probleem bij de functie uit code 4.13 is dat als er door de gebruiker geen getal wordt ingevoerd de `scanf` geen waarde leest en de variabele `c` ongedefinieerd is en het programma is een oneindige loop kan komen. De functie geeft een willekeurige waarde terug aan het hoofdprogramma. De door de gebruiker ingevoerde tekens blijven in de buffer staan. Bij volgende aanroepen gebruikt de functie nog steeds de verkeerde, oude invoer. Het hoofdprogramma kan dan alleen nog met `^C` gestopt worden, zoals figuur 4.12 laat zien.

Een oplossing is om de variabele `c` op regel 36 in code 4.13 te initialiseren met de waarde `0`. Omdat bij foutieve invoer de `scanf`-functie niets doet, zal de functie de waarde `0` teruggeven en zal het hoofdprogramma stoppen.

```
/cc/functions
/cc/functions $ gcc -Wall -o valueScore main.c score.c

/cc/functions $ valueScore

Give score :  9
The score 9 is good.

Give score :  RUBBISH
The score 9 is good.

Give score :  The score 9 is good.

Give score :  The score 9 is good.

Give score :  The score 9 is good.

Give score :  The score 9 is good.

Give score :  The score 9 is good.

Give score :  The score 9 is good.
```

Figuur 4.12 : De uitvoer als er geen getal ingevoerd wordt bij het cijferprogramma.

Een alternatieve functie staat in code 4.14. In pseudocode luidt deze functie

```
zolang er nog geen correct cijfer gegeven is doe
    printf("\nGive score : ");
    lees cijfer
```

De functie `scanf` geeft het aantal gelezen items terug. De retourwaarde wordt aan de variabele `ret` toegekend. Als er een getal (`%d`) gelezen is, wordt `ret` gelijk aan `1`. Zolang `ret` gelijk aan `0` is, wordt de **`while`**-lus herhaald.

Code 4.14 : Een verbeterde functie `getScore`.

```
34  int getScore(void)
35  {
36    int c = 0;
37    int ret = 0;
38
39    while (ret == 0) {
40      printf("\nGive score : ");
41      ret = scanf("%d", &c);
42      if (ret != 1) {
43        emptyBuffer();
44      }
45    }
46
47    return c;
48  }
```

Als er door de gebruiker onzin wordt ingevoerd, laat de `scanf` op regel 41 de foutief ingevoerde gegevens in de invoerbuffer staan. In dat geval wordt op regel 43 met een nog te schrijven functie `empyBuffer` deze buffer leeggemaakt, anders blijft het programma voortdurend de foute invoer lezen. Het leegmaken van de invoerbuffer wordt in paragraaf 5.3 behandeld.

De redenen om functies te gebruiken zijn soms divers. De functie `showResult` bestaat uit één stuk code en doet één klus: het afdrukken van de juiste waarde. De functie `getScore` levert het cijfer op, maar vangt daarbij ook eventuele foutieve invoer af.

4.8 Het verschil tussen == en =

In code 4.14 staat op regel 41 een = en op regel 39 een ==. Dat zijn twee verschillende tekens, die niet verwisseld mogen worden.
De == is de gelijkheidsoperator en het vergelijkt twee waarden. De = is een toekenning. Een veel voorkomende fout is deze toekenningsoperator (=) te gebruiken op een plaats waar een gelijkheidsoperator (==) hoort te staan.

Code 4.15 : Het verschil tussen == en =.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int x;
6
7     printf("Give number:\n");
8     scanf("%d", &x);
9
10    // x can be any number
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    // This statement has no effect
21    x == 3;
22
23    // x will always be 0 at this point too
24    printf("The value of x isn't changed. It stays %d.\n", x);
25
26    return 0;
27 }
```

In code 4.15 staat op regel 11 staat bij de conditie een = en op regel 21 staat een ==. Voor elke waarde van `x` geeft dit programma deze uitvoer:

```
Will always be executed!
The value of x is 0.
The value of x isn't changed. It stays 0.
```

Op regel 11 wordt `x` niet met `0` vergeleken, maar wordt er `0` aan `x` toegekend. De testconditie bij het **`if`**-statement is daardoor altijd niet waar en wordt altijd de **`else`** uitgevoerd. Na afloop van het **`if`**-statement heeft `x` de waarde `0`.
Op regel 21 staat geen toewijzing. In plaats daarvan wordt `x` met 3 vergeleken. Het resultaat is altijd niet waar, maar hiermee wordt niets gedaan.

Met de compiler-optie `-Wall` geeft de compiler twee waarschuwingen:

```
....c:11:4:warning: suggest parentheses around assignment used as truth value
....c:21:4:warning: statement with no effect
```

De compiler ziet dat de code op regel 21 geen effect heeft en bij de toewijzing op regel 11 suggereert de compiler om haakjes om `x = 1` te plaatsen.

4.9 Functies en programmeervaardigheden

De taal C is een procedurele taal. Essentieel daarbij is dat een programma opgesplitst wordt in deelproblemen. Functies zijn bij C daarvoor de elementaire bouwstenen, ze:

- verkleinen de complexiteit;
- maken de code beheersbaar;
- verbeteren de leesbaarheid;
- maken hergebruik mogelijk;
- verminderen ongewenste redundantie;
- maken het gebruik van bibliotheken mogelijk;
- maken het mogelijk om een ontwerp over meerdere programmeurs te verdelen.

Programmeren is niet alleen het schrijven van code voor een applicatie. Programmeren heeft twee kanten: het vinden van algoritmes en het coderen van deze algoritmes. Bij algoritmes gaat het om:

- het zelf bedenken van algoritmes;
- het zoeken naar oplossingen;
- het bestuderen en begrijpen van door anderen bedachte oplossingen;
- de verdeel-en-heersstrategie kunnen gebruiken.

Naast een analytisch denkvermogen is er bij het ontwikkelen en het kiezen van de juiste algoritmes creativiteit nodig. Bij het coderen is kennis van de gebruikte taal essentieel en het netjes en gestructureerd werken belangrijk. Functies helpen bij beide aspecten van het programmeren.

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

De voorbeelden voor in- en uitvoer zijn:

- Het afdrukken met verschillende plaatsvervangers.
- Invoeren en afdrukken van naam en leeftijd.
- Alternatieven voor de functie `getScore` van het cijferprogramma.
- Ongeformatteerd invoeren en afdrukken.
- Een alternatieve `getScore` met `fgets` en `sscanf`.
- Invoeren van naam en leeftijd met behulp van argumenten.
- Robuuste versie van invoeren van naam en leeftijd met behulp van argumenten.

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.
Het schrijven naar en het lezen uit bestanden wordt besproken in paragraaf 13.1. Dit hoofdstuk behandelt de invoer van tekst met het toetsenbord en de uitvoer naar het scherm.

5.1 Geformatteerde uitvoer

In de hoofdstukken 2 en 3 is de functie `printf` geïntroduceerd waarmee gegevens geformatteerd worden afgedrukt. Het eerste argument dat aan `printf` wordt meegegeven, is de zogenoemde *format string* waarin *format specifiers* of plaatsvervangers kunnen staan. Zowel in paragraaf 3.1 en 3.5 is uitgelegd dat de volgende argumenten, die `printf` meekrijgt, op de plaats van de *format specifiers* komen te staan. De betekenis van de enen en nullen is verschillend voor ieder datatype, daarom kent ieder datatype een eigen plaatsvervanger. In tabel 5.1 staan de plaatsvervangers voor de functie `printf`.

Pointers, en dus ook het type **void** *, komen in hoofdstuk 11 aan de orde.

Tabel 5.1 : De format specifiers of plaatsvervangers.

specifier	uitleg	datatype	voorbeeld
%s	string	**char** *, **char** []	tekst
%c	karakter	**char**	c
%d of %i	geheel getal	**int**	100
%u	positief geheel getal	**unsigned int**	100
%x, %X	hexadecimaal	**int**	0x64, 0X64
%o	octaal	**int**	0144
%f	gebroken getal	**float**, **double**	100.03
%e, %E	wetenschappelijk	**float**, **double**	1.030000e+02, 1.030000E+02
%g, %G	gebroken getal of wetenschappelijk	**float**, **double**	100.03 of 1.030000e+02, 1.030000E+02
%p	pointer	**void** *	ab001234
%%	procent teken		%

In code 5.1 staat een programma dat de verschillende uitvoer voor de diverse plaatsvervangers laat zien. Het commentaar aan het eind van iedere regel toont de uitvoer van de betreffende afdrukopdracht.

Een lange *format string* kan met een \ afgebroken worden en dan doorgaan op de volgende regel:

```
printf("Hello \
World \
World\n");
```

Achter de \ mag verder niets staan.

Beter is het om het over meerdere `printf`'s te verdelen:

```
printf("Hello ");
printf("World ");
printf("World\n");
```

Code 5.1 : Afdrukken van met behulp van verschillende plaatsvervangers.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    printf("Hello World!\n");                    // Hello World!
6    printf("%s %s", "Hello", "World!\n");        // Hello World!
7    printf("%s is %d\n", "John", 29);            // John is 29
8    printf("char %c is ASCII: %d\n", 'a', 'a'); // char a is ASCII 97
9    printf("%d + %d = %d\n", 4 , 5, 9);          // 4 + 5 = 9
10   printf("This is %d decimal\n", 24+25);       // This is 49 decimal
11   printf("This is %x hexadecimal\n", 49);      // This is 31 hexadecimal
12   printf("This is %o octaal\n", 49);           // This is 61 octaal
13   printf("This is %f\n", 49.2);                // This is 49.200000
14   printf("This is %e\n", 49.2);                // This is 4.920000+01
15   printf("This is %g\n", 49.2);                // This is 49.2
16
17   return 0;
18 }
```

Datatypen kunnen op allerlei manieren worden weergegeven. Getallen en tekst kunnen bijvoorbeeld links en rechts worden uitgelijnd en een gebroken getal kan als `123.45` of als `1.2345E+02` worden genoteerd.
Speciale extra tekens tussen het procentteken (%) en de conversieletter passen het afgedrukte formaat van het datatype aan. In code 3.5 is op regel 17 het gebroken getal afgedrukt met drie cijfers nauwkeurig met behulp van de plaatsvervanger `%.3f`.

De opties worden gegroepeerd in vier groepen: *flags*, *fieldwidth*, *precision* en *modifier*. Deze vier opties mogen worden gecombineerd. De syntax van een *format specifier* is:

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

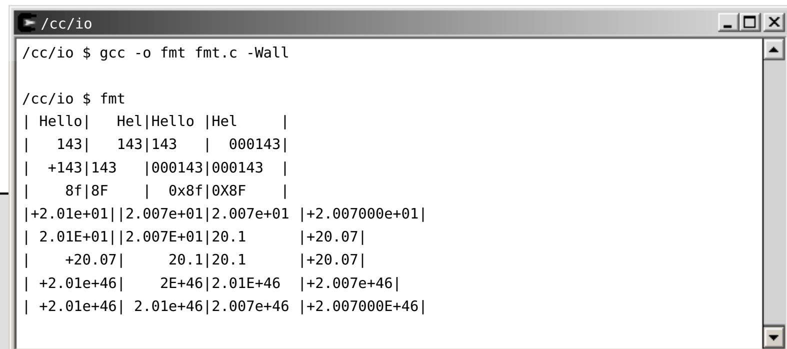
Niet alle opties hebben bij elk datatypen een betekenis of een identieke betekenis. De *modifier* wordt gebruikt bij getallen en past de afmeting van het formaat aan. Standaard worden `%d` en `%i` als een **`int`** en `%f`, `%e`, `%E`, `%g` of `%G` als **`double`** afgedrukt. De *modifier* past deze standaardinstelling aan. De *fieldwidth* geeft de breedte van de tekst die wordt afgedrukt. Als de tekst breder is dan de *fieldwidth* wordt de breedte van de tekst gebruikt. De *precision* stelt de nauwkeurigheid in waarmee een gebroken getal wordt afgedrukt en legt bij strings het aantal karakters vast dat afgedrukt wordt.

Tabel 5.2 geeft alle opties voor de vier groepen en figuur 5.1 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 5.2 : **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, wordt de breedte van de tekst gebruikt.
	`*`	Dit is een plaatsvervanger. In de argumentenlijst staat een extra argument voor het argument waar deze `*` bijhoort. Dit extra argument is een integer dat de breedte aangeeft.
precision	*number*	Is de nauwkeurigheid bij `e`, `f` en `E`, het aantal significante cijfers bij `g` en `G` en het aantal karakters dat afgedrukt bij een string (`%s`).
	`*`	Dit is een plaatsvervanger. In de argumentenlijst staat een extra argument voor het argument waar deze `*` bijhoort. Dit extra argument is een integer dat de precisie aangeeft.
modifier	`l`	Drukt een integer (`%d`, `%i`, `%u`, `%o`, `%x` of `%X`) als `long` af.
	`ll`	Drukt een integer (`%d`, `%i`, `%u`, `%o`, `%x` of `%X`) als `long long` af.
	`L`	Drukt een gebroken getal (`%f`, `%e`, `%E`,`%g` of `%G`) als **`long double`** af.
	`h`	Drukt een integer (`%d`, `%i`, `%u`, `%o`, `%x` of `%X`) als **`short`** af.

Bij MinGW functioneren sommige *modifiers*, zoals `ll` en `L` niet, bij 32-bits Windows-versies. Compileer in dat geval met de optie:
`-D__USE_MINGW_ANSI_STDIO`
of zet in de code voor `#include <stdio.h>` deze definitie:
`#define __USE_MINGW_ANSI_STDIO 1`

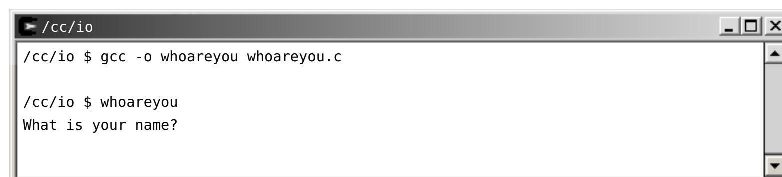



```c
#include <stdio.h>

int main(void)
{
  char s[]="Hello";
  int  i=143;
  float f=20.07;
  double d=20.07e+45;

  printf("|%6s|%6.3s|%-6s|%-8.3s|\n"    ,s,s,s,s);
  printf("|%6d|%6.3d|%-6d|%8.6d|\n"     ,i,i,i,i);
  printf("|%+6d|%-6.3d|%-.6d|%-8.6d|\n" ,i,i,i,i);
  printf("|%6x|%-6X|%#6x|%-#8X|\n"      ,i,i,i,i);
  printf("|%+3.2e||%6.3e|%-10.3e|%+6e|\n",f,f,f,f);
  printf("|% 3.2E||%6.3E|%-10.3G|%+6G|\n",f,f,f,f);
  printf("|%+10.4g|%9.3g|%-10.3g|%+6g|\n",f,f,f,f);
  printf("|%+10.3g|%9.2G|%-10.3G|%+6g|\n",d,d,d,d);
  printf("|%+10.2e|%9.2e|%-10.3e|%+2E|\n",d,d,d,d);

  return 0;
}
```

Figuur 5.1 : Geformatteerd afdrukken. Nevenstaande code geeft een aantal verschillende afdrukmogelijkheden en toont de schermafdruk met het resultaat.

5.2 Geformatteerde invoer

Op dezelfde manier als de functie `printf` tekst naar het beeldscherm schrijft, leest de functie `scanf` tekstinvoer van het toetsenbord. Het eerste argument van de functie `scanf` is eveneens een *format string* met plaatsvervangers voor de te lezen variabelen.
Code 5.2 vraagt achtereenvolgens om je naam en je leeftijd en drukt vervolgens deze informatie geformatteerd af. Figuur 5.2 toont de compilatie en de aanroep van het programma. Het programma `whoareyou` vraagt om je naam op te geven.

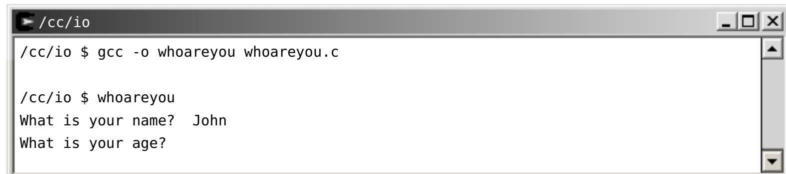


Figuur 5.2 : De compilatie van code 5.2 en de aanroep het programma `whoareyou`. De gebruiker wordt gevraagd zijn naam in te toetsen.

Code 5.2 : Lezen en afdrukken naam en leeftijd.

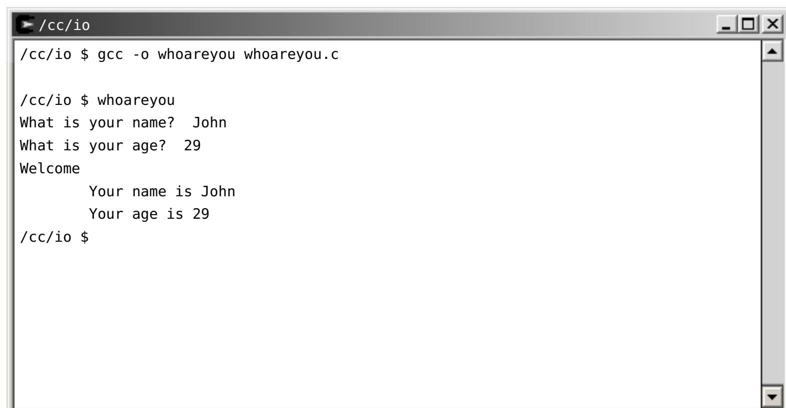
```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char name[16];
6    int  age;
7
8    printf("What is your name?");
9    scanf("%s", name);
10
11   printf("What is your age?");
12   scanf("%d", &age);
13
14   printf("Welcome\n");
15   printf("\tYour name is %s\n", name);
16   printf("\tYour age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

Nadat je je naam hebt opgegeven en op de entertoets hebt gedrukt, wordt er om je leeftijd gevraagd, zoals figuur 5.3 laat zien.



Figuur 5.3 : 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 5.4 laat deze uitvoer zien.



Figuur 5.4 : Na de invoer van je leeftijd drukt het programma `whoareyou` direct de naam en leeftijd af.

Uitleg code 5.2 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 4.5 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 element van de array.

De functie `scanf` is — zeker voor een beginnend programmeur — lastig in het gebruik om deze redenen:

Bij het lezen en schrijven worden gegevens opgeslagen in buffers. Het operating system werkt veel efficiënter wanneer de gegevens worden gebufferd en de karakters niet apart één voor één worden verwerkt. De grootte van de buffer en de wijze van afhandeling hangt af van het operating system en van de gekozen compiler.
De programmeur merkt daar in het algemeen niets van.

- De adresoperator `&` wordt vaak vergeten.
  De functie `scanf` schrijft de ingevoerde waarde dan op een heel andere plaats in het geheugen. Met als gevolg dat het programma crasht of een onvoorspelbaar gedrag vertoont.
- De plaatsvervanger moet correct gekozen worden.
- Foute invoer wordt niet afgevangen
  Als de invoer van de gebruiker niet juist is, worden de ingevoerde gegevens niet goed of helemaal niet verwerkt en blijven dan in de buffer staan. Het programma komt dan in een oneindig lus terecht.

Het doel van `scanf` is om tijdens de executie van een programma de gebruiker om gegevens te vragen. Dat is meteen ook ander nadeel van `scanf`. Bij het testen van de applicatie moet de programmeur dan eveneens voortdurend gegevens invoeren. Dat maakt de testprocedure arbeidsintensief en lastig te automatiseren.

5.3 Voorbeeld: invoer gegevens cijferprogramma

Bij het voorbeeld uit paragraaf 4.7 staat in code 4.13 een functie `getScore`, die gebruik maakt van een functie `emptyBuffer` om de invoerbuffer leeg te maken.

Een alternatief voor de functie `getScore` staat in code 5.3. Als er door de gebruiker onzin is ingevoerd, laat de `scanf` op 8 dit in de invoerbuffer staan. De waarde van `ret` is dan nul. De `scanf` op regel 9 leest in dat geval de foutief ingevoerde gegevens. Deze `scanf` bevat geen variabele. De `*` in de plaatsvervanger `%*s` betekent dat de string nergens aan wordt toegekend. De informatie wordt alleen gelezen en niet gebruikt.

Code 5.3 is nog niet robuust. Figuur 5.5 laat zien dat in het geval de ingevoerde informatie spaties bevat, dat in één keer voor iedere waarde de score wordt gegeven.
De functie `getScore` uit code 5.4 negeert alle invoer na een spatie. De `scanf` op regel 9 maakt de buffer leeg. Deze functie `getScore` negeert daardoor alle invoer nadat er een getal gescand is. De plaatsvervanger `%*s` is op regel 9 vervangen door `%*[^\n]`. De rechte haken in de plaatsvervanger worden gebruikt om in plaats van

Code 5.3 : Een alternatieve functie `getScore`.

```
 1  int getScore(void)
 2  {
 3    int c = 0;
 4    int ret = 0;
 5
 6    while (ret == 0) {
 7      printf("\nGive score : ");
 8      ret = scanf("%d", &c);
 9      if (ret != 1) scanf("%*s");
10    }
11
12    return c;
13  }
```

Code 5.4 : Een verbeterde alternatieve functie `getScore`.

```
 1  int getScore(void)
 2  {
 3    int c = 0;
 4    int ret = 0;
 5
 6    while (ret == 0) {
 7      printf("\nGive score : ");
 8      ret = scanf("%d", &c);
 9      scanf ("%*[^\n]");          // empty buffer
10    }
11
12    return c;
13  }
```

```
/cc/functions
/cc/functions $ valueScore

Give score :  4 5 6 789
The score 4 is insufficient.

Give score :  The score 5 is insufficient.

Give score :  The score 6 is sufficient.

Give score :  The score 789 is not valid.

Give score :
```

Figuur 5.5 : De uitvoer met functie `getScore` uit code 5.3.

een gewone specifier een zogenoemde *scanset* op te geven. De set `[ABCD]` betekent dat de invoer alleen de karakters A, B, C en D mag bevatten. Het dakje betekent dat de invoer de karakters juist niet mag bevatten. De scanset `[^\n]` bevat alle karakters behalve het symbool voor een nieuwe regel. Alle invoer tot het einde van regel wordt hiermee uit de buffer gelezen.

5.4 Ongeformatteerde in- en uitvoer

Naast de geformatteerde in- en uitvoer kent C ook ongeformatteerde in- en uitvoer. Informatie van een toetsenbord en naar een beeldscherm is niets anders dan het versturen en ontvangen van karakter. Code 5.5 laat een voorbeeld met de functies `getchar`, `putchar` en `puts`. Dit programma vraagt om een letter en drukt deze letter vervolgens af. De in- en uitvoer van het programma staat in figuur 5.6.

Uitleg code 5.5 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 13.1 worden besproken.

Code 5.5 : 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 }
```

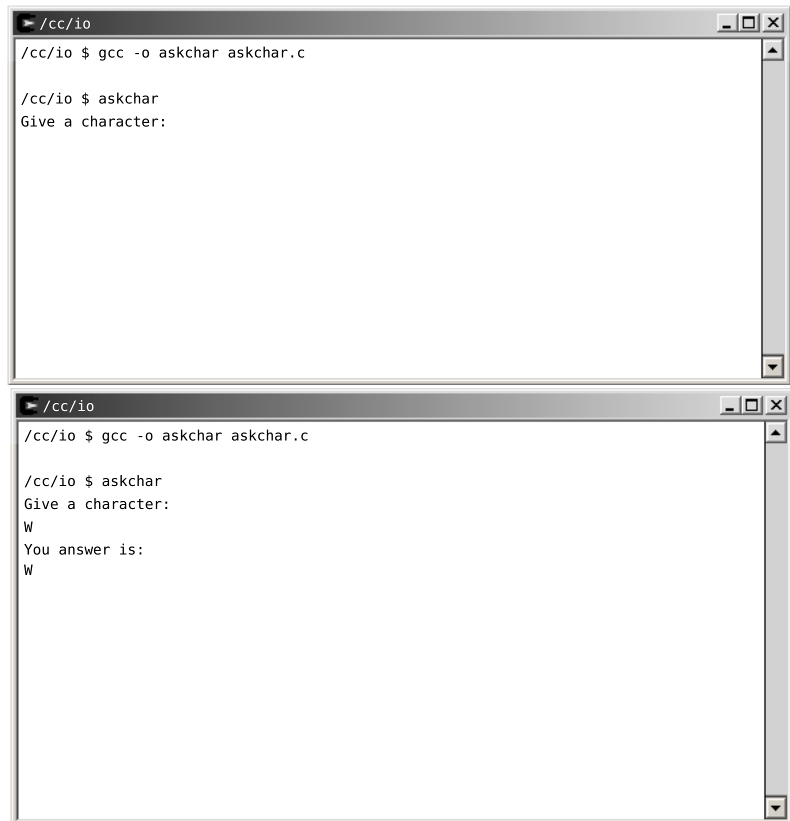
Regel 10
`int putchar(int c);`

De functie `putchar()` schrijft een karakter naar de standaarduitvoer. Het karakter wordt niet als `char` maar als `int` meegegeven. Deze functie hangt nauw samen met of is afgeleid uit de functies `putc()` en `fputc()`, die in de paragraaf 13.1 worden besproken.

Ook bij het ongeformatteerd afdrukken kunnen problemen ontstaan. In code 5.6 staat een variant op code 5.5. Dit programma leest twee karakters in plaats van één karakter. Regel 7 tot en met met 10 uit code 5.5 is in code 5.6 twee keer geplaatst. Op regel 12 is `putchar('\n');` toegevoegd om de tweede vraag op een nieuwe regel te laten beginnen.

Code 5.6 : Lezen en afdrukken van twee karakters.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int c;
6
7    puts("Give a character:");
8    c = getchar();
9    while ( getchar() != '\n' ) {}     // skip rest of line
10   puts("Your answer is:");
11   putchar(c);
12   putchar('\n');
13
14   puts("Give a another character:");
15   c = getchar();
16   while ( getchar() != '\n' ) {}     // skip rest of line
17   puts("Your second answer is:");
18   putchar(c);
19   putchar('\n');
20
21   return 0;
22 }
```



Figuur 5.6 : Het programma van code 5.5 vraagt om een letter (*character*) en nadat de gebruiker een letter (W) heeft opgegeven drukt het deze letter af.

De `while` op regel 9 is nu noodzakelijk omdat de `getchar` op regel 8 alleen het karakter leest en de enter (`'\n'`) in de buffer laat staan. Deze wordt dan automatisch, zonder nieuwe invoer, door de `getchar` op regel 8 als tweede karakter gelezen. Op regel 9 had ook alleen een `getchar()` kunnen staan om de enter te lezen. Er is gekozen voor een `while` om alle eventueel ingevoerde karakters te negeren.

De aanpassing bij code 5.6 en die bij codes 5.3 en 5.4 waren nodig om de invoerbuffer leeg te maken. Soms wordt hiervoor de functie `fflush` gebruikt. Regel 9 en regel 16 in code 5.6 is dan vervangen door:

```
fflush(stdin);
```

Deze functie is echter alleen bedoeld om een *uitvoer*buffer leeg te maken. Het gedrag van `fflush` is voor invoerbuffers ongedefinieerd. Dit is compilerafhankelijk en daarom is het af te raden om dit toe te passen.

Code 5.7 : Invoer met behulp van `fgets` en `sscanf`.

```
1  #define MAXBUF   64
2  char  buffer[MAXBUF];
3
4  int getScore(void)
5  {
6    int c = 0;
7    int ret = 0;
8
9    while (ret == 0) {
10     printf("\nGive score : ");
11     if ( fgets(buffer, MAXBUF-1, stdin) != NULL ) {
12        ret = sscanf(buffer, "%d", &c);
13     }
14   }
15
16   return c;
17 }
```

5.5 Alternatief voor het invoerprobleem

Een alternatief voor de problemen met de invoerbuffer is om altijd eerst de hele regel in een eigen buffer te zetten en daarna deze buffer te evalueren.

In code 5.7 staat een alternatieve functie `getScore`. Op regel 11 leest de functie `fgets` de invoer en zet deze in een string `buffer`. De functie `sscanf` leest op dezelfde manier als `scanf` gegevens uit een string `buffer`. De string waar `sscanf` uitleest, is een extra argument voor de *format string*.

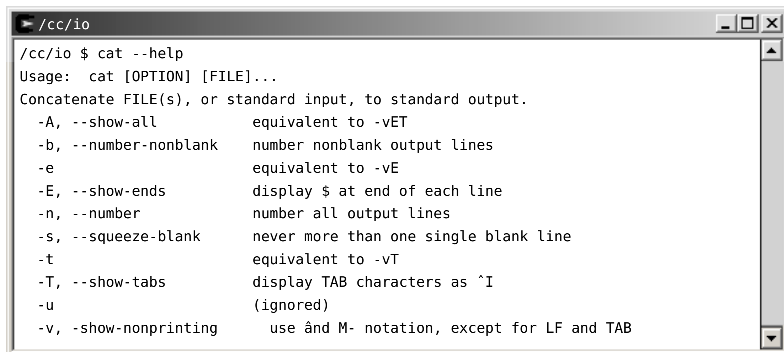
De string `buffer` is niet in de functie maar globaal gedeclareerd. Nadat een functie is uitgevoerd wordt het geheugen voor de lokale variabelen vrijgegeven. Variabelen die veel geheugenruimte innemen worden daarom vaak globaal gedeclareerd. In dit voorbeeld bestaat de buffer uit 64 karakters. Op regel 11 staat de macrodefinitie `MAXBUF`. De directieve **`#define`** wordt in paragraaf 6.7 besproken. In de rest van code leest de compiler in plaats van `MAXBUF` de tekst 64.

Het nadeel van deze invoermethode met `fgets` en `sscanf` is dat vooraf bekend moet zijn hoeveel karakters er ingevoerd kunnen worden. Een buffer van 64 karakters zou te klein kunnen zijn. Als er meer dan 63 karakters worden ingevoerd blijft de rest van de invoer toch nog in de invoerbuffer staan.

De functie `fgets` is bedoeld om bestanden te lezen. Paragraaf 13.1 bespreekt `fgets` en andere in- en uitvoerfuncties voor het lezen uit en het schrijven naar bestanden.

5.6 Argumenten doorgeven aan een programma

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 5.7 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 5.7 : Hier staat een deel van de opties van het Unix-commando `cat`. Met `cat` worden bestanden aaneengevoegd.

In code 5.8 staat een programma dat twee argumenten meekrijgt, namelijk: een naam en een leeftijd. Het programma drukt deze naam en leeftijd af. Figuur 5.8 toont de uitvoer van het programma.

Code 5.8 : 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 }
```

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.

```
/cc/io
/cc/io $ gcc -o argname argname.c -Wall

/cc/io $ argname John 29
Your name is John and you are 29.
```

Figuur 5.8 : Het programma van 5.8 drukt de argumenten `John` en `29` af.

Uitleg code 5.8 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.

Regel 8

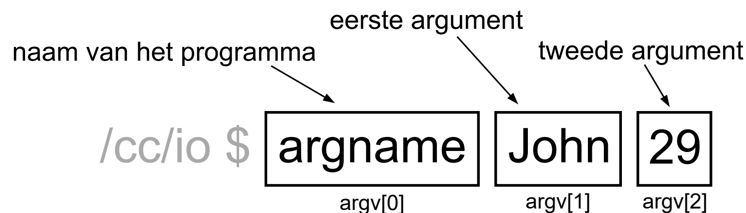
```
int atoi(char *s)
```

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`.

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
```

is `argc` gelijk aan `3` en bevat `argv[0]` de string `"argname"`, `argv[1]` de string `"John"`, `argv[2]` de string `"29"`, zoals figuur 5.9 laat zien.



Figuur 5.9 : De aanroep van het programma `argname`.

Het programma uit code 5.8 is niet robuust. De gebruiker kan informatie opgeven waardoor het crasht of onzin oplevert. In figuur 5.10 wordt het programma eerst aangeroepen met als tweede argument een niet-alfanumerieke waarde. Omdat `atoi` een `0` retourneert als de ingangswaarde niet-alfanumeriek is, wordt de leeftijd `0` in plaats van 29. Als het programma zonder argumenten wordt aangeroepen, wordt het gedrag 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 5.9 staat een verbeterde versie. Bij minder dan drie argumenten (programmanaam,

```
/cc/io
/cc/io $ argname John negenentwintig
Your name is John and you are 0.

/cc/io $ argname
    11 [main] argname 3840 _cygtls::handle_exceptions:Error while dumping
state (probably corrupted stack)
Segmentation fault (core dumped)

/cc/io $
```

Figuur 5.10 : De aanroep van het programma van code 5.8 met verkeerde invoer.

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 is, retourneert `atoi()` de waarde `0` en wordt er ook een foutmelding gegeven.

```
/cc/io
/cc/io $ argname John 29
Your name is John and you are 29.
/cc/io $ echo $?
0

/cc/io $ argname
usage:  argname <name> <age>
/cc/io $ echo $?
1

/cc/io $
```

Figuur 5.11 : Het programma van 5.9 vangt foutieve invoer af. De returncode is `0` bij correcte invoer en 1 bij foutieve invoer.

Uitleg code 5.9 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 5.9 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.

Waarschijnlijk omdat het op tekst gebaseerd is, vinden sommigen het doorgeven van informatie via de commandoregel ouderwets. Toch is het een praktische en veelgebruikte methode bij de ontwikkeling en het testen van software. In het commandovenster kunnen oude opdrachten met de pijltjestoetsen worden teruggehaald. Bij een programma dat om informatie vraagt, moet de gebruiker de gegevens steeds opnieuw intoetsen. Bovendien is het eenvoudiger scripts te maken die de software automatisch testen.

In de voorafgaande paragrafen is naar voren gekomen dat het schrijven van een programma, dat invoer van een gebruiker gebruikt, niet triviaal is. Geformatteerde invoer met `scanf` is zonder meer af te raden. De combinatie van `fgets` en `sscanf` geeft minder problemen. Met `getchar` moet de programmeur er rekening mee houden dat er gegevens in de invoerbuffer kunnen blijven staan.

Code 5.9 : Een robuuste versie van code 5.8.

```
 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  }
```

Programma's hebben in de praktijk vaak opties, die als argumenten meegegeven worden. Daarnaast halen programma's de gegevens uit bestanden, die ze bewerken en daarna weer in een bestand opslaan. Paragraaf 13.1 behandelt het lezen uit en het schrijven naar bestanden.

Bij microcontrollerapplicaties is de interactie met de gebruiker heel anders. De microcontroller heeft sensoren, drukknoppen, leds of eenvoudige display om met een gebruiker te communiceren. De functies `getchar` en `scanf` worden daarbij niet gebruikt. Het schrijven naar een display en het lezen via een seriële verbinding heeft wel veel overeenkomsten met het lezen uit en schrijven naar bestanden.

Code 5.10 : 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  }
```

5.7 Declaratie en het gebruik van strings

Deze paragraaf gaat niet specifiek over in- en uitvoer. Het behandelt een aantal aspecten van het gebruik van strings. Bij invoer van een karakterstrings kan de programmeur hiermee te maken krijgen.

De C-compiler leest de source code sequentieel van boven naar beneden. De declaratie van een type, variabele en functie zorgt ervoor dat de compiler de naam kent en bij variabelen reserveert de compiler de juiste hoeveelheid geheugen. Komt de compiler een gedeclareerde naam tegen, dan weet de compiler hoe deze toegepast gebruikt moet worden. Ongedefinieerde namen geven een foutmelding of een waarschuwing.

Het programma van code 5.10 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`.

Gebruik van de functie `strcpy`

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. Verderop in het programma worden aan de variabelen waarden toegekend. In code 5.11 zijn op regel 6 en 7 de variabelen `age` en `name` gedeclareerd en op regel 9 en 10 krijgen deze variabelen hun waarden.

Code 5.11 : 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 5.11 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 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 11 besproken en daar komt dit soort problemen met pointers uitgebreid aan de orde.

Code 5.12 : Code 5.11 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
12     printf("My name is %s and my age is %d", name, age);
13
14     return 0;
15  }
```

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 12 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 12 komen de stringfuncties uitgebreider aan bod.

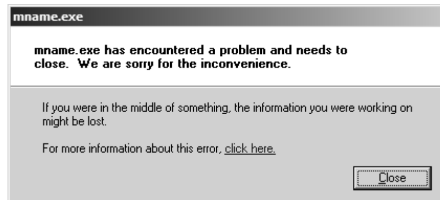
Fouten bij een onjuiste declaratie

In code 5.11 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 5.12 gebeurt, kunnen er fouten optreden.
Wat er precies gebeurt, is vooraf niet altijd te voorspellen. Er zijn ruwweg 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 5.12 een foutmelding, die in figuur 5.12 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.

De eerste mogelijkheid lijkt geen probleem, maar is in feite de ergste. Bij een revisie van het programma wordt er een functionaliteit toegevoegd, die niets te maken heeft met fout die al in de oorspronkelijke code zit. Bij het opnieuw compileren deelt de compiler het geheugen anders in en leidt de oude fout dat het programma crasht. De programmeur zal in dat geval in eerste instantie niet in de oude, maar in de nieuw toegevoegde code zoeken.

Het blauwe scherm met witte karakters bij Windows is ook het gevolg van runtime errors. De uitvoer van figuur 5.12 is daarmee vergelijkbaar. Tegenwoordig vangt Windows deze fouten wat netter af. Bij MinGW geeft Windows deze mededeling:




```
/cc/declaration
/cc/declaration $ gcc -o mname mname.c

/cc/declaration $ mname
  16203 [main] mname 584 _cygtls::handle_exceptions:Error while dumping
state (probably corrupted stack)
Segmentation fault (core dumped)
```

Figuur 5.12 : Runtime error bij het uitvoeren programma code 5.12.

Kleine veranderingen met grote gevolgen

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 5.13. 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.

Code 5.13 : 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 5.14 is een variabele `place` voor de woonplaats toegevoegd.

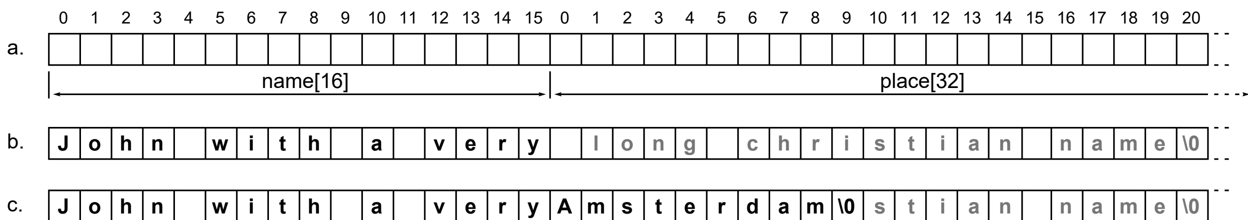
Figuur 5.13 laat zien dat in het geheugen de string `place` gedeeltelijk over string `name` komt te liggen en dat string `name` een combinatie wordt van het begin van string `name` en van string `place`.

De uitvoer van het programma van code 5.14 staat in figuur 5.14. Er treedt nu geen *runtime error* op, maar de uitvoer levert wel een vreemd resultaat op.

Bij een taal als Java is dit soort fouten niet mogelijk. Bij C moet de programmeur er zelf op letten dat er niet per ongeluk buiten een array of string wordt geschreven.

Code 5.14 : 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 5.13 : Het gebruik van het geheugen bij code 5.14.
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`.

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.

```
/cc/declaration
/cc/declaration $ gcc -o mname mname.c -Wall

/cc/declaration $ mname
My name is John with a veryAmsterdam, my age is 49 and I live in Amsterdam
```

Figuur 5.14 : De uitvoer van het programma van code 5.14.

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.
- Definities en macro's met behulp van **`#define`**
- 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 voorbeeld met macrodefinities.
- Een voorbeeld van het gebruik van **`switch`** bij een toestandsmachine.

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 op en bewerkingen met 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 op en bewerkingen met 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
```

```
17    if ( isalnum(x) ) {
18      printf("%c is alphanumeric ", x);
19      if ( isdigit(x) ) {
20        printf("is a digit");
21      } else {
22        if ( isupper(x) ) {
23          printf("is uppercase");
24        } else {
25          printf("is lowercase");
26        }
27      }
28    } else {
29      printf("%c is non-alphanumeric ", x);
30      if ( ispunct(x) ) {
31        printf("and is an interpunction");
32      }
33    }
34    printf("\n");
35
36    return 0;
37  }
```

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.

```
/cc/conditional
/cc/conditional $ gcc -Wall -o whatisthis whatisthis.c

/cc/conditional $ whatisthis D
D is alphanumeric is uppercase

/cc/conditional $ whatisthis 5
5 is alphanumeric is a digit

/cc/conditional $ whatisthis ,
, is non-alphanumeric and is an interpunction

/cc/conditional $ whatisthis " "
  is non-alphanumeric
```

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.

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3
 4 void print_ctype(int x);
 5
 6 int main(int argc, char *argv[])
 7 {
 8   if ( argc < 2 ) {
 9     printf("usage: %s <token>\n", argv[0]);
10     return 1;
11   }
12
13   print_ctype(argv[1][0]);
14
15   return 0;
16 }
```

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

```
 1 void print_ctype(int x)
 2 {
 3   printf("%c ", x);
 4   if ( isalnum(x) ) {
 5     printf("is alphanumeric ");
 6     if ( isdigit(x) ) {
 7       printf("and is a digit\n");
 8       return;
 9     }
10     if ( isupper(x) ) {
11       printf("and uppercase\n");
12       return;
13     }
14     printf("and lowercase\n");
15     return;
16   }
17   printf("is non-alphanumeric ");
18   if ( ispunct(x) ) {
19     printf("and is an interpunction\n");
20     return;
21   }
22   printf("\n");
23 }
```

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.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  void print_digit(int n);
 5
 6  int main(int argc, char *argv[])
 7  {
 8    if ( argc < 2 ) {
 9      printf("usage: %s <digit>\n", argv[0]);
10      return 1;
11    }
12
13    if ( strlen(argv[1]) > 1 ) {
14      printf("usage: %s <digit>\n", argv[0]);
15      return 2;
16    }
17
18    print_number(argv[1][0]-'0');
19
20    return 0;
21  }
```

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

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

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:

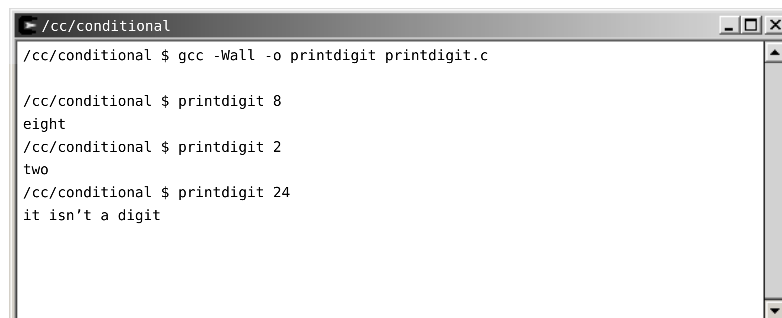
```
switch ( numerieke uitdrukking ) {
case voorwaarde_1:   [statements;]
                     [break;]
case voorwaarde_2:   [statements;]
                     [break;]
        ⋮
case voorwaarde_n:   [statements;]
                     [break;]
[default:]           [statements;]
                     [break;]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

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.

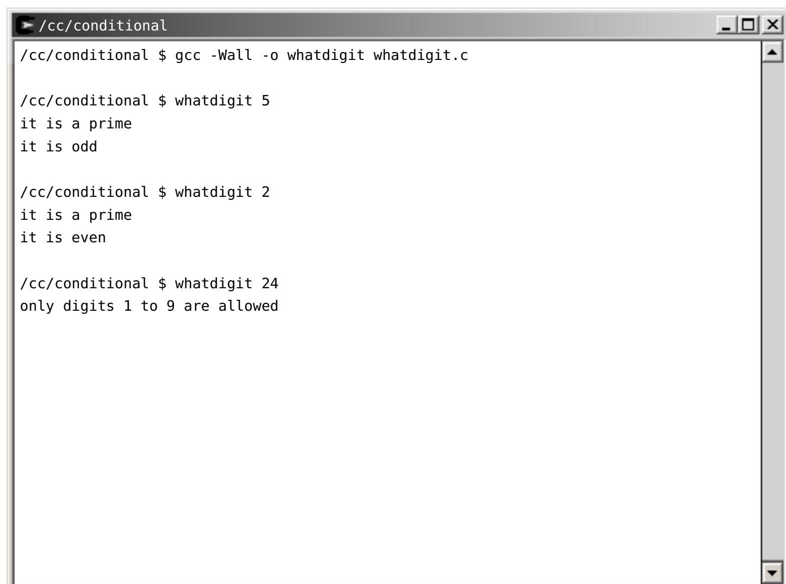


Figuur 6.4 : De uitvoer van het programma van code 6.6.

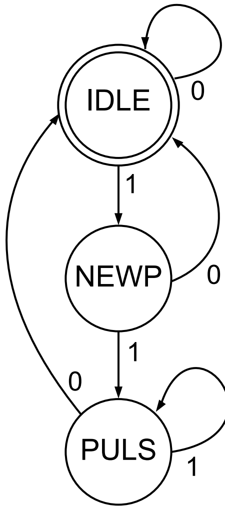
Code 6.9 : Programma dat eigenschappen van cijfers afdrukt.

```
 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 een 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[]="00011100011101100001111111000100011111101111000";
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 de programmacode vervangt de preprocessor `IDLE` door een `0`, `NEWP` door een `1` en `PULS` door een `2`.

6.7 Definities en macro's

Definities verbeteren de leesbaarheid en onderhoudbaarheid van de code. De namen van de macro's `IDLE`, `NEWP` en `PULS` staan voor de begrippen: *idle*, *new pulse* en *pulse*. Deze definities maken het **`switch`**-statement van code 6.10 beter leesbaar. In toestand `NEWP` is er een nieuwe puls gevonden en gaat de toestandsmachine afhankelijk van de waarde van `s[i]` naar de toestand `PULSE` of naar `IDLE`.

Onderhoudbaarheid is een andere belangrijke reden om definities te gebruiken. In code 5.7 is op regel 1 de grootte van de buffer gedefinieerd met:

```
#define MAXBUF   64
```

Op regel 2 en op regel 11 wordt in deze code `MAXBUF` gebruikt. Als de bufferafmeting verandert naar bijvoorbeeld 1024 karakters, is er slechts op één plaats — bij de **`#define`** op regel 1 — een aanpassing nodig. Als er geen definitie was gebruikt, moet de code op twee plaatsen worden gewijzigd, namelijk op regel 2 en op regel 11. Zeker bij grotere programma's is de kans groot dat er inconsistenties in de code ontstaan.

De **`#define`** voldoet aan deze syntax:

```
#define symbolic_name replacement_code
```

Een definitie wordt een macrodefinitie of een macro genoemd. Ook wordt hier het Engelse woord *define* voor gebruikt.

Met een macro bedoelt men gewoonlijk een definitie met argumenten.

De preprocessor vervangt voordat de code gecompileerd wordt overal in de programmacode de symbolische naam *symbolic_name* door *replacement_code*. Er zijn twee soorten definities te onderscheiden: definities zonder argumenten en definities met argumenten.

Definities zonder argumenten zijn vaak constanten. Dit is de reden dat symbolische namen gewoonlijk met hoofdletters en niet met kleine letters worden geschreven. Definities met argumenten lijken meer op functies en hebben daarom vaak kleine letters.

Definities zonder argumenten

De **`#define`** kent ook parameters en kan daarom ook gebruikt worden om macro's of macrodefinities te maken. De preprocessor vult overal, waar de symbolische naam staat, de vervangende tekst in met op de juiste plaatsen de waarden van de parameters. Een definitie met argumenten wordt een macro of macrodefinitie genoemd.

De `main` begint in code 6.12 op regel 211. De preprocessor heeft daarvoor de inhoud van `stdio.h` geplaatst.

In code 6.11 staat een voorbeeld en in code 6.12 staat dezelfde code na de preprocessing. Op regel 11 wordt de macro `printText` van regel 5 aangeroepen met het argument `The numbers are`. De preprocessor plaats dit argument op de plaats van parameter `x` in de *replacement code* van de macro en plaatst deze code op regel 213 in code 6.12.

Code 6.11 : Een voorbeeld met macrodefinities

```
 1 #include <stdio.h>
 2
 3 #define NUMBER            456
 4 #define printNumber(x)    printf("%d",(x))
 5 #define printText(x)      printf("%s",(x))
 6 #define printSpace        printf(" ")
 7 #define printEndOfLine()  printf("\n")
 8
 9 int main(void)
10 {
11   printText("The numbers are");
12   printSpace;
13   printNumber(123);
14   printText(" and ");
15   printNumber(NUMBER);
16   printEndOfLine();
17
18   return 0;
19 }
```

Code 6.12 : Code 6.11 na de preprocessing

```
211 int main(void)
212 {
213   printf("%s","The numbers are");
214   printf(" ");
215   printf("%d",123);
216   printf("%s"," and ");
217   printf("%d",456);
218   printf("\n");
219   return 0;
220 }
```

In code 6.11 wordt op regel 15 de definitie `NUMBER` als argument aan de macro `printNumber` meegegeven. De preprocessor vervangt `NUMBER` door `456` en gebruikt deze tekst als argument bij de macro `printNumber` en plaats de *replacement code* op regel 217 in code 6.12.

Hoewel de notatie van `printText` op regel 11 en van `printNumber` op regel 15 op die van een functie lijkt, is een macro *geen* functie.

De definities van `printSpace` en `printEndOfLine()` hebben geen argumenten, bij de eerste definitie staan geen haakjes en bij de tweede staan juist wel haakjes. De macro `printEndOfLine()` lijkt net als de macro's `printText` en `printNumber` op een functie. Op regel 12 staan bij `printSpace` geen haakjes, terwijl het statement net als een functie wel een bepaalde actie voorstelt. Veel programmeurs geven macro's, die een actie representeren, daarom het uiterlijk van een functie.

Macro's en functies

Eenvoudige functies worden vaak in de vorm van een macro geschreven. De belangrijkste reden daarvoor, is dat het programma daardoor sneller wordt. Voor de functieaanroepen heeft de processor extra handelingen nodig. Het programma moet naar de functie springen, er worden lokale variabelen gecreëerd en na afloop weer opgeruimd. Bij macro's wordt de code vervangen. Er hoeft niet door het programma te worden gesprongen en er zijn geen lokale variabelen die gecreëerd en opgeruimd moeten worden.
Een nadeel van macro's is dat de hoeveelheid programmacode toe kan nemen. Bij een functie staat de code op één plaats en bij een macro op iedere plaats waar de macro gebruikt wordt.

De taal C kent geen kwadraat. In plaats daarvan wordt een kwadraat berekend door een getal met zichzelf te vermenigvuldigen, zoals deze functie `square`:

```
int square(int x)
{
  return x*x;
}
```

Deze macro `square` levert eveneens een kwadraat op:

```
#define square(x)     ((x)*(x))
```

Een lange definitie mag met een \ afgebroken worden en doorlopen op de volgende regel:

```
#define      \
   square(x) \
   ((x)*     \
   (x))
```

Achter de \ mag verder niets staan.

In beide gevallen kent de toekenning:

```
z = square(5);             // result is 25
```

het kwadraat van 5 toe aan de variabele `z`.

Het definiëren van een macro is lastig. Er treden vaak fouten en vreemde neveneffecten op. Bij de macrodefinitie van square staan heel veel haakjes. Zonder haakjes geeft de macro niet in alle gevallen het correcte resultaat:

```
#define square(x)     x*x
z = square(5);             // result is 25
y = square(5+3);           // result is 23
```

Variabele `y` krijgt niet de te verwachten waarde 64, maar wordt gelijk aan 23. Dit komt omdat de preprocessor dit verandert in:

```
y = 5+3*5+3;
```

De uitkomst van deze berekening is 23. Bij de macrodefinitie van `square` met haakjes is de code na de preprocessing gewijzigd in:

```
y = ((5+3)*(5+3));
```

Deze toekenning geeft wel het correcte resultaat, namelijk 64.

Een andere fout is dat er een neveneffect optreedt:

```
#define square(x)     ((x)*(x))
i = 5;
z = square(++i);           // result is 49
```

Overigens geeft de compiler in dit voorbeeld wel een waarschuwing:
`warning: operation on 'i' may be undefined`

Veelal zal men verwachten dat `i` de waarde 6 en `z` de waarde 36 krijgt, De bijbehorende redenering is dat vanwege `++i` wordt `i` eerst opgehoogd wordt naar 6 en dat vervolgens aan `z` het kwadraat van 6 wordt toegekend. Bij de macro `square` is deze redenering niet correct. De preprocessor vervangt dit door:

```
z = ((++i)*(++i));
```

Er staat twee maal `++i`, zodat `i` twee keer met één verhoogd wordt, dus gelijk aan 7 is en dat `z` de waarde 49 krijgt.

Hoewel macro's lastig in gebruik zijn, is er nog een ander belangrijk voordeel. Macro's zijn type onafhankelijk. De functie `square` is bijvoorbeeld alleen geschikt voor integers. In dit voorbeeld:

```
printf("%d\n", square(2.5));   // result is 4  (square is a function)
```

geeft de functie `square` de uitkomst 4. Het gebroken getal 2,5 wordt afgekapt naar 2 en daarna gekwadrateerd. Voor ieder type is er vaak een aparte functie nodig.

In onderstaand voorbeeld geeft de macro de uitkomst 6,25:

```
printf("%g\n", square(2.5));    // result is 6.25  (square is a macro)
```

Het gebroken getal 2,5 wordt nu niet afgekapt maar twee keer in *replacement code* in de macro ingevuld `((2.5)*(2.5))` en met elkaar vermenigvuldigd . Het gedrag van de macro `square` hangt niet af van het gebruikte datatype.

Een macro is zoals eerder is gezegd *geen* functie. Macro's zijn vaak handig, maar zowel bij het maken als bij de toepassing ervan is er veel aandacht van de programmeur nodig.

6.8 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's `max` en `min` bepalen respectievelijk de maximale en minimale waarde van twee getallen:

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

De macro `max` komt overeen met deze functie `max`:

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

Deze macro `max` en `min` worden als volgt gebruikt:

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

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

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 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:

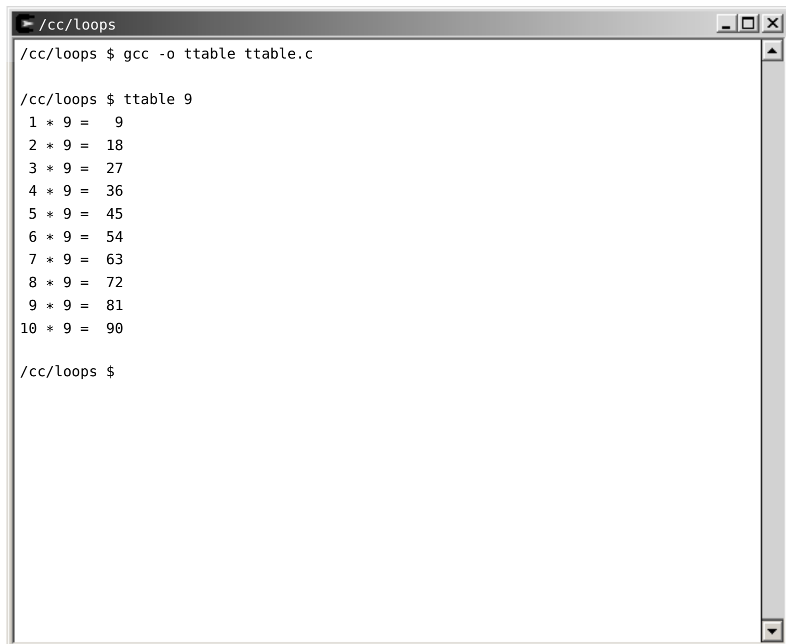
```
for(x = 12 ; (x < f(x)) || (a < 7); b = b(x) ) {
  // statements
}
```

Er mogen ook delen weg worden gelaten. Alles mag zelfs 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 compiler-optie `-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. Alleen bij de startconditie en het iteratie-deel van een **`for`**-lus wordt de komma-operator gebruikt.

Overigens zijn de meeste komma's, die bij C gebruikt worden, helemaal geen komma-operatoren. De komma is meestal een scheidingsteken tussen de parameters bij functie of tussen de waarden van array-elementen.

```
int a[] = {1, 2, 3, 5, 8};     // comma is separator in array-list
printf ("%d %d", a[2], a[4]);  // comma is separator between parameters
```

7.3 De while-lus

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
}
```

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.

Zolang de conditie waar is, worden de statements tussen de accolades uitgevoerd. Iedere **`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
}
```

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 }
```

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 );
```

De accolades waren oorspronkelijk bij de `do-while` niet nodig. De `do` impliceert een bloktoewijzing en de `while` sluit deze impliciet af. De accolades werden vaak toegevoegd voor de leesbaarheid. Tegenwoordig eist de compiler dat er accolades gebruikt worden.

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`:

```
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.

Een voorbeeld voor een toepassing van de `do-while` is de functie `getScore` uit code 5.4. In deze functie wordt een `while` gebruikt, die eerst test of er een correcte waarde is gelezen en dan om een score vraagt. In code 7.3 staat een versie met een `do-while` Nu wordt er eerst om een score gevraagd en daarna wordt getest of deze voldoet.

Code 7.3 : Functie `getScore` met `do-while`-statement.

```
1  int getScore(void)
2  {
3    int c;
4    int ret;
5
6    do {
7      printf("\nGive score : ");
8      ret = scanf("%d", &c);
9      scanf ("%*[^\n]");
10   } while ( ret != 1 );
11
12   return c;
13 }
```

7.5 Het break-statement en het continue-statement

Het `break`-statement is al eerder besproken bij de `switch`, maar kan ook bij de `for`, `while` en `do-while` gebruikt worden om de lus voortijdig te verlaten. Code 7.4 drukt van een getal af of het een priemgetal is of niet. Het programma vraagt om 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 geen priemgetal

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

Code 7.4 : 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 }
```

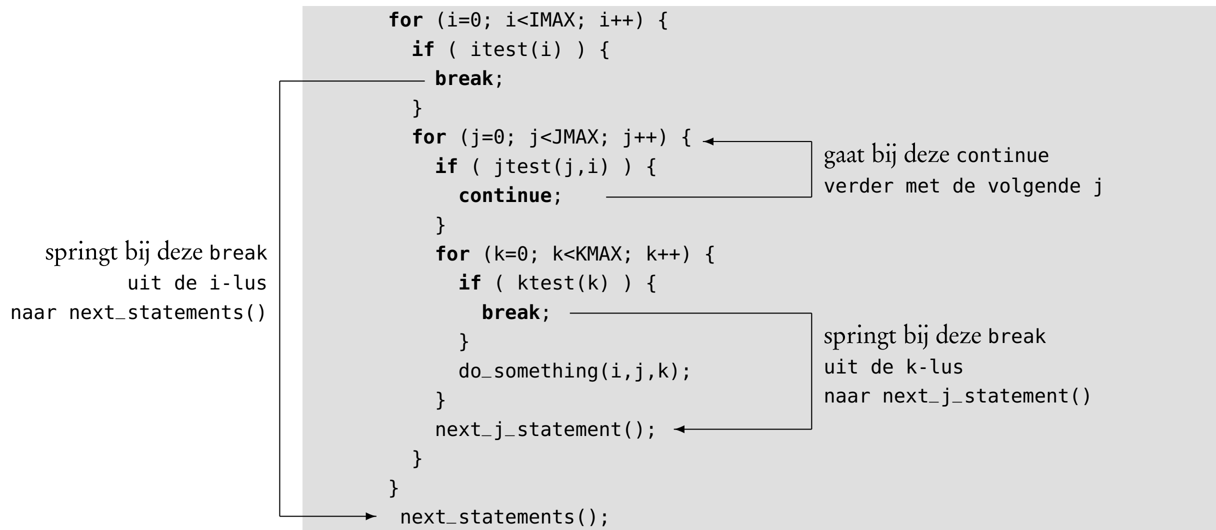
is. Als er geen deeltal is gevonden, wordt er afgedrukt dat het een priemgetal is. De `printf` gebruikt een conditionele operator om `is` of `isn't` af te drukken.

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 7.5 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.

Code 7.5 : 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 }
```

Het voorwaardelijk afdrukken bij code 7.5 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.



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: ▪ De algemene structuur van een programma. ▪ Het gebruik van commentaar: `/*`, `*/` en `//`. ▪ De opmaak van toewijzingen. ▪ Het gebruik van haakjes ▪ Het gebruik van spaties en lege regels. ▪ Het opmaakprogramma `indent`. ▪ 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.

Steve McConnell, auteur van veel boeken over software engineering, stelt: “If the code is so complicated that it needs to be explained, it’s nearly always better to improve the code than it is to add comments.”.

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 7.4 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 parentheses
b = (((-a) - (6*b)) < (z + 4));  // equivalent, parentheses showing the precedence
b = (-a - 6*b) < (z + 4);        // equivalent, with a few parentheses to improve readability
```

De volgorde waarin operatoren geëvalueerd worden, ligt vast met de prioriteits- en associativiteitsregels. Deze regels staan in tabel 9.8 van paragraaf 9.14. Maar de volgorde waarin de operanden geëvalueerd worden, ligt niet vast. Er treden zogenoemde neveneffecten (*side effects*) op. Meer informatie hierover staat in paragraaf 9.14. 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 ligt de volgorde waarin de argumenten van een functie worden berekend niet vast. Bij deze alternatieve formulering komen beide genoemde problemen niet voor:

```
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 block-statements 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.

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.

Gebruik altijd een vaste opmaak.

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.

```
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;
}
```

```
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;
}
```

```
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:

```
#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:

```
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.

```
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;
}
```

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.

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.

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

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.
- Integers met vaste afmetingen.
- Suffixen bij getallen.
- De sleutelwoorden `const`, `register` en `volatile`.
- Statische variabelen en statische functies.
- De `sizeof`-operator.
- Typedefinities, enumeraties,

De voorbeelden demonstreren:

- Typecasting.
- Het gebruik van `float` bij het bepalen van de Quételet-index.
- Het vergelijken van een `double` en een `float` met een constante.
- Het hexadecimaal en octaal van gehele getallen.
- Het binair afdrukken van gehele getallen.
- Toepassingen met de `sizeof`-operator.
- Het gebruik van statische en niet-statische variabelen.

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 gebroken getal en is het dan een *fixed point* of een *floating point* getal? Hoeveel bits zijn er beschikbaar om het getal vast te leggen? Bij de uitleg van code 5.11 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 achtbits getal, dat in het algemeen een ASCII-waarde voorstelt. De waarde 65 is bijvoorbeeld het karakter `'A'` en 97 is een `'a'`. Sommige karakters, zoals `'\n'` hebben een bijzondere betekenis. Tabel I.1 in bijlage I geeft een overzicht van alle ASCII-waarden `0` tot en met 127.

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 Xmega 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.

Tegenwoordig kent C ook integers met vaste afmetingen, zoals `uint8_t` en `uint32_t`. Meer informatie staat hierover in paragraaf 9.16.

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`.

Binair	Waarde
11111111	+255
11111110	+254
⋮	
10000001	+129
10000000	+128
01111111	+127
01111110	+126
⋮	
00000010	+2
00000001	+1
00000000	0

Figuur 9.1 : De binaire representatie van een unsigned char.

Binair	Waarde
01111111	+127
01111110	+126
⋮	
00000010	+2
00000001	+1
00000000	0
11111111	-1
11111110	-2
⋮	
10000001	-127
10000000	-128

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 de meest significante bit 1 is, is het two's complement getal negatief.

Tabel 9.1 geeft voor de `gcc`-compiler 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`, zie ook tabel G.7 in bijlage G, 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`.

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

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.

```
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.

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

/cc/format $ cast
          1000_1101  unsigned char    : 141
          1000_1101  (signed char)    : -115
0000_0000_1000_1101  (signed short)   : 141
0000_0000_1000_1101  (unsigned short) : 141

          1101_0111  signed char      : -41
          1101_0111  (unsigned char)  : 215
1111_1111_1101_0111  (signed short)   : -41
1111_1111_1101_0111  (unsigned short) : 65495

1010_1100_1000_1100  unsigned short   : 44172
1010_1100_1000_1100  (signed short)   : -21364
          1000_1100  (signed char)    : -116
          1000_1100  (unsigned char)  : 140

1110_1111_1111_0111  signed short     : -4105
1110_1111_1111_0111  (unsigned short) : 61431
          1111_0111  (signed char)    : -9
          1111_0111  (unsigned char)  : 247
```

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.

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 programmacode 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 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.

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

```c
#include <stdio.h>

void printb(long long int x, int nbytes, int space);

void print(int v, int nbytes, char *type, char *s)
{
  printf("%s", s);
  printb(v, nbytes, 1);
  printf("  %-17s: %d\n", type, v);
}

int main(void)
{
  unsigned char   uc = 141;
  signed char     sc = -41;
  unsigned short  us = 44172;
  signed short    ss = -4105;

  print(                 uc, 1, "unsigned char",    "          ");
  print((signed char)    uc, 1, "(signed char)",    "          ");
  print((signed short)   uc, 2, "(signed short)",   "");
  print((unsigned short) uc, 2, "(unsigned short)", "");

  print(                 sc, 1, "signed char",      "\n          ");
  print((unsigned char)  sc, 1, "(unsigned char)",  "          ");
  print((signed short)   sc, 2, "(signed short)",   "");
  print((unsigned short) sc, 2, "(unsigned short)", "");

  print(                 us, 2, "unsigned short",   "\n");
  print((signed short)   us, 2, "(signed short)",   "");
  print((signed char)    us, 1, "(signed char)",    "          ");
  print((unsigned char)  us, 1, "(unsigned char)",  "          ");

  print(                 ss, 2, "signed short",     "\n");
  print((unsigned short) ss, 2, "(unsigned short)", "");
  print((signed char)    ss, 1, "(signed char)",    "          ");
  print((unsigned char)  ss, 1, "(unsigned char)",  "          ");

  return 0;
}
```

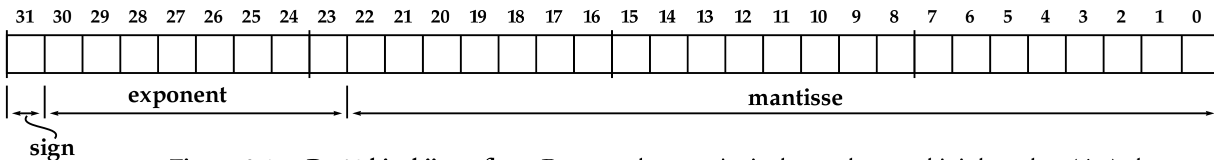
De programmacode waarmee de uitvoer van figuur 9.3 is gemaakt, staat in code 9.1. Op regel 5 tot en met 10 staat een functie `print` die de binaire waarde en de decimale waarde van een integer `v` afdrukt. De variabele `nbytes` bevat het aantal bytes dat `v` bevat. De variabele `type` wijst naar een 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.

9.3 Gebroken getallen

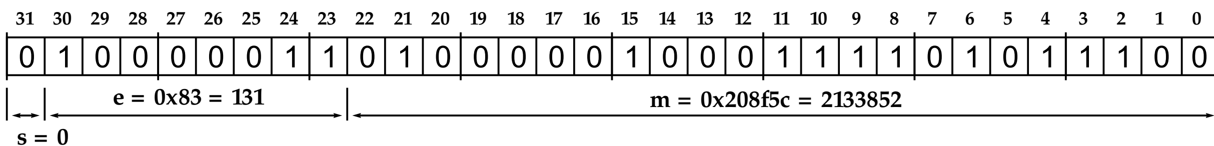
Gebroken getallen zijn getallen die niet geheel zijn, zoals 1,23; 50,6; 0,00338; 6,0 en 4,83 10^{13}. Voor gebroken getallen of drijvende-kommagetallen bestaan er drie typen: **`float`**, **`double`** en **`long double`**. Bij `gcc` neemt een **`float`** vier bytes ruimte in beslag, een **`double`** 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 Xmega, is een **`double`** vier bytes groot, net als een **`float`**.

Gebroken getallen worden opgeslagen volgens het IEEE754 Single Precision Format: de 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`**.



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

De exponent bepaalt het bereik en de mantissa 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 verschillende soorten gebroken getallen met de afmeting en de bijbehorende minimale en maximale waarden voor de `gcc`-compiler.

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`

Kleinere microcontrollers kennen vaak alleen de **`float`** of heeft de **`double`** dezelfde afmeting als een **`float`**. Bovendien is er meestal een speciale bibliotheek nodig om dit type te kunnen gebruiken.

De exponent bepaalt het bereik van de gebroken getallen. Het bestand `floats.h`, zie ook tabel G.8 in bijlage G, 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`**.

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.

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 een **`float`** of **`double`** worden gerepresenteerd. De programmeur zal daar voortdurend rekening mee moeten houden.

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 zou het antwoord 21,0 zijn 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.

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 }
```

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 9.1 laat een aantal toepassingen zien van typecasting bij gehele getallen. Code 9.3 toont een aantal situaties met en zonder typecasting bij het delen en bij de modulusoperator met gebroken getallen.

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).*

9.5 Constanten bij gebroken getallen

Omdat gebroken getallen of drijvende-kommagetallen, 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.

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

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.

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  }
```

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. Bij C-compilers voor eenvoudige microcontrollers, zoals de AVR-gcc voor de Xmega, zijn de `double` en de `float` beide type 32 bits groot en is het beter om altijd `double` te gebruiken.

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  }
```

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.

Standaard is er geen *format specifier* voor het binair afdrukken. Een `%b` ontbreekt. Code 9.6 van paragraaf 9.15 geeft een functie `printb`, die gehele getallen binair afdrukt.

Tabel 9.3 : 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	$\frac{\sinh x}{\cosh x}$
`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	$\lceil x \rceil$
`floor(x)`	afronden naar beneden	$\lfloor x \rfloor$
`round(x)`	afronden	
`fabs(x)`	absolute waarde	$\lvert x \rvert$
`pow(x,y)`	machtverheffen	x^y

9.7 Rekenkundige operatoren

Zoals in paragraaf 3.4 al is verteld, kent C standaard vijf rekenkundige bewerkingen: optellen, aftrekken, vermenigvuldigen, delen en de modulus. Tabel 3.1 geeft een overzicht van deze rekenkundige bewerkingen.

In paragraaf 3.4 is ook al gesteld dat de modulus in C — net als bij veel andere programmeertalen — geen wiskundige modulo is en dat het de rest van een geheeltallige deling is. De modulusfunctie `%` had daarom beter de *remainder* in plaats van *modulo* genoemd kunnen worden.

Deze macro bepaalt wel de echte, wiskundige modulus van twee getallen:

```
#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 de modulus-operator te gebruiken.

De bibliotheek *math*, zie tabel 9.3, bevat alle gangbare wiskundige functies. De parameters en de retourwaarde van de bewerkingen uit tabel 9.3 zijn allemaal van het type **double**. Ook de afronding floor(5.3) levert het gebroken getal 5,0 op.

Net als Java kent C geen symbool voor machtverheffen. De bibliotheekfunctie pow() rekent de macht van twee getallen uit. Voor het kwadrateren is deze functie niet nodig. Het kwadraat van a is ook gelijk aan a*a.

Tabel 9.4 : De rekenkundige bewerkingen uit de stdlib-bibliotheek.

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

Naast de bewerkingen uit math.h staan er in de standaardbibliotheek stdlib.h ook een aantal geheeltallige rekenkundige bewerkingen, zie tabel 9.4. Om de bewerkingen uit math.h en stdlib.h 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 : Escape sequences voor format strings.

escape sequence	hexadecimale ASCII-waarde	betekenis
\n	0x0A	newline
\r	0x0D	carriage return
\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	null character

Een enkele terugstreep, \, wordt gebruikt om lange definities en *format strings* af te breken en op de volgende regel te laten doorlopen:

```
#define voffset(x) \
    (0.05 *        \
    (((double) (x)) \
    / 1.6))
```

Achter de \ mag verder niets staan.

9.8 Het karaktertype char en de speciale karakters

In paragraaf 9.1 is gesteld dat **char** een 8-bits getal is, dat ook karakter kan representeren. C gebruikt voor de representatie de ASCII-tabel. In bijlage I geeft tabel I.1 een overzicht van alle ASCII-waarden 0 tot en met 127.

De meeste ASCII-waarden staan voor een bepaald symbool. In C worden er enkele aanhalingstekens om heen gezet. Voorbeelden zijn: `'a'`, `'A'`, `'3'`, `'+'`, `';'` en `'$'`. Voor sommige ASCII-waarden bestaat er geen representatie met een letter of symbool. In het geval er voor een bepaalde ASCII-waarde geen karaktersymbool bestaat, lost C dat op met speciale tekens. In de voorgaande hoofdstukken is al veelvuldig gebruik gemaakt van het symbool voor het einde van een regel: `\n`. Ook is in code 5.2 het symbool `\t` voor de tab of tabulator toegepast en is er een aantal keer het *null character* `\0` gebruikt dat het einde van de string aangeeft. Deze symbolen worden *escape sequences* genoemd. Tabel 9.5 geeft een overzicht.

In paragraaf 3.3 is al verteld dat in C een array van het type **`char`** afgesloten met het *null character* een string is. Strings worden in hoofdstuk 12 besproken.

Het headerbestand `ctype.h` bevat de prototypen en macro's van eenvoudige tests op en bewerkingen met karakters. Paragraaf 6.1 zijn een aantal van deze functies gebruikt en staat in tabel 6.1 geeft een overzicht.

9.9 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* betekent en dat alle andere 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;
```

Sommige C-compilers kennen het headerbestand `stdbool.h` met een eigen boolean type.

9.10 De relationele bewerkingen

Een relationele bewerking test of een getal groter dan, kleiner dan of gelijk is aan een ander getal. In paragraaf 3.6 zijn de relationele bewerkingen al besproken en geeft tabel 3.2 een overzicht. In tabel 9.6 is hieraan het domein toegevoegd.

Tabel 9.6 : De relationele operatoren met het domein waarvoor de uitkomst waar en niet waar is.

bewerking	wiskundig symbool	symbool in C	voorbeeld	domein *waar*	domein *niet waar*
groter dan	$>$	`>`	`x > 3`	3	3
groter dan of gelijk aan	$\geq$	`>=`	`x >= 3`	3	3
kleiner dan	$<$	`<`	`x < 3`	3	3
kleiner dan of gelijk aan	$\leq$	`<=`	`x <= 3`	3	3
gelijk aan	$=$	`==`	`x == 3`	3	3
ongelijk aan	$\neq$	`!=`	`x != 3`	3	3

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 9.7 : 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 Logische operatoren

C kent drie logische bewerkingen: de EN, de OF en de NIET. In C worden deze respectievelijk geschreven als `&&`, `||` en `!`. Figuur 9.7 geeft de waarheidstabellen voor deze drie bewerkingen. C kent standaard geen boolean type. In C betekent `0` niet waar en is elk ander geheel getal waar.

Bij ingewikkelde logische uitdrukkingen is het verstandig om ronde haken om elke deelbewerking te zetten, zodat er geen onduidelijkheid is over welke bewerking voorrang heeft:

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

9.12 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.

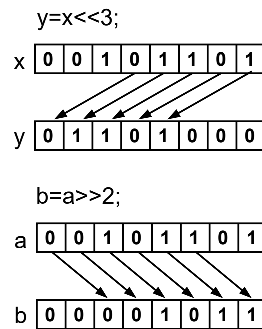
C is een taal, die dicht bij de hardware staat, daarom is het niet vreemd dat deze taal ook een aantal bitbewerkingen kent. Bij het programmeren van microcontrollers moeten bits uit allerlei registers gemanipuleerd kunnen worden. Bitbewerkingen zijn daarbij essentieel. C kent vier logische bitbewerkingen en twee schuifoperatoren. Al deze bewerkingen worden uitgevoerd op gehele getallen. Figuur 9.8 laat de vier logische bitbewerkingen zien met een achtbits getal (**`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`.

```
0 0 1 0 1 1 0 1       0 0 1 0 1 1 0 1       0 0 1 0 1 1 0 1
0 0 0 0 1 0 1 1  &    0 0 0 0 1 0 1 1  |    0 0 0 0 1 0 1 1  ^    0 0 0 0 1 0 1 1  ~
---------------       ---------------       ---------------       ---------------
0 0 0 0 1 0 0 1       0 0 1 0 1 1 1 1       0 0 1 0 0 1 1 0       1 1 1 1 0 1 0 0
```

Figuur 9.8 : 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.

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 9.9 : De schuifoperatoren.
Bij << worden de bits naar links geschoven en komen er rechts nullen bij. Bij >> worden de bits naar rechts geschoven en komen er links nullen bij.

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 9.9 toont het schuiven voor de onderstaande 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;             //  b is 0x00001011 (11)
```

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 de meest significante bit weg en krijgt `y` de waarde `360-256=104`.

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.

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.

9.13 Verkorte schrijfwijze bij toekenningen

Op pagina 27 zijn de incrementoperator (++) en de decrementoperator (--) al kort besproken. De ++ en -- operatoren verhogen of verlagen de waarde van een variabele met 1. Deze operatoren zijn op twee manieren te gebruiken. Bij de *pre*-notatie staat de operator voor de variabele en bij de *post*-notatie staat de operator achter de variabele.

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

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 9.7 geeft alle toekenningsoperatoren met een voorbeeld en de betekenis van het voorbeeld.

Tabel 9.7 : 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`

9.14 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:

$$20/2 * 5 = 20/(2 * 5) = 2 \quad \text{associatie van rechts naar links}$$
$$20/2 * 5 = (20/2) * 5 = 50 \quad \text{associatie van links naar rechts}$$

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 9.8 : Prioriteit en associativiteit van operatoren

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

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 9.8. 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.

Neveneffecten

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 bewerkin-

gen 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.

9.15 Voorbeeld: afdrukken binaire waarden

In code 9.1 is de functie `printb` gebruikt, die een geheel getal binair afdrukt. Deze functie was nodig omdat `printf` standaard geen *format specifier* voor een binaire representatie kent. De implementatie van deze functie staat in code 9.6 en gebruikt allerlei rekenkundige, logische, bit- en schuifbewerkingen.

Sommige C-compilers kennen de *format specifier* `%b` om getallen binair af te drukken. Dit creëert in ieder geval geen portabele code.

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 }
```

Met een **`for`**-lus worden alle bits vanaf het meest significante bit een voor een geëvalueerd. Als de bit hoog is, wordt er een 1 en als de bit laag is, wordt er een `0` afgedrukt. Als de parameter `space` hoog is, wordt er tussen iedere vier bits een lage streep (`'_'`) afgedrukt.

Uitleg code 9.6 regel 1
`long long`

De parameter `x` is van het type **`long long`**, daardoor kan de functie voor alle soorten gehele getallen gebruikt worden. Als het type **`unsigned`** is wordt het getal met nullen uitgebreid en met enen of nullen als het **`signed`** is. Deze uitbreiding kan worden afgedrukt, mits het aantal bytes `nbytes` correct is opgegeven:

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

printb(i, 8, 1);        // print: 0000_0000_0000_0000_0011_0000_0011_1001
printb(u, 2, 0);        // print: 10010110
printb(u, 4, 0);        // print: 0000000010010110
```

Het bepalen van het aantal bytes kan ook met de **`sizeof`**`()` operator. De aanroep van `printb` luidt dan:

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

In paragraaf 9.16 wordt op bladzijde 116 de **`sizeof`**-operator besproken.

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

Om de bits te selecteren wordt er een masker `mask` gebruikt met de lengte (`nbits`) van het af te drukken getal. De meest significante bit krijgt de waarde 1, de andere bits zijn `0`. 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 bewerking (&) maakt de uitdrukking `x & mask` waar is als de meest significante bit hoog is. De bitbewerkingen zijn in paragraaf 9.12 besproken. De bitbewerking `&` voert een bitsgewijze EN uit. Dat betekent dat een bit alleen hoog is als de betreffende bits van beide operanden hoog zijn.

Figuur 9.10 : Bitmaskering voor test of meest significante bit hoog is.

Figuur 9.10 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 9.13 De operator `<<` is een schuifoperator en schuift de bits in dit geval één positie naar links, zie ook paragraaf 9.12. Samen met de gekozen bitmaskering zorgt dit er voor dat de meest significante bit eerst wordt afgedrukt en de minst significante bit het laatst.

9.16 Meer over operatoren, datatypen en declaraties

Deze paragraaf bespreekt een aantal sleutelwoorden, operatoren en datatypen, die in de rest van het boek gebruikt worden.

Integers met vaste afmetingen

Omdat de afmeting van de standaard integers in C niet vastligt en van systeem tot systeem kan verschillen, zijn er integer gedefinieerd met een vaste grootte. Het headerbestand `stdint.h`, dat ook onderdeel is van `inttypes.h`, bevat de definities van deze integers. Het datatype `uint8_t` is een 8-bits unsigned integer en `int8_t` een 8-bits signed integer. Het headerbestand bevat soortgelijke definities voor in ieder geval 16, 32 en indien mogelijk ook voor 64 getallen. Vooral bij microcontrollers wordt deze integers veel gebruikt, omdat ze een eenduidige afmeting hebben. Het headerbestand `avr/io.h` van de Xmega sluit automatisch `stdint.h` in.

Tabel 9.9 : Integers met een vaste afmeting.

naam	betekenis	bits
int8_t	signed	8
int16_t	signed	16
int32_t	signed	32
uint8_t	unsigned	8
uint16_t	unsigned	16
uint32_t	unsigned	32

Het sleutelwoord `const`

C kent ook constanten. In C is een constante is een variabele die niet van waarde kan veranderen. Dit wordt aangegeven met het sleutelwoord **`const`**:

```
const int  c = 10;
const int  d = 12 + (2 * 3) - 8;
const char text[] = "tekst";
```

De constante `d` krijgt ook de waarde 10, omdat het rechter lid eerst geëvalueerd en daarna de uitkomst aan `d` wordt toegekend. De waarden van `c`, `d` en de tekst van `text` kunnen hierna niet meer gewijzigd worden.

Dit werkt ook bij pointers, maar de positie van **`const`** bij de declaratie bepaalt of de pointer constant is of hetgeen waar de pointer naar wijst constant is:

```
    const int * variablePointerToConstantInteger
    int const * variablePointerToConstantInteger
    int * const constantPointerToVariableInteger
```

In veel gevallen wordt een constante niet met **`const`** gedefinieerd, maar met een definitie of een enumeratie:

```
#define   a   10
enum  {b = 10};
const int c = 10;
```

Suffix bij gehele en gebroken getallen

Zonder suffix bepaalt de compiler de afmeting die nodig is voor een geheel getal. Met één of meerdere suffixen kan expliciet worden gemaakt dat een getal een **`long long`** of een **`unsigned long`** is. De letters `U` en `u` betekenen **`unsigned`** en de letters `L` en `l` betekenen **`long`**. De hoofd- en kleine letters mogen door elkaar worden gebruikt.

```
  unsigned int u1       = 123U;
  unsigned long u2      = 123UL;
  unsigned long long u3 = 123ULL;
  signed long u2        = 123L;
  signed long long u3   = 123LL;
```

In code 9.6 is op regel 5 de suffix `ULL` gebruikt. Zonder suffix is het getal 1 geen 64 bits breed. Bij grote waarden schuift deze 1 links het getal uit en krijgt `mask` de waarde nul.

Voor gebroken getallen kent C ook suffixen. De letters `F` en `f` betekenen **float** en `L` en `l` staan weer voor **long**. Zonder suffix is het een **double**, met `F` een **float** en met `L` een **long double**:

```
double       x1   = 1.0;
float        x2   = 1.02e-4F;
long double  x3   = 3.14L;
```

In figuur 9.11 staat drie keer hetzelfde voorbeeld met steeds andere suffixen. Aan de **float** `f` wordt het gebroken getal `0.123` toegekend. In de linker figuur wordt de float `f` vergeleken met de **double** `0.123`, Een gebroken getal in C is bijna nooit de exacte waarde en daardoor verschillen de **float** en de **double**. In de twee rechter figuren is vanwege de suffix `0.123` expliciet een **float**.

```
float f = 0.123;

if (f == 0.123) {
  printf("gelijk");
} else {
  printf("niet gelijk");
}
```

Drukt `niet gelijk` af, omdat `f` een **float** en `0.123` een **double** is.

```
float f = 0.123;

if (f == 0.123f) {
  printf("gelijk");
} else {
  printf("niet gelijk");
}
```

Drukt `gelijk` af, omdat `f` en `0.123f` allebei **float** zijn.

```
float f = 0.123f;

if (f == 0.123f) {
  printf("gelijk");
} else {
  printf("niet gelijk");
}
```

Drukt `gelijk` af, omdat `f` en `0.123f` allebei **float** zijn.

Figuur 9.11 : Het effect van het gebruik van de suffix `f` bij gebroken getallen.

Enumeratie

Een enumeratie creëert eigen symbolische namen voor een lijst met overeenkomstige zaken. De dagen van de week worden met een enumeratie `days_enum` als volgt gedefinieerd:

```
enum days_enum { SUNDAY, MONDAY, TUESDAY,
                 WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };
enum days_enum  day = THURSDAY;
```

De dagen staan tussen accolades en zijn symbolische namen voor de waarden 0 tot en met 6: `SUNDAY` staat voor 0, `MONDAY` staat voor 1 en zo verder. Bij de toekenning krijgt `day` de waarde 4. De symbolische namen zijn constanten en worden daarom vaak met hoofdletters geschreven. De elementen uit de lijst kunnen ook een andere waarde krijgen, zoals:

```
enum days_enum { SUNDAY=1, MONDAY=2, TUESDAY=4,
                   WEDNESDAY=8, THURSDAY=16, FRIDAY=32, SATURDAY=64 };
```

Vaak wordt de enumeratie direct in een typedefinitie geplaatst. Een eigen type maakt de declaratie van variabelen duidelijker:

```
typedef enum days_enum { SUNDAY, MONDAY, TUESDAY,
                         WEDNESDAY, THURSDAY, FRIDAY, SATURDAY } days_t;


days_t day = THURSDAY;
```

De naam van de enumeratie `days_enum` mag in dit geval bij de typedefinitie worden weggelaten.

Code 9.7 : **Het gebruik van de `sizeof`-operator.**

```
1  #include <stdio.h>
2
3  typedef struct naw {        // data structure
4    char naam[128];
5    char adres[128];
6    char woonplaats[128];
7  } naw_t;
8
9  int main(void)
10 {
11     int     i;
12     double  d;
13     int     ia[10];
14     char    ca[3][7];       // two dimensional array
15     naw_t   n;
16
17     printf("sizeof(i)   : %d\n", sizeof(i));   // sizeof(i)   : 4
18     printf("sizeof(d)   : %d\n", sizeof(d));   // sizeof(d)   : 8
19     printf("sizeof(ia)  : %d\n", sizeof(ia));  // sizeof(ia)  : 40
20     printf("sizeof(ca)  : %d\n", sizeof(ca));  // sizeof(ca)  : 21
21     printf("sizeof(n)   : %d\n", sizeof(n));   // sizeof(n)   : 384
22
23     return 0;
24 }
```

Het samengestelde type **`struct`**, waarmee een datastructuur wordt vastgesteld, wordt in paragraaf 13.5 besproken.

De operator `sizeof`

Tweedimensionale arrays komen in paragraaf 10.7 bij de multidimensionale arrays aan de orde.

De operator **`sizeof`**`()` bepaalt de grootte van een datatype uitgedrukt in het aantal bytes. In code 9.7 staan een aantal voorbeelden. De integer `i` is vier bytes groot en de **`double`** `d` is acht bytes.

Deze operator wordt vooral gebruikt bij dynamische geheugenallocatie, dit onderwerp komt aan de orde in paragraaf 11.5. Als er een geheugenruimte van `N` integers nodig is wordt de ruimte die nodig is berekend met `N * sizeof (int)`:

```
int *p = (int *) malloc(N * sizeof(int));
```

Een ander voorbeeld is de afmeting van de buffer bij de functie `getScore` uit paragraaf 5.4. Omdat een **`char`** één byte groot is, kan in code 5.7 op regel 11 `MAXBUF` vervangen worden door **`sizeof`**`(buffer)`:

```
fgets(buffer, sizeof(buffer)-1, stdin)
```

Als de elementen van array groter zijn dan één byte, kan het aantal elementen berekend worden. Het aantal elementen `n` van integerarray `ai` uit code 9.7 is:

```
n = sizeof(ai)/sizeof(int);
```

Typedefinitie

Het sleutelwoord **`typedef`** definieert een naam, die gebruikt kan worden als synoniem voor een type of een afgeleid type. In tegenstelling tot een **`struct`** of een **`enum`** creëert **`typedef`** geen nieuwe datatypes; het geeft alleen een andere naam voor een bestaand type. De syntax is:

```
typedef type-declaration synonym;
```

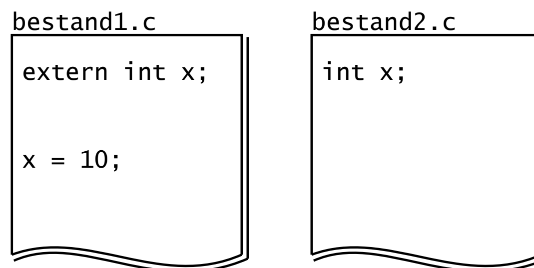
Een voorbeeld is:

```
typedef double resistance_t;
resistance_t = 45.6;
```

Het is een goede gewoonte om de naam van de typedefinitie te laten eindigen op `_t`, zodat duidelijk herkenbaar is dat het om een type gaat. Dit is ook het geval bij de typedefinities van de integers met een vaste afmeting, zoals `uint16_t`.

Externe variabelen

Het sleutelwoord **`extern`** bij de declaratie van een variabele betekent dat er voor de variabele geen geheugenruimte wordt gereserveerd en dat deze variabele al op een andere plaats volledig is gedeclareerd.



Figuur 9.12 : De toepassing van een externe variabele. Bestand `bestand2.c` declareert een variabele `x`. Bestand `bestand1.c` gebruikt deze zelfde variabele en daarom wordt `x` in `bestand1.c` **`extern`** gedeclareerd.

In figuur 9.12 staat een `bestand2.c` waarin een variabele `x` is gedeclareerd. Deze variabele krijgt in `bestand1.c` een waarde. De compiler moet dan weten hoe `x` eruit ziet. Zonder het sleutelwoord **`extern`** zijn er twee verschillende variabelen `x`. Met het sleutelwoord **`extern`** weet de compiler bij de compilatie van bestand `bestand1.c` hoe `x` eruit ziet, maar maakt de compiler geen nieuwe variabele voor `x` aan.

Statische variabelen en statische functies

Een functie of een variabele kan statisch zijn door voor de functiedeclaratie en de variabele declaratie het sleutelwoord **`static`** te plaatsen. Er zijn twee betekenissen te onderscheiden:

- Als de statische variabele in een functie staat, blijft de waarde van de variabelen behouden tussen twee aanroepen.
- Een statische functie en een globale statische variabele, dat is een variabele die niet in een functie staat, zijn alleen zichtbaar in het bestand waarin deze gedeclareerd zijn.

De laatste methode wordt bij bibliotheken gebruikt om functie en variabelen af te schermen van de gebruiker. C kent geen *private* en *public* functies zoals Java deze kent. In principe zijn in C alle functies *public*. Met **`static`** kunnen functies en globale variabelen *private* gemaakt worden.

Code 9.8 toont het effect van een lokale statische variabele. De gewone lokale variabele `n` wordt bij elke functieaanroep opnieuw gecreëerd en geïnitialiseerd.

Code 9.8 : Het verschil tussen een lokale statische en niet-statische variabele.

```
1  #include <stdio.h>
2
3  void static_nonstatic(void)
4  {
5    int n        = 1;
6    static int s = 1;
7
8    printf("non static %d  static %d\n", n++, s++);
9  }
10
11 int main(void)
12 {
13   static_nonstatic();   // prints:  non static 1  static 1
14   static_nonstatic();   // prints:  non static 1  static 2
15   static_nonstatic();   // prints:  non static 1  static 3
16
17   return 0;
18 }
```

Bij het afdrukken heeft deze variabele altijd de waarde 1. De statische variabele `s` blijft na afloop in het geheugen staan en behoudt zijn waarde. Bij iedere volgende aanroep bestaat `s` al en behoudt `s` de originele waarde.

Het sleutelwoord `volatile`

Het Engelse woord *volatile* betekent veranderlijk of vluchtig. Met het sleutelwoord **`volatile`** wordt in C aangegeven dat een variabele ook op een andere plaats veranderd kan worden. Dit wordt veel gebruikt bij microcontrollerprogramma's met interruptroutines. Als het hoofdprogramma een variabele gebruikt, die door een interruptroutine gewijzigd wordt, moet de variabele **`volatile`** zijn. De compiler ziet het hoofdprogramma en de interruptroutine als twee aparte onderdelen en behandelt de variabele in het hoofdprogramma dan als een constante, die nooit van waarde verandert.

Pointers mogen ook volatile zijn. In dat geval is de positie van **`volatile`** in de declaratie belangrijk:

```
volatile int *p;   // p is a pointer to a volatile int
int* volatile q;   // q is a volatile pointer to an int
```

De classificaties `auto` en `register`

Naast **`static`** en **`extern`** kent C nog twee sleutelwoorden die de classificatie van een variabele beïnvloeden: **`auto`** en **`register`**. Omdat **`auto`** de standaardclassificatie is voor een variabele, is er geen enkele reden om voor een variabele **`auto`** te plaatsen. De classificatie **`register`** geeft de compiler de aanwijzing dat de variabele in één van de registers van de processor moet worden opgeslagen. De compiler kan deze aanwijzing negeren. Als een variabele als registervariabele wordt opgeslagen, mag de adresoperator, `&`, niet worden gebruikt.

Door de registers van de processor voor variabelen te gebruiken kan de code sneller en efficiënter worden.

10

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: ▪ De declaratie van arrays. ▪ Toewijzen aan en aanroepen van arrays. ▪ Lezen en schrijven buiten het bereik van een array. ▪ Meerdimensionale arrays. ▪ De declaratie bij meerdimensionale arrays. ▪ De toewijzingen bij meerdimensionale arrays. De voorbeelden gebruiken arrays voor het berekenen van de getallen van Fibonacci en het creëren van de driehoek van Pascal: ▪ Het berekenen van de getallen van Fibonacci en de Gulden Snede. ▪ Het afdrukken van een tweedimensionaal array. ▪ Het vullen en afdrukken van een tweedimensionaal array. ▪ Het afdrukken van de driehoek van Pascal en de diagonaal met de getallen van Fibonacci.

In paragraaf 3.3 is de array geïntroduceerd en in de voorgaande hoofdstukken op verschillende plaatsen 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 10.1 : 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.

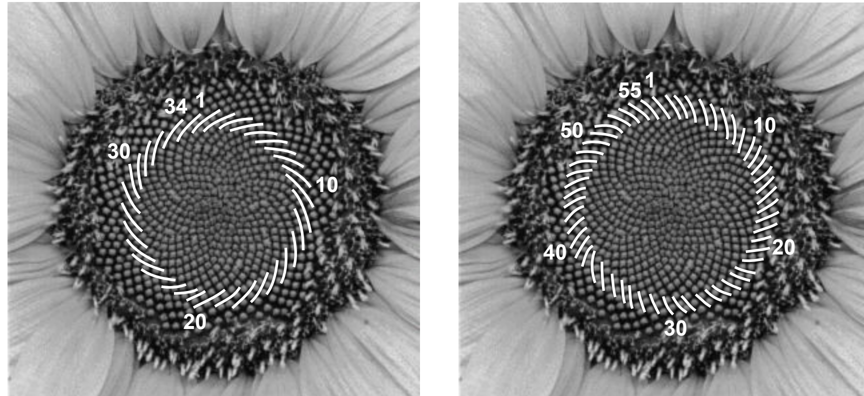
10.1 De getallen van Fibonacci en de Gulden Snede

Het programma uit code 10.1 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 Φ (*Phi*) en is gelijk aan:

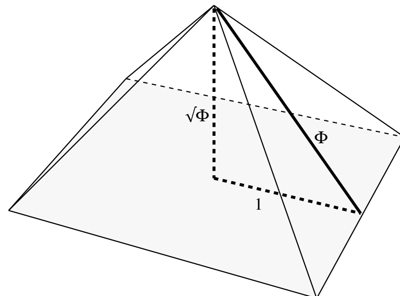
$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1{,}618 \tag{10.1}$$

In figuur 10.2 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 10.2 : 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 10.3 : 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.

10.2 Berekenen getallen van Fibonacci en de Gulden Snede

Het programma uit code 10.1 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 10.1 : 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 10.1 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 deze 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 10.1 Φ en drukt dit getal af.

De uitvoer van het programma staat in figuur 10.4 en laat zien dat de verhouding tussen twee opeenvolgende getallen van Fibonacci bij grote waarden de Gulden Snede steeds dichter nadert.

```
/cc/array
/cc/array $ gcc -o fibonacci fibonacci.c

/cc/array $ fibonacci
    1     1     1.0000000
    2     1     2.0000000
    3     2     1.5000000
    5     3     1.6666667
    8     5     1.6000000
   13     8     1.6250000
   21    13     1.6153846
   34    21     1.6190476
   55    34     1.6176471
   89    55     1.6181818
  144    89     1.6179775
  233   144     1.6180556
  377   233     1.6180258
  610   377     1.6180371
  987   610     1.6180328
 1597   987     1.6180344
 2584  1597     1.6180338
 4181  2584     1.6180341
 6765  4181     1.6180340
10946  6765     1.6180340
                Dit nadert tot 1.6180340 = (1+sqrt(5))/2
/cc/array $
```

Figuur 10.4 : De uitvoer van code 10.1 toont aan dat de verhouding tussen twee opeenvolgende getallen van Fibonacci nadert tot Φ.

10.3 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.

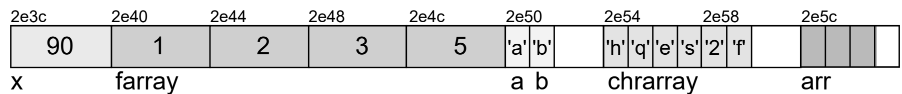
```
int  x = 90;
int  farray[4] = {1, 2, 3, 5};
char a = 'a';
char b = 'b';
char chrarray[] = {'h','q','e','s','2','f'};
char arr[3];
```

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 10.5 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.



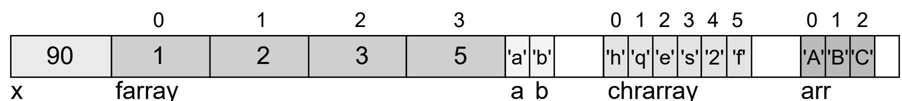
Figuur 10.5 : De geheugenorganisatie bij de declaratie van arrays. De `int` variabele `x` neemt vier bytes in beslag. De array `farray` heeft 24 bytes nodig. De variabelen `a` en `b` zijn elk een byte groot. Voor `chrarray` zijn zes bytes nodig en voor `arr` worden drie bytes gereserveerd.

10.4 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 10.6 laat dit zien en toont de indices van de drie arrays.



Figuur 10.6 : 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` één 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'
```

10.5 Lezen buiten het bereik van een array

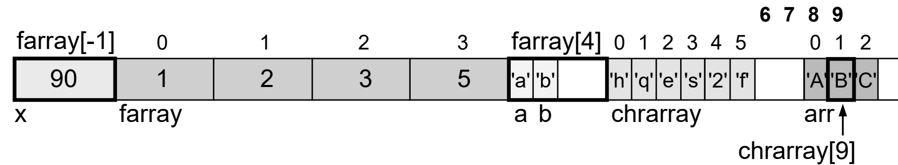
De hoeveelheid geheugen, die nodig is voor een array, wordt bij de declaratie vastgelegd. De taal C houdt op geen enkele manier bij of er ook netjes binnen dit deel

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
```

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 10.5 tot en met 10.8 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 10.7 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.



Figuur 10.7 : Lezen buiten het bereik van een array. Met een vette rechthoek zijn de elementen aangegeven, die in bovenstaand voorbeeld worden afgedrukt.

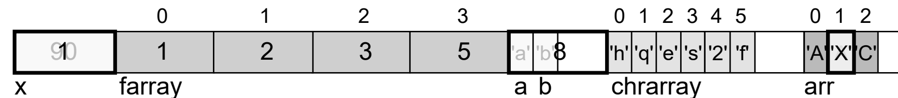
De tweede regel drukt een element af dat precies voor de beginpositie van `farray` staat. In dit geval is dat de variabele `x`. De derde regel drukt de inhoud van een cel ter grootte van een **`int`** af een positie na de array. In dit geval zijn dat de **`char`**'s `a` en `b` en wat rommel uit de rest 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.

10.6 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 10.8 : Schrijven buiten het bereik van een array. Met een vette rechthoek zijn de elementen aangegeven, die in bovenstaand voorbeeld worden overschreven.

10.7 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.

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 10.9. 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`.

kolom i

rij j	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0
5	1	5	10	10	5	1	0	0	0
6	1	6	15	20	15	6	1	0	0
7	1	7	21	35	35	21	7	1	0
8	1	8	28	56	70	56	28	8	1

array[j][i]

rij kolom

Figuur 10.9 : **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.

Toewijzingen bij een multidimensionaal array

De toewijzing aan een element van een meerdimensionale array gaat op dezelfde manier als bij eendimensionaal array:

```
z[0][0] = 9;
c[4][5] = ’a’;
f[1][0] = 1.62;
```

Meerdimensionale arrays zijn altijd lastig. Zorg dat de methode van indexeren altijd consequent is.

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 10.2 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 10.2 staat in figuur 10.10.

```
/cc/array
/cc/array $ gcc -o multiarray1 multiarray1.c

/cc/array $ multiarray1
4 5 6
3 2 1

/cc/array $
```

Figuur 10.10 : De uitvoer van code 10.2.

Code 10.2 : 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 10.3 : 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 10.3 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 10.11

```
/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 10.11 : De uitvoer van code 10.3.

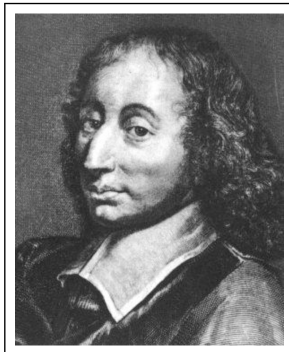
10.8 De driehoek van Pascal

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 10.13 toont voor 0 tot en met 4 de macht van de som van twee getallen en het bijbehorende deel van de driehoek van Pascal.

$$
\begin{aligned}
(a+b)^0 &= 1 \\
(a+b)^1 &= a+b \\
(a+b)^2 &= a^2+2ab+b^2 \\
(a+b)^3 &= a^3+3a^2b+3ab^2+b^3 \\
(a+b)^4 &= a^4+4a^3b+6a2b^2+4ab^3+b^4
\end{aligned}
$$

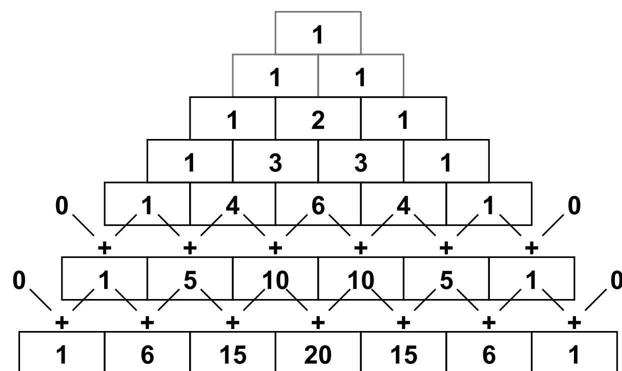
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Figuur 10.13 : 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 10.12 : 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 10.14. 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.

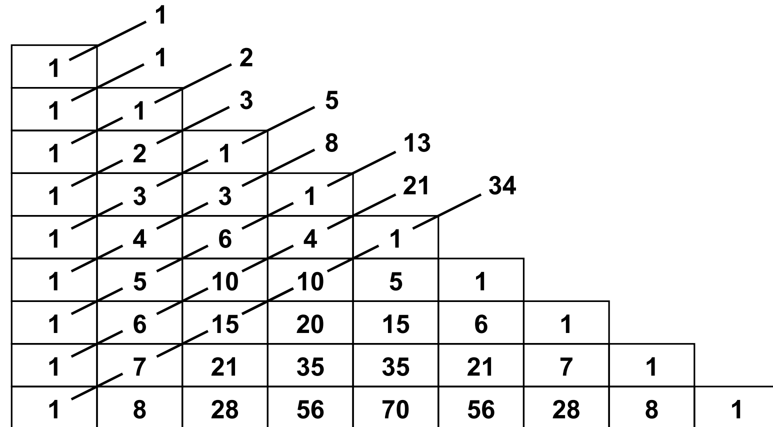


Figuur 10.14 : De opbouw van de driehoek van Pascal. Een getal uit de driehoek is de som van de twee bovenliggende getallen.

Er zijn meerdere manieren om de driehoek van Pascal op te schrijven. In figuur 10.15 zijn alle elementen links uitgelijnd. De som van de getallen op de diagonalen levert dan de reeks van Fibonacci op.

10.9 Berekening driehoek van Pascal en getallen van Fibonacci

Het programma van code 10.4 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 10.15 : **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.

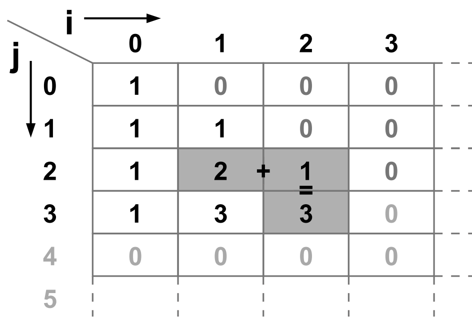
De getallen van de driehoek worden in een tweedimensionaal array `triangle` bewaard. In dit geval heeft deze array evenveel rijen als kolommen. Omdat het over een driehoek gaat, worden niet alle hokjes uit de array gebruikt. Voor de som van de diagonalen is een eendimensionaal array `diagonal` nodig. De afmetingen van triangle en de afmeting van `diagonal` zijn vastgelegd met de constante `DIMENSION`. In dit voorbeeld is deze constante negen.

Uitleg code 10.4 regel 12-16

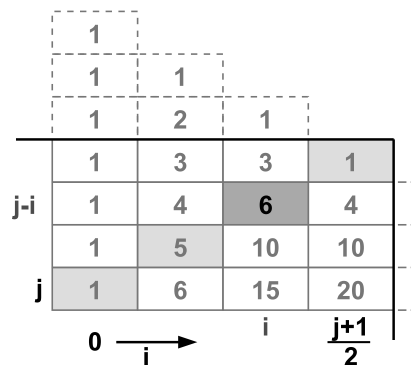
Het programma bestaat uit vier delen. Omdat bij het vullen van een rij de getallen uit de vorige rij gebruikt worden, wordt eerst de array `triangle` met nullen gevuld. Bij de declaratie wordt een array immers niet automatisch geïnitialiseerd. Er zou dus van alles in het geheugen kunnen staan.

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 10.16 : **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 10.17 : **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 10.16 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 10.4 : 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 10.4 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 10.17 is weergegeven. De index van de rij moet steeds met één verlaagd worden en de index van de kolom met één verhoogd.

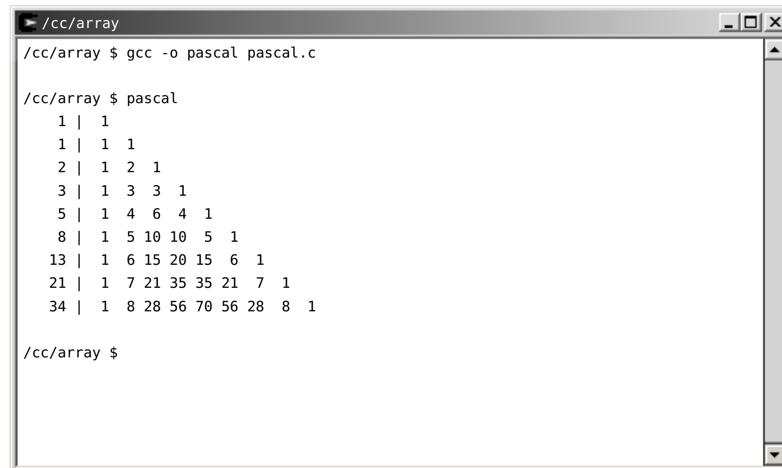
Bij kolom `i` is de index `j` van de rij `i` lager, dus: `j-i`. Het eerste element is altijd 1. De som van de diagonaal voor rij `j` is zodoende:

```
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 10.18



Figuur 10.18 : De uitvoer van code 10.4 geeft de driehoek van Pascal en geeft voor elke regel de som op de diagonaal.

10.10 Dynamische geheugenallocatie

Een nadeel bij een array is dat de afmeting vooraf bekend moet zijn. Bij de compilatie reserveert de compiler de benodigde geheugenruimte. In veel gevallen is dan onbekend hoeveel ruimte er nodig is. De grootte hangt bijvoorbeeld af van de invoer van de gebruiker.

Een oplossing is om de arrays zo groot te maken, dat de gegevens altijd in de arrays passen. Vooral bij meerdimensionale arrays heeft de applicatie dan vaak veel computergeheugen nodig.

C kent een aantal mogelijkheden om arrays dynamisch te alloceren. Dynamische geheugenallocatie betekent dat er tijdens de uitvoering van het programma extra geheugenruimte wordt toegewezen. Geen van deze mogelijkheden is triviaal en bij een aantal wordt er intensief gebruik gemaakt van pointers. Daarom wordt dit onderwerp besproken bij de pointers in de paragrafen 11.8 en 11.10.

11

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: ▪ De declaratie van een pointer. ▪ De toewijzing aan een pointer. ▪ Het overnemen van de inhoud van de plaats waar de pointer naar wijst. ▪ De adresoperator (&) en de dereferentie-operator (∗). ▪ Rekenen met pointers. ▪ Fouten bij pointers. ▪ Toepassingen met pointers. ▪ Dynamische geheugenallocatie bij één- en tweedimensionale arrays. ▪ De *variable length array*. Voorbeelden met pointers zijn: ▪ Berekenen van de getallen van Fibonacci en de Gulden Snede. ▪ De functies `strcpy` en `strcmp`. ▪ Geheugenallocatie van een eendimensionaal arrays. ▪ Voorbeeld met een *variable length array*. ▪ Geheugenallocatie voor tweedimensionaal array.

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 10 en strings komen aan bod in hoofdstuk 12.

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 4.5, bij de uitleg van de functie `scanf` in paragraaf 5.2 over de geformatteerde in- en uitvoer. De adresoperator `&` is daar ook al aan de orde gekomen. 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 het voorbeeld van code 10.1 wordt herschreven met pointers in plaats van arrays.

11.1 Declaraties van pointers

Een pointer wordt gedeclareerd door een asterisk (∗) bij 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 11.1 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));
```

11.2 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.

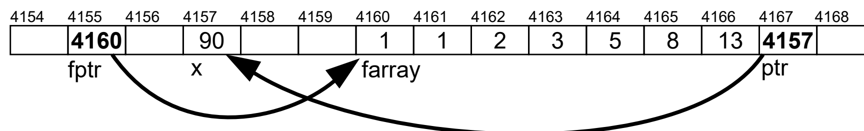
In C wordt `&` gebruikt als bitwise and-operator `z=a&b`, als logische and-operator `c&&d` en als adresoperator `ptr=&x`.

```
int *ptr;
int  x = 90;
int  farray = {1, 1, 2, 3, 5, 8, 13};
int *fptr;

ptr  = &x;
fptr = farray;
```

De array `farray` kan ook worden beschouwd als een pointer naar de geheugenplaats waar de elementen van de array staan. De variabele `farray` is dan feitelijk het adres van het eerste element uit de array. Omdat `farray` ook een pointer is, staat er bij de toekenning aan `fptr` voor de variabele `farray` geen adresoperator. 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]` aan `fptr` toegekend.
Figuur 11.1 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 11.1 : 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.

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 Engels *dereference operator*. In het Nederlands wordt dit soms vertaald met dereferentie- of indirectie-operator.

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.

```
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
```

11.3 Rekenen met pointers

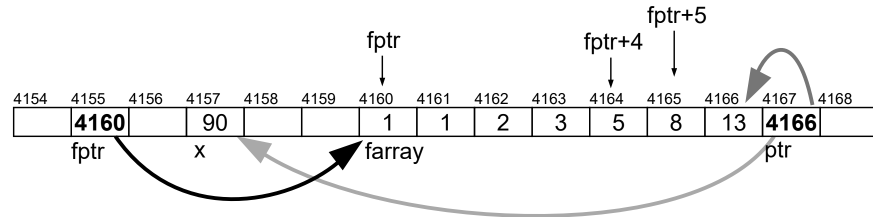
Pointers zijn gewone getallen waar mee gerekend kan worden. Bewerkingen als vermenigvuldigingen of delen van pointers zijn weinig zinvol, maar het optellen en aftrekken van een constante bij een pointer of het aftrekken van twee pointers is wel zinvol. Dit optellen en aftrekken van pointers wordt pointer 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 11.2. 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:

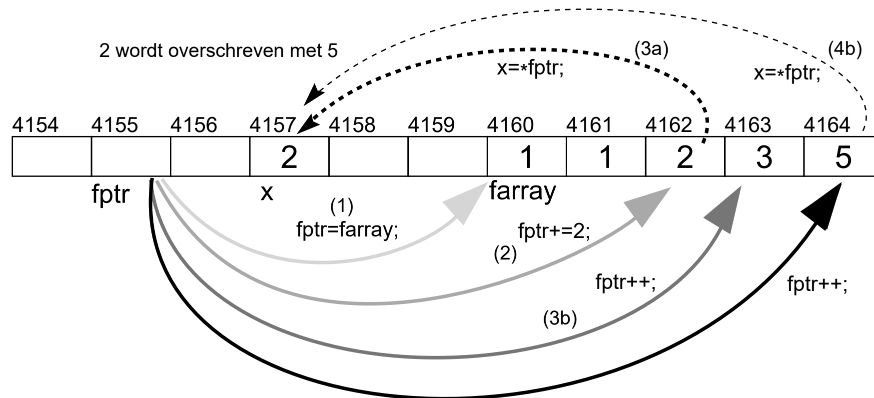
```
fptr  = farray;
fptr += 2;
x     = *fptr++;
x     = *++fptr;
```

Figuur 11.3 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 bestaat uit twee bewerkingen: eerst (`x=*fptr`) wordt de inhoud waar de pointer



Figuur 11.2 : 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.

naar wijst aan `x` toegekend en daarna wordt er een bij opgeteld (`fptr++`). Na de derde bewerking wijst de pointer 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 11.3 : 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).

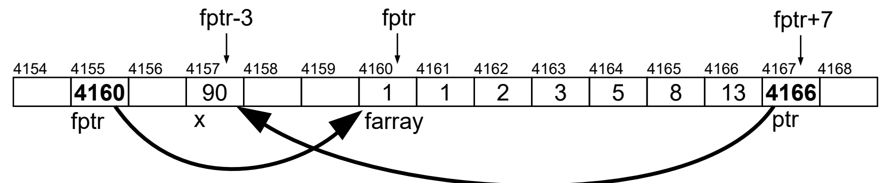
11.4 Fouten met pointers

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 11.4 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.

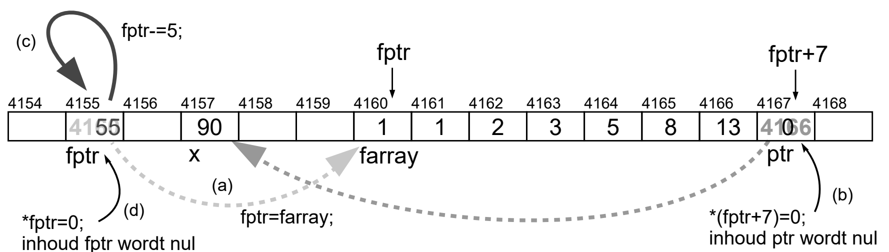


Figuur 11.4 : **Leesfouten met pointers.** De pointer `fptr-3` wijst naar `x` en de pointer `fptr+7` wijst naar `ptr`.

Nog erger wordt het als de inhoud waar de pointers naar wijzen wordt veranderd. Figuur 11.5 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 toewijzing 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 11.5 : **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 `fptr` schuift vijf posities naar links (c). Toevallig is dat het adres van de pointer zelf. Tenslotte overschrijft `fptr` zichzelf met nul (d). Via `fptr` is de array `farray` nu niet meer bereikbaar.

11.5 Berekenen getallen van Fibonacci en Gulden Snede met pointers

Het programma uit code 11.1 onderzoekt opnieuw de relatie tussen de reeks van Fibonacci en de Gulden Snede. Net als het programma van code 10.1 uit paragraaf 10.1 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 11.1 een dynamisch array en worden er pointers gebruikt bij het vullen en bij het afdrukken van de array.

Code 11.1 : 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 11.1 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()`

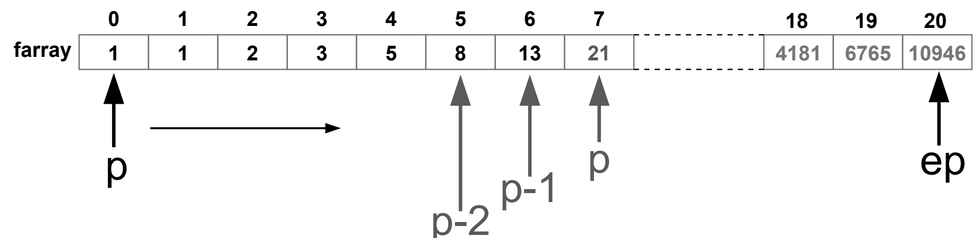
De operator **`sizeof()`** geeft de grootte van een bepaald datatype weer:

```
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 11.6 : **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 11.1 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 11.6.

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;`.

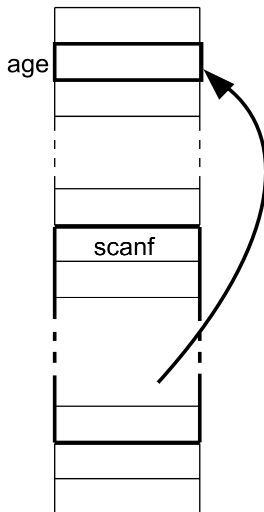
Vervolgens wordt er opnieuw langs de array gegaan en wordt steeds de inhoud waar de pointers `p` en `p-1` naar wijzen en het quotiënt van deze waarden afgedrukt.

11.6 Toepassingen pointers

Het programma van code 11.1 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 5.2

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

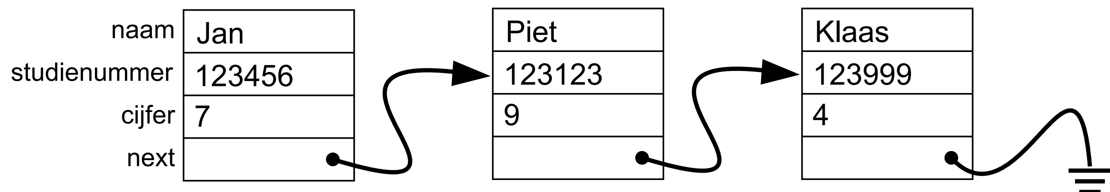


Figuur 11.7 : Pointers bij functies. Door aan de functie `scanf` het adres van `age` mee te geven, vult `scanf` het resultaat op de locatie `age` in.

  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 11.7.
- *Arrays*
  Zoals voorbeeld 11.1 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 12 behandelt de strings. Paragraaf 11.7 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 11.8 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 13.5 laat zien hoe dat gaat bij een lijst.



Figuur 11.8 : Voorbeeld van een lijst met datastructuren.

11.7 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.

11.8 Dynamische geheugenallocatie bij eendimensionale arrays

In paragraaf 10.10 is al aangegeven dat er bij C meerdere methoden zijn om dynamisch geheugenruimte voor arrays te alloceren. Voor eendimensionale arrays is dat eenvoudiger dan voor meerdimensionale arrays. Deze paragraaf bespreekt de dynamische allocatie bij eendimensionale arrays en paragraaf 11.10 die bij tweedimensionale arrays.

Er zijn in principe twee methoden om een eendimensionaal array dynamisch te alloceren:

- met behulp van de functie `malloc`,
- met een zogenoemd *variable length array* of VLA.

Het programma uit code 11.2 heeft één argument dat bij de aanroep het aantal elementen van de array bevat. Op regel 6 is een pointer `array` gedeclareerd. De functie `malloc` maakt op regel 16 voldoende ruimte vrij voor een array met `n` integers.

Op regel 11 en 17 is `stderr` in plaats van `stdout` gebruikt. Bij beide wordt in principe de tekst naar het scherm geschreven. Alleen bij *IO-redirection* is er onderscheid. Bij deze aanroep van programma `a`:

```
a 4 > res.out
```

wordt alles naar `stdout` in bestand `res.out` geplaatst en komt een foutmelding met `stderr` op het scherm. Bij:

```
a 4 > res.out 2> err.out
```

komt de foutmelding in `err.out` en niet op het scherm.

Code 11.2 : Geheugenallocatie van een eendimensionaal array met `malloc`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6    int *array;
7    int n;
8    int i;
9
10   if ( (n = atoi(argv[1])) == 0 ) {
11     fprintf(stderr, "Usage: %s <number>\n", argv[0]);
12     return 1;
13   }
14
15   // allocate memory for array
16   if ( (array = malloc(n * sizeof(int))) == NULL ) {
17     fprintf(stderr, "out of memory\n");
18     return 2;
19   }
20
21   // fill array
22   for(i = 0; i < n; i++) {
23     array[i] = i;
24   }
25
26   // print array
27   for(i = 0; i < n; i++) {
28     printf("%d\n", array[i]);
29   }
30
31   return 0;
32 }
```

Het nu gevormde array wordt eerst gevuld met opeenvolgende getallen, die daarna worden afgedrukt. Bij de toewijzing op regel 22 is, net als bij de `printf` op regel 28, de normale notatie `array[i]` voor het i^{e} array-element gebruikt.

Code 11.3 : Geheugenallocatie bij een eendimensionaal *variable length array*.

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

int main(int argc, char *argv[])
{
  int n;
  int i;

  if ( (n = atoi(argv[1])) == 0 ) {
    fprintf(stderr, "Usage: %s <number>\n", argv[0]);
    return 1;
  }

  // declaration of a variable length array
  int array[n];

  // fill array
  for(i = 0; i < n; i++) {
    array[i] = i;
  }

  // print array
  for(i = 0; i < n; i++) {
    printf("%d\n", array[i]);
  }

  return 0;
}
```

Een tweede mogelijkheid om een array dynamisch te alloceren is een zogenoemde *variable length array* of VLA te gebruiken. Deze feature is bij versie C99 en GNU99 aan de taal toegevoegd.
In code 11.3 is op regel 15 een *variable length array* gedeclareerd. Bij de compilatie van de code is het aantal elementen `n` nog niet bekend. Nadat het programma is gestart, bepaalt — uit het argument dat aan het programma is meegegeven — het programma de waarde van `n`. Op regel 15 wordt automatisch een array van `n` elementen gemaakt dat daarna wordt gevuld met opeenvolgende getallen en wordt afgedrukt.

11.9 VLA: *variable length array*

Bij code 11.3 lijkt de declaratie van `array` veel eenvoudiger dan die bij code 11.2. Dat is echter niet het geval, de *variable length array* kent veel meer restricties en heeft veel meer beperkingen.

Bij de dynamische geheugenallocatie met `malloc` wordt er plek op de zogenoemde *heap* of *data segment* gemaakt. Bij de VLA wordt de array altijd op de stack geplaatst. In principe is er veel minder ruimte op de stack dan op de *heap*. Bij grote arrays kan dit problemen geven.

Een statische arraydeclaratie kan in een functie staan, maar kan ook globaal worden gedeclareerd. Sterker, bij grote arrays is dat juist verstandig omdat globale variabelen altijd op *heap* komen te staan. Een VLA mag nooit globaal gedeclareerd worden en de variabelen die de afmetingen geven moeten bekend zijn voor de declaratie. De declaratie van een VLA staat altijd in een functie. Omdat statistische variabelen op de *heap* komen te staan, mag een VLA ook nooit `static` zijn.

Code 11.4 geeft een voorbeeld waarbij een VLA nuttig kan zijn. Op regel 3 is een VLA gedeclareerd waarin voldoende ruimte is om de strings, waar de pointers `s1` en `s2` naar wijzen, samen in op te slaan.

De functies `ex_vla` uit code 11.4 en code 11.5 controleren niet of er voldoende geheugenruimte beschikbaar is.
Bij de functie uit code 11.5 is dit eenvoudig toe te voegen. De functie `malloc` geeft immers `NULL` terug als er geen ruimte is. Voor de functie uit code 11.4 is er geen mogelijk om te controleren of er voldoende ruimte is.

Code 11.4 : Een zinvol voorbeeld met *variable length array*.

```
1 void ex_vla(char *s1, char *s2)
2 {
3   char s[strlen(s1) + strlen(s2) + 1];
4   strcpy(s, s1);
5   strcat(s, s2);
6   do_something(s);
7 }
```

Code 11.5 : Voorbeeld 11.4 met behulp van `malloc`.

```
1 void ex_vla(char *s1, char *s2)
2 {
3   char *s = malloc(strlen(s1) + strlen(s2) + 1);
4   strcpy(s, s1);
5   strcat(s, s2);
6   do_something(s);
7   free(s);
8 }
```

Code 11.6 : Een foute functie `exvla` zonder geheugenallocatie.

```
1 void ex_vla(char *s1, char *s2)
2 {
3   char *s;
4   strcpy(s, s1);
5   strcat(s, s2);
6   do_something(s);
7 }
```

In code 11.5 staat dezelfde functie zonder VLA, maar met de gewone dynamische geheugenallocatie. Op regel 3 is met `malloc` weer voldoende geheugenruimte gealloceerd. Pointer `s` wijst naar deze geheugenplaats. Na afloop geeft de functie `free` de gealloceerde geheugenruimte weer vrij.

Een veel gemaakte fout staat in de code 11.6. Er is wel een pointer `s` gedeclareerd, maar er is geen geheugenruimte toegewezen. Deze functie zal er in het algemeen voor zorgen dat het programma vastloopt.

11.10 Dynamische geheugenallocatie bij tweedimensionale arrays

Er zijn drie manieren om een tweedimensionaal array dynamisch te alloceren:

- door de functie `malloc` te gebruiken om een array van pointers te maken en daarna voor iedere pointer een array te alloceren met het juiste datatype;
- door met de functie `malloc` één eendimensionaal array te maken waarin alle elementen staan en met pointerrekenen zelf de positie van de elementen te bepalen;
- met behulp van een *variable length array*.

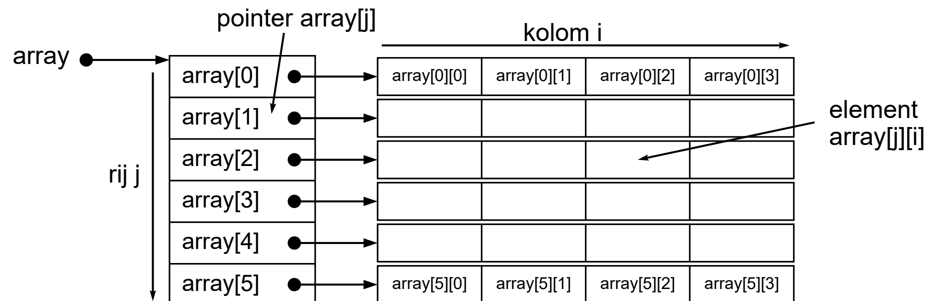
Tweedimensionale array met array van pointers

Een manier om een tweedimensionaal array dynamisch te declareren is door eerst een array met pointers te declareren en daarna voor iedere pointer een array te maken voor het betreffende datatype. In figuur 11.9 is dit getekend voor een array met zes rijen en vier kolommen.

Code 11.7 : Een tweedimensionaal array gebaseerd op een apart pointerarray.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int **build_array(int nrow, int ncolumn)
6  {
7    int **array;
8    int i, j;
9    int n;
10
11   // create pointers to rows
12   array = malloc(nrow * sizeof(int *));
13   if ( array == NULL ) {
14     return NULL;
15   }
16
17   // create rows with integers
18   for(j = 0; j < nrow; j++) {
19     array[j] = malloc(ncolumn * sizeof(int));
20     if ( array[j] == NULL ) {
21       return NULL;
22     }
23   }
24
25   // fill array
26   n = 0;
27   for(j = 0; j < nrow; j++) {
28     for(i = 0; i < ncolumn; i++) {
29       array[j][i] = n++;
30     }
31   }
32
33   return array;
34 }
```

```
37 void print_array(int nrow, int ncolumn,
38                                   int **array)
39 {
40   int i, j;
41   int size = ceil(log10(nrow*ncolumn)) + 1;
42
43   for(j = 0; j < nrow; j++) {
44     for(i = 0; i < ncolumn; i++) {
45       printf("%*d", size, array[j][i]);
46     }
47     printf("\n");
48   }
49 }
50
51 int main(int argc, char *argv[])
52 {
53   int **array;
54   int nrow     = atoi(argv[1]);
55   int ncolumn  = atoi(argv[2]);
56
57   array = build_array(nrow, ncolumn);
58   if ( array == NULL ) {
59     fprintf(stderr, "out of memory\n");
60     return 0;
61   }
62   print_array(nrow, ncolumn, array);
63
64   return 0;
65 }
```



Figuur 11.9 : De declaratie van een tweedimensionaal array met een apart pointerarray. De pointer `array` wijst naar de array van pointers `array[j]`. Iedere pointer wijst weer naar een array met vier elementen `array[j][i]`.

In code 11.7 staat op regel 5 de functie `build_array`. Deze functie declareert een pointer naar pointers **`int`** `**` en alloceert eerst een array met pointers, voor iedere rij één. Voor iedere rij wordt daarna een array met integers gecreëerd. Tenslotte worden alle elementen van de array gevuld met opeenvolgende getallen. Als er niet genoeg geheugenruimte is geeft `build_array` de *nullpointer* `NULL` terug. De functie `print_array` drukt deze getallen allemaal af. Bij de toekenning op regel 29 en bij het afdrukken op regel 45 wordt de normale notatie voor het arrayelement gebruikt, namelijk `array[j][i]`.

In het hoofdprogramma retourneert de functie `build_array` een **`int`** `**`. Deze pointer naar pointers wordt aan `array` toegekend. Als er door `build_array` geen array is gecreëerd, meldt het programma dat er onvoldoende geheugenruimte is en stopt het. Als er wel een array is gemaakt, drukt de functie `print_array` de array af.

```
/cc/array
/cc/array $ gcc -Wall -o array_dynamic array_dynamic.c

/cc/array $ array_dynamic 6 4
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
16 17 18 19
20 21 22 23

/cc/array $ array_dynamic 80000 6000 > output.txt

/cc/array $ array_dynamic 80000 8000
out of memory
```

Figuur 11.10 : De uitvoer van code 11.7 voor verschillende afmetingen.

Een aanroep van het programma met de waarde 6 en 4 toont op het scherm een array met 24 opeenvolgende getallen, zie figuur 11.10. Bij grote getallen wordt de uitkomst naar een bestand `output.txt` gestuurd. Voor de waarden 80000 en 6000 is er voldoende geheugen beschikbaar, maar voor 80000 en 8000 is er niet genoeg ruimte. De beschikbare geheugenruimte is systeemafhankelijk. In dit geval gaat het om een 32-bits Windows computer. De maximale geheugenruimte is dan ongeveer 2 GB, hetgeen overeenkomt met 500 miljoen integers.

De functie `print_array` drukt alle getallen kolomsgewijs af. Voor een goede uitlijning wordt het aantal cijfers gebruikt dat voor het grootste getal nodig is. Dat aantal cijfers wordt op regel 41 berekend met:

$$n_{\text{size}} = \lceil \log (n_{\text{row}} \, n_{\text{column}}) \rceil$$

De asterisk in de *format specifier* op regel 45 betekent dat het af te drukken getal een variabele grootte heeft. Deze grootte `size` staat dan eveneens in de parameterlijst voor het af te drukken getal.

Tweedimensionale array op basis van een eendimensionaal array

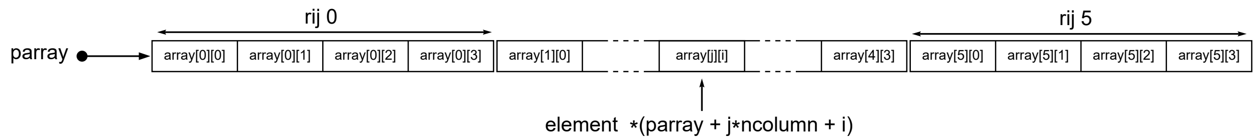
Een andere, veelgebruikte methode om een tweedimensionaal array dynamisch te declareren gebruikt één keer de functie `malloc`. Er wordt eigenlijk een eendimensionaal array gemaakt. Met pointerrekenen wordt het juiste element in de array gevonden.

Code 11.8 : Een tweedimensionaal array gebaseerd op een eendimensionaal array.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int *build_array(int nrow, int ncolumn)
6  {
7    int *parray;
8    int i, j;
9    int n;
10
11   parray = malloc(nrow * ncolumn * sizeof(int));
12   if ( parray == NULL ) {
13     return NULL;
14   }
15
16   n = 0;
17   for(j = 0; j < nrow; j++) {
18     for(i = 0; i < ncolumn; i++) {
19       *(parray + j*ncolumn + i) = n++;
20     }
21   }
22
23   return parray;
24 }
```

```
27 void print_array(int nrow, int ncolumn,
28                               int *parray)
29 {
30   int i, j;
31   int size = ceil(log10(nrow*ncolumn)) + 1;
32
33   for(j = 0; j < nrow; j++) {
34     for(i = 0; i < ncolumn; i++) {
35       printf("%*d", size,
36                   *(parray + j*ncolumn + i));
37     }
38     printf("\n");
39   }
40 }
41
42 int main(int argc, char *argv[])
43 {
44   int *parray;
45   int nrow     = atoi(argv[1]);
46   int ncolumn  = atoi(argv[2]);
47
48   parray = build_array(nrow, ncolumn);
49   if ( parray == NULL ) {
50     fprintf(stderr, "out of memory\n");
51     return 0;
52   }
53   print_array(nrow, ncolumn, parray);
54
55   return 0;
56 }
```

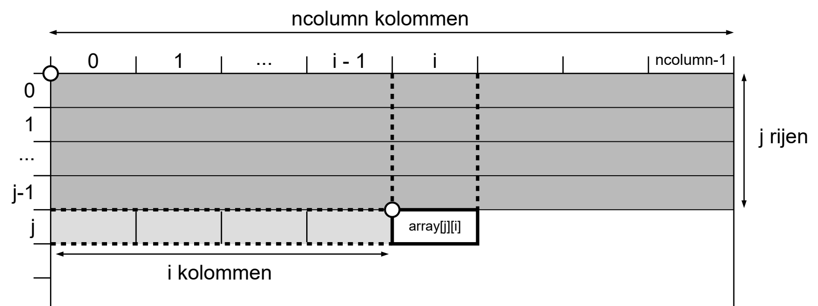
Het grote verschil met de methode uit code 11.7 is dat nu alle elementen achter elkaar staan in een eendimensionaal array, zoals dat in figuur 11.11 is getekend. De rijen uit figuur 11.9 staan nu achter elkaar.



Figuur 11.11 : De declaratie van een tweedimensionaal array met een eendimensionaal array. De rijen van het tweedimensionale array staan nu allemaal achter elkaar.

Code 11.7 beschrijft de array daadwerkelijk als een tweedimensionaal array. Code 11.8 is gebaseerd op pointers. De methoden van deze twee codes verschillen op drie aspecten:

- Code 11.7 benadert de array met een `int**`. Deze pointer naar pointers is `array` genoemd, omdat de elementen als een gewoon arrayelement worden benaderd `array[j][i]`. Code 11.8 benadert de array daarentegen met een `int*`. Deze pointer is `parray` genoemd, om aan te geven dat het een pointer naar de array is.
- De geheugenallocatie is in code 11.8 veel eenvoudiger. Er is maar één aanroep van `malloc` nodig.
- De benadering van een arrayelement is verschillend. Bij code 11.7 is dat identiek met de toewijzing bij een statisch array: `array[j][i]`. In code 11.8 wordt pointerrekenen gebruikt. Het element uit kolom `i` van rij `j` wordt gevonden door `j` rijen over te slaan `j*ncolumn` en daar `i` bij te tellen, zie ook figuur 11.12. Als `parray` naar het begin van de array wijst wordt de inhoud van het betreffende element gevonden met: `*(parray + j*ncolumn + i)`.



Figuur 11.12 : De pointerberekening van de positie van element `array[j][i]`. Voor het gezochte element staan `j` rijen met ieder `ncolumn` elementen, oftewel `j * ncolumn` elementen. In rij `j` staan voor het gezochte element `i` kolommen en dus `i` elementen. Totaal staan er voor `j * ncolumn + i` elementen.

De verschillen zijn niet enorm groot. Bij de eerste methode worden vaak fouten gemaakt bij de geheugenallocatie en de twee methode wordt vaak lastig gevonden vanwege het pointerrekenen.
De resultaten van beide methoden zijn identiek. De uitvoer bij code 11.8 is gelijk aan die van figuur 11.10. In beide gevallen wordt er geheugenruimte op de zogenoemde *heap* gereserveerd en is veel geheugen beschikbaar.

Tweedimensionale array met behulp van een VLA

In code 11.9 is in het hoofdprogramma op regel 39 een VLA, *variable length array*, toegepast om dynamisch een tweedimensionaal array te declareren. De functie `build_array` uit de vorige twee methoden is vervangen door een functie `fill_array`. Deze functie vult alleen de array met opeenvolgende waarden. De geheugenallocatie vindt automatisch plaats bij de declaratie van de VLA.

Code 11.9 : Een tweedimensionaal array met behulp van een VLA.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4
 5 void fill_array(int nrow, int ncolumn,
 6                      int array[nrow][ncolumn])
 7 {
 8   int i, j;
 9   int n;
10
11   n = 0;
12   for(j = 0; j < nrow; j++) {
13     for(i = 0; i < ncolumn; i++) {
14       array[j][i] = n++;
15     }
16   }
17 }
```

```
20 void print_array(int nrow, int ncolumn,
21                       int array[nrow][ncolumn])
22 {
23   int i, j;
24   int size = ceil(log10(nrow*ncolumn)) + 1;
25
26   for(j = 0; j < nrow; j++) {
27     for(i = 0; i < ncolumn; i++) {
28       printf("%*d", size, array[j][i]);
29     }
30     printf("\n");
31   }
32 }
33
34 int main(int argc, char *argv[])
35 {
36   int nrow    = atoi(argv[1]);
37   int ncolumn  = atoi(argv[2]);
38
39   int array[nrow][ncolumn];
40
41   fill_array(nrow, ncolumn, array);
42   print_array(nrow, ncolumn, array);
43
44   return 0;
45 }
```

Op het eerste gezicht lijkt deze oplossing veel eenvoudiger dan de methoden met de geheugenallocatie. Dit is echter niet het geval. Ten eerste moet bij de declaratie van de VLA op regel 39 de afmeting van de array bekend zijn. Ten tweede mag de declaratie niet globaal zijn. Ten derde vereisen de functieheaders speciale aandacht.

De variabelen `nrow` en `ncolumn` in de header van de functie `fill_array` en `print_array` zijn niet alleen nodig omdat deze variabelen in de functie gebruikt worden, maar ook nodig bij de declaratie van de array. Een VLA is nooit globaal. Als een functie een VLA gebruikt, moet deze dus altijd als parameter aan de functie worden meegegeven. Dit is in code 11.9 gedaan door de VLA voluit in de functieheader op te nemen:

```
void fill_array(int nrow, int ncolumn, int array[nrow][ncolumn])
```

De variabelen `nrow` en `ncolumn` moeten dan ook in de functieheader staan. Zonder indices bij de arraydeclaratie

```
void fill_array(int nrow, int ncolumn, int array[][])
```

geeft de compiler deze foutmelding:

```
error: array type has incomplete element type
```

Met andere indexnamen:

```
void fill_array(int nrow, int ncolumn, int array[rij][kolom])
```

meldt de compiler deze fouten:

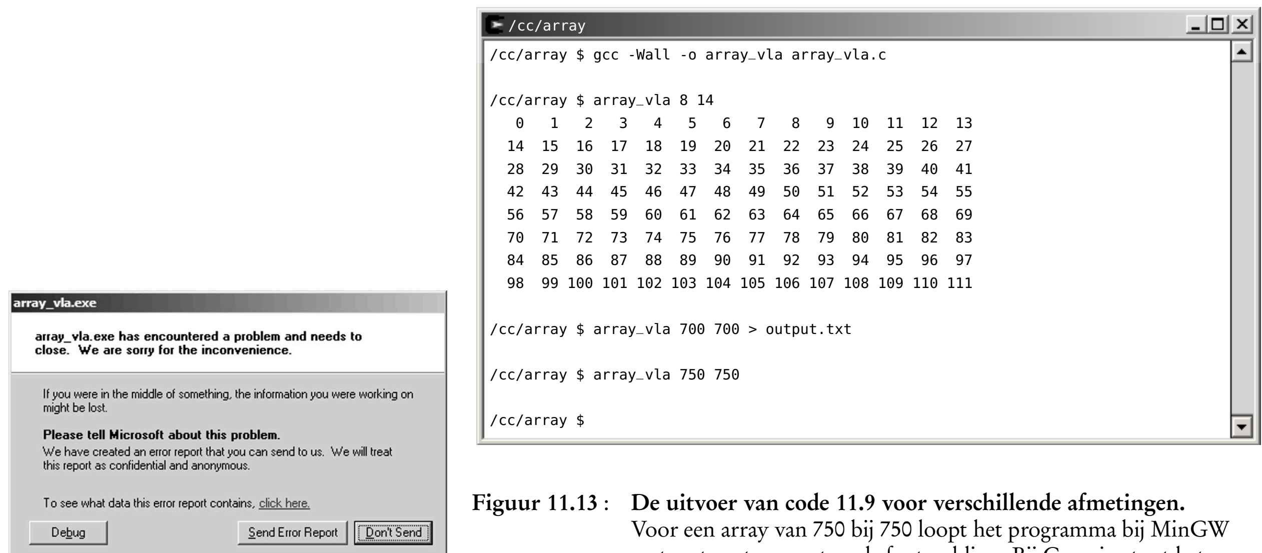
```
error: 'rij' undeclared here (not in a function)
error: 'kolom' undeclared here (not in a function)
```

Wel mag `nrow` worden weggelaten in de functieheader bij de declaratie van `array`.

```
void fill_array(int nrow, int ncolumn, int array[][ncolumn])
```

Net zoals in code 11.8 bij de berekening van de positie van een element in de array de breedte `ncolumn` nodig is, heeft een VLA deze breedte ook nodig.

De plaats van de declaratie van de VLA is erg belangrijk en de waarden van de arrayparameters `nrow` en `ncolumn` moeten bekend zijn op het moment dat de declaratie wordt uitgevoerd. Zolang de declaratie niet bekend is, zijn functieaanroepen met de VLA nog niet mogelijk. Een praktische oplossing is om, zoals dat in code 11.9 is gedaan, de declaratie in de `main` te plaatsen en daarna de VLA met functies te manipuleren.

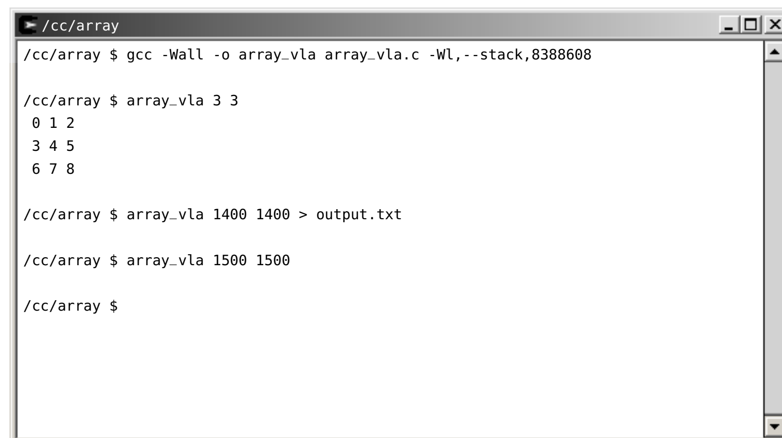


Figuur 11.13 : De uitvoer van code 11.9 voor verschillende afmetingen. Voor een array van 750 bij 750 loopt het programma bij MinGW vast en toont nevenstaande foutmelding. Bij Cygwin stopt het programma zonder uitvoer en zonder foutmelding.

Het verschil met de andere twee methoden is dat er minder plaats is. Standaard is op een 32-bits machine bij Cygwin de stack 2MB groot. Er is dus plaats voor maximaal 50000 integers. Lastig is dat er geen eenvoudige mogelijkheid bestaat om te testen of er nog genoeg geheugen op de stack is voor de array. Het is dus niet mogelijk om het programma robuust te maken voor geheugenproblemen.

Omdat een VLA altijd op de stack wordt geplaatst, treden er bij grote arrays veel eerder geheugenproblemen op dan bij de twee andere methoden. De functie `malloc` alloceert altijd ruimte op de *heap*. Geheugenallocatie op de *stack* is sneller, maar er is veel minder geheugen beschikbaar. In figuur 11.13 wordt het programma uit code 11.9 eerst uitgevoerd voor een array van 8 bij 14. Voor een array van 700 bij 700 integers is er ook nog voldoende ruimte op de stack. Het resultaat wordt met `> output.txt` naar een bestand `output.txt` omgeleid. Bij een array van 750 bij 750 crasht het programma. Er is onvoldoende geheugenruimte op de stack.

Overigens kan de stack wel worden vergroot met een compiler-optie. De stack is standaard bij Cygwin en MinGW op een 32-bits Windows machine 2097152 bytes. In figuur 11.14 is de stack vier keer zo groot gemaakt door bij de compilatie de optie `-Wl,--stack,8388608` toe te voegen. Een array van 1400 bij 1400 integers past nu wel, maar er is te weinig ruimte voor een array van 1500 bij 1500.



Figuur 11.14 : De uitvoer van code 11.9 met een grotere stack. Het programma werkt wel bij een array van 1400 bij 1400, maar loopt vast bij een array van 1500 bij 1500.

Samenvatting declaratie tweedimensionale arrays

Gebruik bij de dynamische declaratie van grote arrays altijd een methode die `malloc` gebruikt. De verschillen tussen de methode van code 11.7 en code 11.8 zijn niet groot. De keuze tussen twee methoden zal van de stijl afhangen die de programmeur prefereert.

De methode met VLA is alleen praktisch voor tijdelijke allocatie van kleine arrays, zoals dat in code 11.4 is gedaan. De VLA is niet geschikt voor grotere arrays.

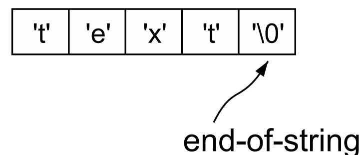
12

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: ▪ Het *end-of-string*-teken. ▪ Het verschil tussen een **`char`** en een string. ▪ De declaraties van strings. ▪ Het toekennen aan en het overnemen van strings. ▪ De stringfuncties uit `string.h`. ▪ Arrays van strings. ▪ De argumentenlijst bij `main`. Voorbeeldprogramma's voor strings tonen: ▪ Het omzetten van jaar, maand en dag in een datumcode. ▪ 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 10 en 11 zijn behandeld.

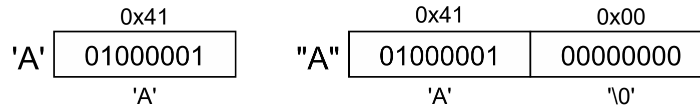
Figuur 12.1 visualiseert de string `"text"`. De string `"text"` bestaat uit de vier karakters van het woord *text* en het *end-of-string*-teken. Om het woord *text* te bewaren is dus een array van minimaal vijf **`char`**'s nodig.



Figuur 12.1 : 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 12.2 is het geheugengebruik van `'A'` en `"A"` getekend.



Figuur 12.2 : 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 12.1 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.

12.1 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";
```

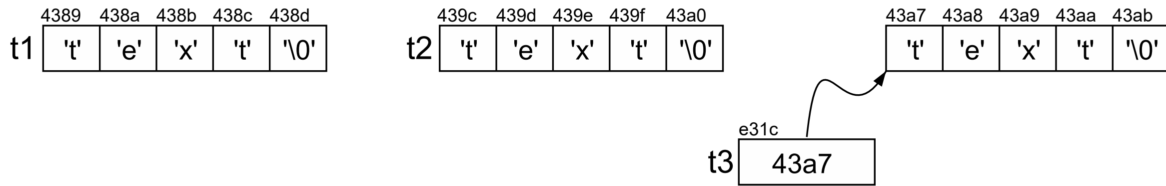
Bij de declaratie van `t3` reserveert de compiler geheugenruimte voor de pointer `t3` en ruimte voor de string `"text"`. Het beginadres van de string wordt aan de pointer `t3` toegekend. Figuur 12.3 visualiseert het geheugengebruik bij de declaratie van `t1`, `t2` en `t3`.

Code 12.1 : 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
```

```
11 int getmonth(char *m)
12 {
13   if (strcmp(m, "January") == 0) {
14     return 1;
15   } else if (strcmp(m, "February") == 0) {
16     return 2;
17   } else if (strcmp(m, "March") == 0) {
18     return 3;
19   } else if (strcmp(m, "April") == 0) {
20     return 4;
21   } else if (strcmp(m, "May") == 0) {
22     return 5;
23   } else if (strcmp(m, "June") == 0) {
24     return 6;
25   } else if (strcmp(m, "July") == 0) {
26     return 7;
27   } else if (strcmp(m, "August") == 0) {
28     return 8;
29   } else if (strcmp(m, "September") == 0) {
30     return 9;
31   } else if (strcmp(m, "October") == 0) {
32     return 10;
33   } else if (strcmp(m, "November") == 0) {
34     return 11;
35   } else if (strcmp(m, "December") == 0) {
36     return 12;
37   } else {
38     return 0;
39   }
40 }
41
```

```
43 int main (int argc, char **argv)
44 {
45   char exename[32];
46   char year[5], month[10], day[3];
47   int  y,m,d;
48
49   strcpy(exename, *argv++);
50   if ( argc < 4 ) {
51     printf(MSG_USAGE, exename);
52     return 1;
53   }
54
55   strcpy(year, *argv++);
56   strcpy(month, *argv++);
57   strcpy(day, *argv++);
58
59   y = atoi(year);
60   if ( (y < 2000) || (y > 2099) ) {
61     printf(MSG_USAGE, exename);
62     printf(MSG_YEAR);
63     return 1;
64   }
65   y -= 2000;
66
67   d = atoi(day);
68   if ( (d < 1) || (d > 31) ) {
69     printf(MSG_USAGE, exename);
70     printf(MSG_DAY);
71     return 1;
72   }
73
74   if ( ( m = getmonth(month)) == 0) {
75     printf(MSG_USAGE, exename);
76     printf(MSG_MONTH);
77     return 1;
78   }
79
80   printf("%s %s %s is %02d%02d%02d",
81               day, month, year, y, m, d);
82
83   return 0;
84 }
```



Figuur 12.3 : 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.

12.2 Op veilige wijze strings gebruiken

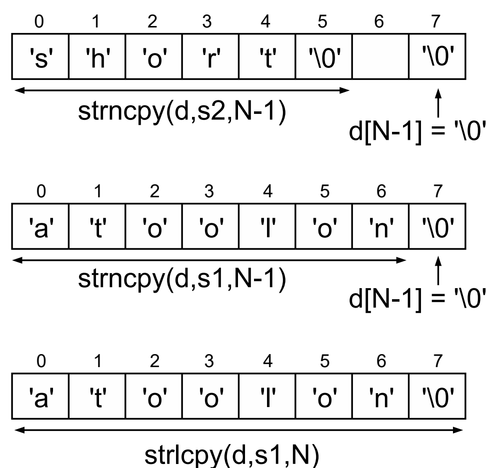
De functie `strcpy` kan, net als bijvoorbeeld de functie `strcat`, op ongewenste plaatsen in het geheugen schrijven.
Deels kan dit worden voorkomen door in plaats van `strcpy` de functie `strncpy` of `strlcpy` te gebruiken. In code 12.2 heeft de string `d` plaats voor acht karakters inclusief de *end-of-string*. De functie `strncpy` kopieert maximaal N-1 karakters van

`src` naar `des`. Figuur 12.4 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 12.2 : 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-1] = '\0';
14   printf("%s\n", d);
15
16   strncpy(d, s2, N-1);
17   d[N-1] = '\0';
18   printf("%s\n", d);
19
20   strlcpy(d, s1, N);
21   printf("%s\n", d);
22   return 0;
23 }
```



Figuur 12.4 : 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.

12.3 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 12.1 geeft een overzicht van de meest gebruikte stringfuncties. Het headerbestand voor deze functies is `string.h`.

Tabel 12.1 : 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 154 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, maar nu met `NULL` als eerste parameter. De tekens 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 12.5 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.

```
/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 12.5 : Een deel van de *man page* van de stringfunctie strncpy. Deze handleiding wordt getoond als achter de prompt de opdracht `man strncpy` **te geven.**

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.

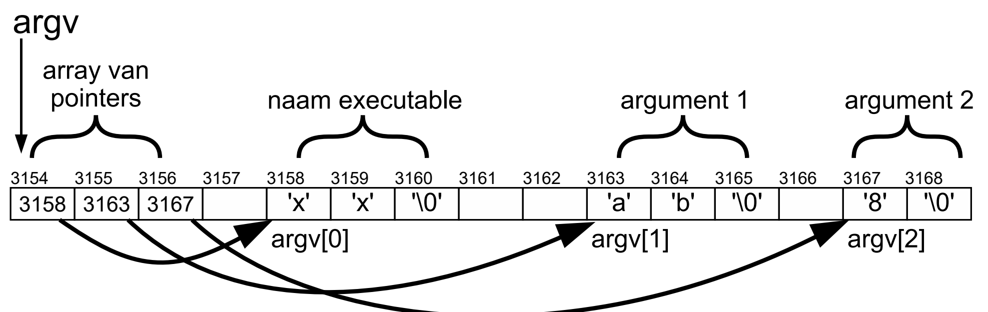
12.4 Array van strings

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 5.8 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 12.6 : 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 12.1 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 12.6 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 12.1 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]);
```

13

Advanced C

Doelstelling

Dit hoofdstuk behandelt een aantal geavanceerde onderwerpen, die voor het programmeren van microcontrollers soms 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, wat datastructuren zijn en wat een variabele argumentenlijst is.

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.
- Functies met een variabele argumentenlijst.

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` en `qsort`.
- Het afdrukken van een array van gegevens met een bepaalde datastructuur.
- Het gebruik van een lijst met records.
- Een functie met een variabele argumenten lijst.

13.1 Lezen en schrijven naar bestanden

Het lezen uit en het schrijven naar bestanden is bij microcontrollers minder belangrijk. Wel komen veel aspecten bij het schrijven naar en lezen van EEPROM, flash en allerlei interfaces overeen met het manipuleren van bestanden. Een verhaal over standaard C zonder bestandswerkingen is ook 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 voormalige 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 13.1 : **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 13.1 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 13.1 staat, is het resultaat de uitvoer van figuur 13.2.

```
/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 13.2 : **Een deel van de uitvoer van code 13.1.**

Uitleg code 13.1 regel 7
`FILE`

De datastructuur `FILE` is nodig bij bestandsbewerkingen. Deze datastructuur bevat onder andere een pointer naar de buffer, de grootte van de buffer en pointers naar twee functies voor het schrijven en het lezen van een karakter.

Code 13.1 : 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 modus 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 modus kan een `b` of `t` en een `+` worden toegevoegd. De modus `"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 oude 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 Windows-systeem twee karakters (<CR><LF>) en op een Unix-systeem maar één 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 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 13.1.

Tabel 13.1 : De lees- en schrijffuncties.

leesfuncties	
`fscanf()`	leest tekst met een gegeven formaat
`fgets()`	leest een regel (tot en met de end-of-line)
`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 13.2 geeft een overzicht van de kenmerken van de vier leesfuncties.

Tabel 13.2 : 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. • Spaties in velden zijn lastig. • Tab's en *end-of-lines* worden als white space gelezen. • Uit een gewone tekst verdwijnt de regelopmaak. • 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*. • Er is een buffer nodig voor de regels. • De grootte van de buffer is vooraf moeilijk te bepalen.	• Alle karakters worden een voor een gelezen. • Dat is niet erg efficiënt. • Parsers gebruiken vaak `getc`, omdat het checken van de syntax van de data hiermee eenvoudig is.	• Leest grote datablokken in een keer binnen. • Probleem is dat de grootte van de datablokken vaak moeilijk te bepalen is. • Een oplossing is alle data (de hele file) in een keer te lezen. • 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 13.2 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 13.2 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 13.3 toont het resultaat van het programma van code 13.2.

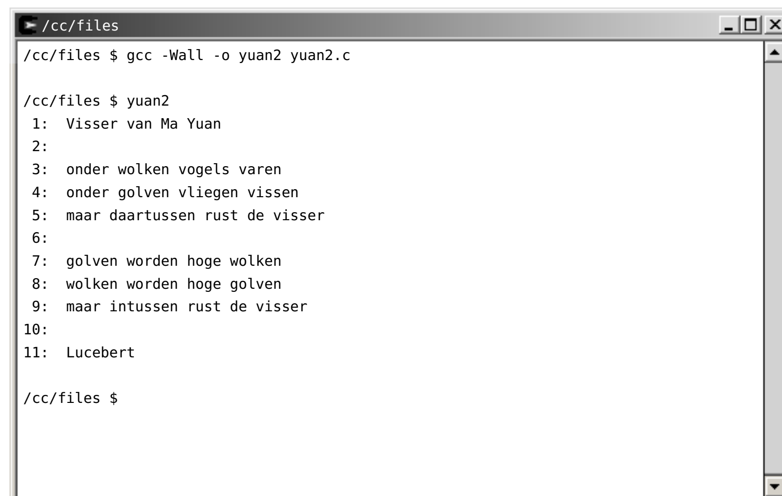
Uitleg code 13.2 regel 18
`fgets()`

```
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 is,

Code 13.2 : Het lezen uit een bestand met fgets

```
#include <stdio.h>

#define  MAXBUF 256

char buf[MAXBUF];

int main(void)
{
  FILE * fp;
  int i;

  if ( (fp = fopen("ma_yuan.txt", "r")) == NULL ) {
    printf("error: couldn't find or open file");
    return 1;
  }

  i=1;
  while ( fgets(buf, MAXBUF-1, fp) != NULL ) {
    printf("%2d: %s", i++, buf);
  }
  fclose(fp);

  return 0;
}
```



Figuur 13.3 : De uitvoer van code 13.2.

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 het 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 '\0', 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 13.3 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 13.3 staat het resultaat.

Uitleg code 13.3 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 13.3 : 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 }
```

Er is ook een functie `feof`:
`int feof(FILE *f);`
Deze functie geeft als het einde van het bestand is gevonden een waarde ongelijk aan `0` terug en anders `0`.

Een andere interessante functie is `ungetc`:
`int ungetc(char c, FILE *f);`
Deze functie zet het karakter `c` terug in de datastroom. Hiermee is het mogelijk om te spieken: `fgetc` leest het karakter, `ungetc` zet het terug, waardoor het volgende karakter uit de datastroom al bekend is.

`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 13.4 bepaalt eerst de grootte van 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.

```
/cc/files

/cc/files $ gcc -Wall -o yuan3 yuan3.c

/cc/files $ yuan3
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

/cc/files $
```

Figuur 13.4 : De uitvoer van code 13.3.

Nadat de grootte van het bestand bekend is, wordt met `malloc` geheugenruimte gealloceerd en wordt met `fread` het hele bestand naar deze geheugenplek gekopieerd.

```
/cc/files

/cc/files $ gcc -Wall -o yuan4 yuan4.c

/cc/files $ yuan4
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

/cc/files $
```

Figuur 13.5 : De uitvoer van code 13.4.

Uitleg code 13.4 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 13.4 : 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()`

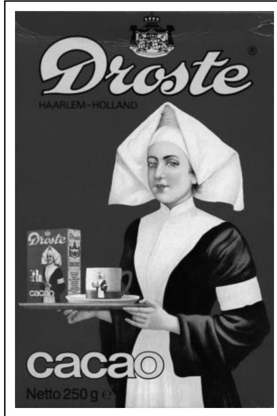
`long ftell(FILE *f);`
De functie `ftell` geeft de huidige positie van de filepointer in het bestand terug, gerekend vanaf het begin.

Regel 17
`rewind()`

`void rewind(FILE *f);`
De functie `rewind` zet de filepointer terug naar het begin van het bestand.

13.2 Recursie

Een recursieve functie is een functie die zichzelf aanroept. Deze functies geven vaak elegante oplossingen voor ingewikkelde problemen.



Figuur 13.6 : 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 $\cdots$

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:

$$n! \;=\; \begin{cases} 1 & \text{als} \quad n \;=\; 0 \\ n*(n-1)! & \text{als} \quad n \;>\; 0 \end{cases} \tag{13.1}$$

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 13.5 gebruikt een recursieve functie `fac` om 17! te berekenen. Deze functie is gebaseerd op de vergelijking 13.1. Omdat de faculteit van een getal snel heel groot wordt, is de retourwaarde van `fac` van het type **`unsigned long long`**.

Sommige acroniemen (letterwoorden) zijn recursief. Voorbeelden zijn GNU (*GNU's Not Unix*) en PHP (*PHP: Hypertext Preprocessor*).

Gebruik bij MinGW in code 13.5 in plaats van `%llu` de format specifier `%I64u`.

Code 13.5 : 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:

$$\text{fib}(n) \;=\; \begin{cases} 1 & \text{als}\, n \;=\; 0 \vee n \;=\; 1 \\ \text{fib}(n-1)+\text{fib}(n-2) & \text{als}\, n \;>\; 0 \end{cases} \tag{13.2}$$

Code 13.5 gebruikt een recursieve functie `fib` om de eerste twintig getallen uit de reeks van Fibonacci te berekenen. Deze functie is gebaseerd op vergelijking 13.2.
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 13.7 en in code 13.8 staan de iteratieve functies voor de faculteit en voor de getallen van Fibonacci.

Code 13.6 : 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 13.7 : 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 }
```

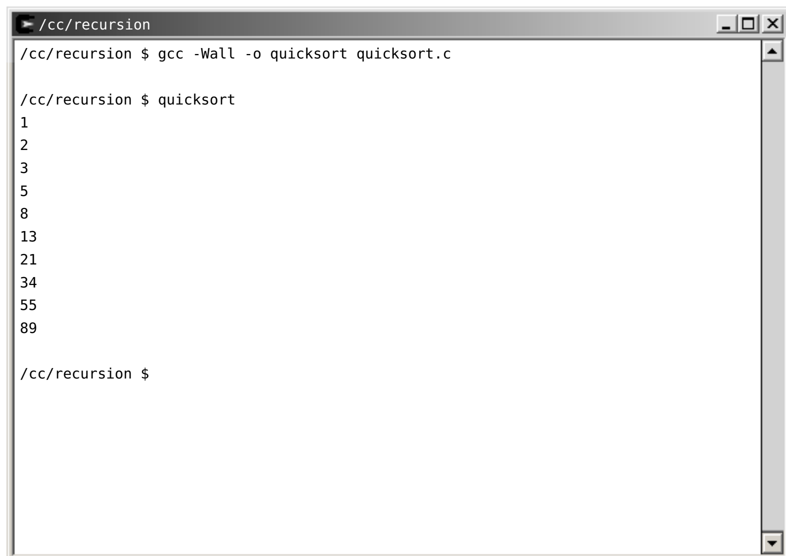
Code 13.8 : Een iteratieve functie `fib`.

```
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 13.6, 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 13.8 is vele malen sneller dan de recursieve functie `fib` uit figuur 13.6. 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.

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 ontwikkeld 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 13.9 sorteert met behulp van de functie `quicksort` een array van integers en drukt de gesorteerde rij af. Het resultaat staat in figuur 13.7.



Figuur 13.7 : Het resultaat van de quicksort van code 13.9.

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 13.9 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.

Code 13.9 : 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 left, int right)
{
  int i,j;
  int v;

  if (right > left) {
    v = a[right];
    i = left-1;
    j = right;
    while (1)
    {
      while (a[++i] < v);
      while (a[--j] > v);
      if (i >= j) break;
      swap(a,i,j);
    }
    swap(a,i,right);
    quicksort(a, left, i-1);
    quicksort(a, i+1, right);
  }
}

int main(void)
{
  int i;
  int a[]={89,2,1,13,21,8,55,5,3,34};

  quicksort(a, 0, 9);

  for(i=0; i<=9; i++) {
    printf("%d\n", a[i]);
  }

  return 0;
}
```

De standaardbibliotheek bevat een functie `qsort` die breder inzetbaar is dan de `quicksort` uit code 13.9. 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 13.10 : 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 13.10 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 13.8.

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 13.8 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-

```
/cc/recursion
/cc/recursion $ gcc -Wall -o starsort starsort.c
starsort.c:  In function ‘main’:
starsort.c:22:  warning:  passing arg 4 of ‘qsort’ from incompatible
pointer type

/cc/recursion $ starsort
Aretha Franklin
Blondie
Bob Dylan
Bruce Springsteen
Carly Simon
Dave Davies
David Bowie
Diana Ross
Elton John
Elvis Presley
Eric Clapton
Grace Slick
Janis Joplin
Jeff Beck
Jim Morrison
Jimmy Page
Joan Baez
John Lennon
Joni Mitchell
Keith Richards
Lou Reed
Mick Jagger
Neil Young
Paul McCartney
Peter Townshend
Robert Plant
Roy Orbison
Tina Turner

/cc/recursion $
```

Figuur 13.8 : Code 13.10 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));
}
```

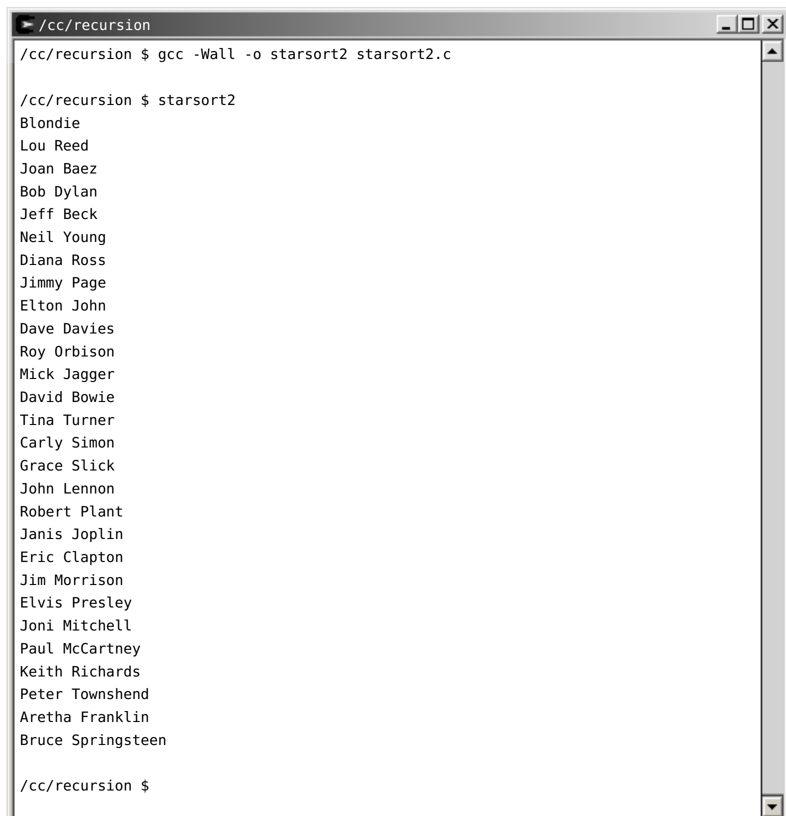
Een groot voordeel is dat er zo veel meer testvarianten gemaakt kunnen worden. Door `p2` en `p1` te verwisselen worden de popsterren in omgekeerde volgorde afgedrukt.

```
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 teruggeven: één waarde kleiner dan 0, één waarde groter dan 0 of de waarde 0. Voor het geval de positie van de eerste parameter voor, na of identiek is met die van de tweede parameter. 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 13.10 wordt gebruikt in plaats van strcmp worden de namen van de sterren gesorteerd op grootte. Figuur 13.9 laat dit zien en toont bovendien aan dat de compiler nu geen waarschuwing geeft.



Figuur 13.9 : De testfunctie starcmp_size sorteert de namen op grootte.

13.3 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. 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)
```

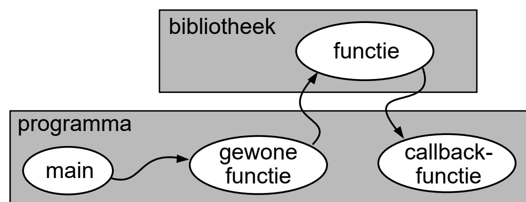
In tabel 13.3 staan een aantal voorbeelden van pointers naar functies. Pointer `q` kan als volgt gebruikt worden:

```
q = square;
printf("%d %d\n", square(13), q(13));  // print 169 169
```

De uitdrukkingen `square(13)` en `q(13)` zijn identiek.

Callbackfunctie

Pointers naar functie zijn gerelateerd met het principe van een zogenoemde callbackfunctie. Dat is de functie, die een functie in een bibliotheek nodig heeft, om haar taak te kunnen uitvoeren.



Figuur 13.10 : De werking van de callbackfunctie. De functie in de bibliotheek heeft de callbackfunctie nodig om de juiste actie te kunnen doen.

In figuur 13.10 roept de `main` een gewone functie aan, die een bibliotheekfunctie aanroept en die op haar beurt weer de callbackfunctie in het hoofdprogramma aanroept. Bij het sorteervoorbeeld uit de vorige paragraaf is de functie `qsort` de bibliotheekfunctie en zijn `starcmp`, `starcmp_reverse` en `starcmp_size` mogelijke callbackfuncties.

Tabel 13.3 : 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.
`(int **) (*y) (int, int)`	Dit is een pointer `y` naar een functie met twee keer een `int` als argument en met een pointer naar een pointer van `int`'s als retourwaarde.

13.4 Samengestelde datatypes

In paragraaf 3.3 zijn de samengestelde datatypen array en string geïntroduceerd. Deze datatypen zijn opgebouwd uit elementen van dezelfde soort. C kent ook twee samengestelde datatypen die uit verschillende datatypen kunnen bestaan: de **`struct`** en de **`union`**.

Het samengestelde datatype `struct`

In C wordt een structuur 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_t;
```

Tegenwoordig is het een goede gewoonte om de naam van de typedefinitie te laten eindigen op `_t`, zodat zichtbaar is dat het een typenaam is.

Hiervoor is een type `naw_t` gedefinieerd die uit de structuur `naw` bestaat. De declaratie van een variabele `x` luidt in dit geval:

```
naw_t x;
```

Er staat geen **`struct`** voor `naw_t` bij de declaratie van `x`.

Code 13.11 bevat een typedefinitie `stud_t` 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_t` en is gevuld met de gegevens van drie studenten. Het programma drukt de gegevens van deze studenten af.

Code 13.11 : 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_t))
5
6  typedef struct stud {
7    char   id[MAX_SHORT];
8    char   name[MAX_NAME];
9    char   mobile[MAX_SHORT];
10 } stud_t;
11
12 stud_t 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 }
```

Het samengestelde datatype `union`

Het samengestelde datatype **`union`** lijkt sterk op een **`struct`**. Het verschil is dat bij een **`struct`** voor ieder veld geheugen wordt gereserveerd. Bij een **`union`** staan alle velden op dezelfde geheugenplaats. De grootte van een **`struct`** is de som van de afmetingen van alle velden. De grootte van een **`union`** is gelijk aan het grootste veld. Een voorbeeld van een **`union`** is:

```
union integer {
  uint8_t   u8;
  uint16_t  u16;
  uint32_t  u32;
} var;
```

De variabele `var` is 32 bits groot, omdat `uint32_t` het grootste veld in de **`union`** is. De toekenning:

```
var.u32 = 0x12345678L;
```

definieert tegelijkertijd de waarde van `var.u16` en `var.u8`. Twee afdrukopdrachten laten dat direct zien:

De **`union`** wordt weinig toegepast. Serieuze toepassingen met een **`union`** zijn spaarzaam.

```
printf("%#x", var.u16);   // prints 0x1234
printf("%#x", var.u8);    // prints 0x12
```

Met een **`union`** kunnen kleiner datatypes uit een groter datatype worden gehaald. De **`union`** wordt ook gebruikt om de volgorde van bytes in een integer te wijzigen.

Samengestelde datatypes, (type-)definities en enumeraties bij de Xmega

Een microcontroller is opgebouwd uit een processor met naast de gewone in- en uitgangen een groot aantal extra faciliteiten, zoals timers en analoge en digitale interfaces. Om deze faciliteiten te gebruiken zijn registers nodig waarin instellingen en gegevens kunnen worden opgeslagen en uitgehaald.

De voorbeelden van de Xmega zijn ontleend aan het headerbestand `iox256a3u.h` van de Xmega256a3u.

Code 13.12 : Voorbeeld van de toepassing van `struct` bij de Xmega.

```
1922  /* I/O Ports */
1923  typedef struct PORT_struct
1924  {
1925      register8_t DIR;           /* I/O Port Data Direction */
1926      register8_t DIRSET;        /* I/O Port Data Direction Set */
1927      register8_t DIRCLR;        /* I/O Port Data Direction Clear */
1928      register8_t DIRTGL;        /* I/O Port Data Direction Toggle */
1929      register8_t OUT;           /* I/O Port Output */
1930      register8_t OUTSET;        /* I/O Port Output Set */
1931      register8_t OUTCLR;        /* I/O Port Output Clear */
1932      register8_t OUTTGL;        /* I/O Port Output Toggle */
1933      register8_t IN;            /* I/O port Input */
1934      register8_t INTCTRL;       /* Interrupt Control Register */
1935      register8_t INT0MASK;      /* Port Interrupt 0 Mask */
1936      register8_t INT1MASK;      /* Port Interrupt 1 Mask */
1937      register8_t INTFLAGS;      /* Interrupt Flag Register */
1938      register8_t reserved_0x0D;
1939      register8_t REMAP;         /* I/O Port Pin Remap Register */
1940      register8_t reserved_0x0F;
1941      register8_t PIN0CTRL;      /* Pin 0 Control Register */
1942      register8_t PIN1CTRL;      /* Pin 1 Control Register */
1943      register8_t PIN2CTRL;      /* Pin 2 Control Register */
1944      register8_t PIN3CTRL;      /* Pin 3 Control Register */
1945      register8_t PIN4CTRL;      /* Pin 4 Control Register */
1946      register8_t PIN5CTRL;      /* Pin 5 Control Register */
1947      register8_t PIN6CTRL;      /* Pin 6 Control Register */
1948      register8_t PIN7CTRL;      /* Pin 7 Control Register */
1949  } PORT_t;
```

Deze registers zijn geheugenlocaties met een bepaald adres. Voor de vele honderden registers, die de Xmega van Atmel kent, worden structuren, macro's en enumeraties gebruikt om deze te kunnen benaderen. Voor iedere Xmega bestaat een headerbestand waarin alle registers beschreven staan. In code 13.12 staat als

voorbeeld de **struct** uit het headerbestand, die de in- en uitgangsregisters van een poort beschrijft. Bij een microcontroller zijn de IO-pinnen gegroepeerd in groepen van acht. Een dergelijke groep wordt een poort genoemd.

Code 13.13 : De typedefinitie van `register8_t` bij de Xmega.

```
97   #include <stdint.h>
98
99   typedef volatile uint8_t register8_t;
100  typedef volatile uint16_t register16_t;
101  typedef volatile uint32_t register32_t;
```

In code 13.13 staat de typedefinitie van het type `register8_t` dat bij code 13.12 toegepast is om de registers te definiëren.
Code 13.14 past een enumeratie toe om de verschillende niveaus van interrupt 1 te vast te leggen.

Code 13.14 : Voorbeeld van de toepassing van een enumeratie bij de Xmega.

```
1959  /* Port Interrupt 1 Level */
1960  typedef enum PORT_INT1LVL_enum
1961  {
1962      PORT_INT1LVL_OFF_gc = (0x00<<2),  /* Interrupt Disabled */
1963      PORT_INT1LVL_LO_gc  = (0x01<<2),  /* Low Level */
1964      PORT_INT1LVL_MED_gc = (0x02<<2),  /* Medium Level */
1965      PORT_INT1LVL_HI_gc  = (0x03<<2),  /* High Level */
1966  } PORT_INT1LVL_t;
```

De zogenoemde interruptvectoren zijn bij de Xmega met gewone definities vastgelegd. Code 13.15 toont de definities van de interruptvectoren van poort A.

Code 13.15 : Voorbeeld van de toepassing van een enumeratie bij de Xmega.

```
7310  /* PORTA interrupt vectors */
7311  #define PORTA_INT0_vect_num  66
7312  #define PORTA_INT0_vect      _VECTOR(66)  /* External Interrupt 0 */
7313  #define PORTA_INT1_vect_num  67
7314  #define PORTA_INT1_vect      _VECTOR(67)  /* External Interrupt 1 */
```

De **union** wordt bij de Xmega gebruikt om met 16- en 32-bits getallen te kunnen werken. Het onderstaande vereenvoudigde fragment uit het headerbestand laat dat zien:

```
union {
  register16_t regname;
  struct {
    register8_t regnameL ;
    register8_t regnameH ;
  };
}
```

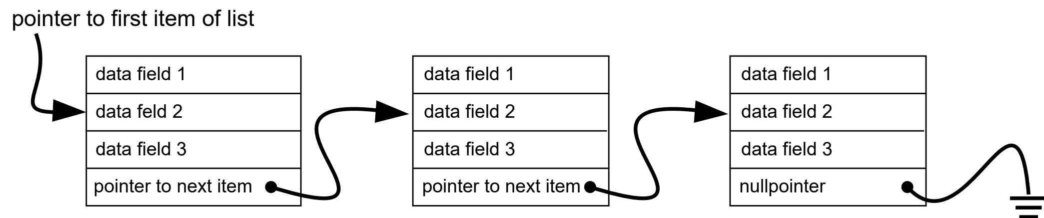
De voorbeelden is deze subparagraaf laat zien dat typedefinities, definities, structuren, enumeraties veel gebruikt worden bij het maken van bibliotheken. Een microcontrollerprogammeur moet de velden in een **struct** kunnen lezen en toewijzen en de definities op een correcte manier kunnen toepassen.

13.5 Datastructuren

Een **struct** wordt vaak gebruikt om een verzameling van dezelfde soort gegevens vast te leggen. In code 13.11 is dit een array van **struct**'s. 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 array kan de functionaliteit van het programma beperken. Daarom worden vaak dynamische datastructuren toegepast.

Lijsten

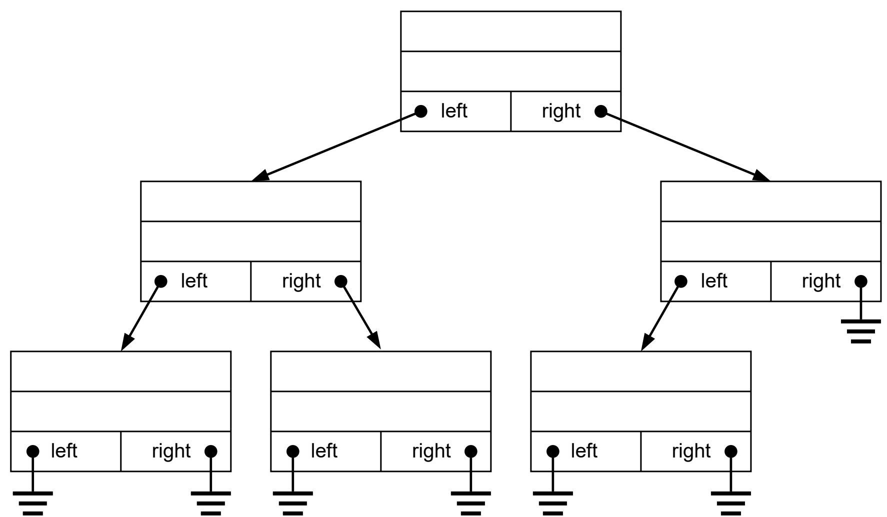
Een voorbeeld van een dynamische datastructuur is de lijst. In figuur 13.11 staat een voorbeeld. Het basiselement is een **struct** met een pointer, die naar het volgende element wijst. Alle elementen van de lijst zijn via deze pointers bereikbaar. Een lijst kan relatief eenvoudig worden uitgebreid of verkleind door er vooraan, achteraan of op een willekeurig plaats een element toe te voegen of te verwijderen.



Figuur 13.11 : Een voorbeeld van een lijststructuur.

Bomen

Een ander voorbeeld van een dynamische datastructuur is de boom. Figuur 13.12 geeft een voorbeeld. Het basiselement is een **struct** met twee pointers, die naar twee andere elementen uit de boom wijzen. Een boomstructuur is zeer geschikt om gegevens te ordenen.



Figuur 13.12 : Een voorbeeld van een boomstructuur.

Code 13.16 : Afdrukken lijst met studenten: main.c.

```
#include <string.h>
#include "student.h"

int main(void)
{
  stud_t *studentlist=NULL;
  stud_t *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 13.17 : 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_t;

stud_t *newStud(char *id, char *name);
stud_t *appendStud(stud_t *slist, stud_t *news);
void  printStuds(stud_t *slist);
void  freeStuds(stud_tD *slist);
```

Code 13.18 : Afdrukken lijst met studenten: student.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "student.h"

stud_t *newStud(char *id, char *name)
{
   stud_tD *s;

   s = (stud_t *) malloc(sizeof(stud_t));
   strcpy(s->id, id);
   strcpy(s->name, name);
   strcpy(s->mobile, "");
   s->next  = NULL;

   return s;
}

stud_t *appendStud(stud_t *slist,
                   stud_t *news)
{
  stud_t *s = slist;

  if ( s == NULL ) {
    return news;
  }

  while ( s->next != NULL ) {
    s = s->next;
  }
  s->next = news;

  return slist;
}

void printStuds(stud_t *slist)
{
  while ( slist != NULL ) {
    printf("%-7s %-22s %s\n", slist->id,
            slist->name, slist->mobile);
    slist = slist->next;
  }
}

void freeStuds(stud_t *slist)
{
  stud_t *s;

  while ( slist != NULL ) {
    s = slist;
    slist = slist->next;
    free(s);
  }
}
```

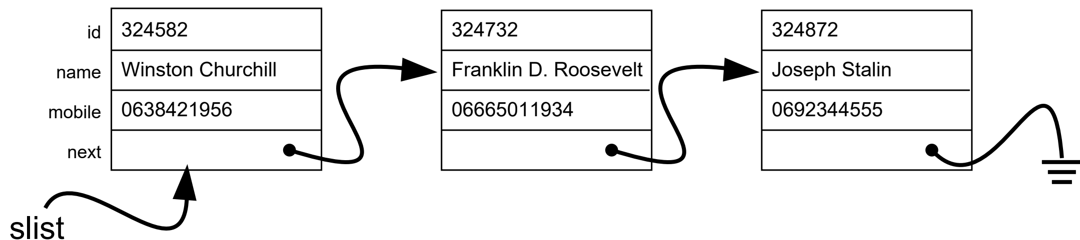
Voorbeeld met een lijststructuur

Een record is een set met gegevens. Een serie records vormen een lijst. Een record is dus een object van de lijst.

In code 13.16, 13.17 en 13.18 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 13.17 regel 4-9

In het headerbestand `header.h` wordt het type `stud_t` gedefinieerd, deze is afgeleid van `struct stud`. Dit type komt overeen met het type `struct stud` uit code 13.11, 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 13.13 is getekend.



Figuur 13.13 : **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`.

Uitleg code 13.17 regel 11-14

Het headerbestand `header.h` bevat ook de prototypen van de functies voor de bewerkingen van de studentenlijst.

Uitleg code 13.18 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`.

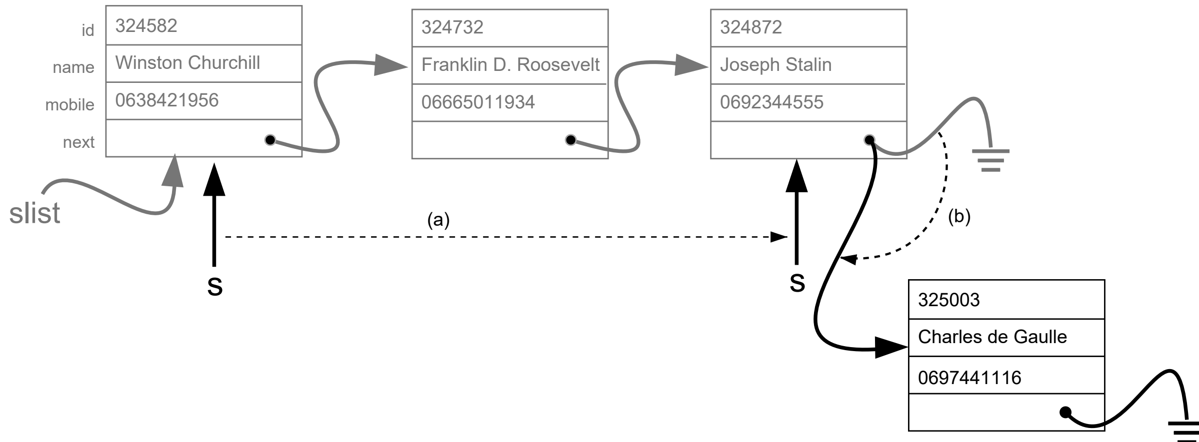
Uitleg code 13.18 regel 14
`->`
`.`

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 13.18 regel 20-34

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 13.14 is dit grafisch weergeven. 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.



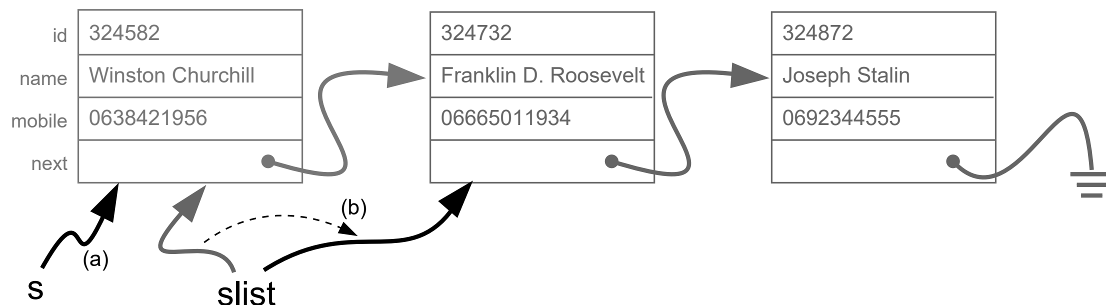
Figuur 13.14 : Het achteraan toevoegen van een record aan de lijst. Eerst (a) wordt met een hulppointer `s` het laatste record van de lijst gezocht. Vervolgens (b) wordt er voor gezorgd dat de pointer `next` van het laatste element naar het nieuwe record gaat wijzen.

Uitleg code 13.18 regel 20-34

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 gedefinieerd. De pointer `studentlist` in `main.c` onthoudt waar het begin van de lijst is.

Uitleg code 13.18 regel 45-53

De functie `freeStuds` geeft de geheugenruimte die voor de studentenlijst gebruikt is weer vrij. Figuur 13.15 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 13.15 : 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.

13.6 Functies met een variabele argumentenlijst

De `printf`- en `scanf`-functies hebben een variabele argumentenlijst en zijn te gebruiken met één of meer argumenten. De functie `printf` heeft minimaal één argument — de formatstring — en heeft hier één, twee, of zes argumenten:

```
printf("formatstring zonder specifier");
printf("formatstring met een specifier %d", 2);
printf("formatstring met vijf specifiers %d %d %d %d %d", 2, 3, 4, 5, 6);
```

De functie `fprintf` heeft minstens twee argumenten, namelijk een filepointer en een formatstring:

```
fprintf(fp , "formatstring is tweede argument");
fprintf(fp , "formatstring is tweede argument met een specifier %d", 42);
```

De C-bibliotheek bevat het headerbestand `stdarg.h` met een typedefinitie en een aantal macro's om zelf een functie met een variabele argumentenlijst te maken. In code 13.19 staat een voorbeeld van een functie `average`, die het gemiddelde van een aantal getallen berekent.

Code 13.19 : De functie `average` berekent het gemiddelde van een aantal getallen.

```
1  #include <stdarg.h>
2
3  double average(int numberOfValues, ... )
4  {
5    va_list values;
6    double sum = 0;
7    int i;
8
9    va_start(values,  numberOfValues);
10   for (i=0; i < numberOfValues; i++) {
11     sum += va_arg(values, double);
12   }
13   va_end(values);
14
15   return  sum/numberOfValues;
16 }
```

De drie puntjes heten in het Engels *ellipsis* en dat is te vertalen met ellips of weglating.

Het eerste argument van de functie is het aantal getallen dat gemiddeld moet worden. De rest van de argumenten is variabel. In de functieheader op regel 5 is dit aangegeven met ... en bij onderstaande aanroepen worden er respectievelijk twee en vier getallen meegegeven als argument:

```
avg2 = average(2, 23.4, 31.2);                  // averaging 2 numbers
avg4 = average(4, 23.4, 31.2, 13.4, 91.6);      // averaging 4 numbers
printf(f, "%.1f %.1f", avg2, avg4);             // print 27.8 39.9
```

Om de argumenten te kunnen benaderen, moet in de functie een variabele gedeclareerd zijn van het type `va_list`. In code 13.19 is dit op regel 5 de variable `values`. Deze variable wordt bij de aanroep van de macrodefinities `va_start`, `va_arg` en `va_end` steeds als eerste argument meegegeven. Op regel 9 definieert `va_start` het begin van de argumentenlijst. In de functieheader staan de variabele argumenten altijd na deze variabele. De macrodefinitie `va_arg` geeft het eerstvolgende argument uit de argumentenlijst terug en schuift daarna één positie in de lijst op. Op regel 11 gebeurt dit `numberOfValues` keer, waarbij het resultaat steeds bij `sum` opgeteld wordt.

De bezwaren tegen een variabele argumentenlijst

Een groot bezwaar van een variabele argumentenlijst is dat de compiler geen typechecking kan doen op de argumenten en een ander bezwaar is dat er eenvoudig een verkeerd aantal argumenten kan worden meegegeven. Als bij `average` het opgegeven aantal kleiner is dan het aantal meegegeven argumenten, rekent de functie een verkeerd gemiddelde uit. Als het opgegeven aantal groter is dan het aantal argumenten, geeft de functie een onzin antwoord terug. Als de meegegeven argumenten van een andere type zijn, ziet de compiler dat niet en geeft `average` een onzinnige waarde terug:

```
avg2 = average(2, 23.4, 31.2, 13.4);   // average of 23.4 and 31.2 is 27.3
agv3 = average(3, 23.4, 31.2);         // average of 23.4, 31.2 and ?? is ??
avg4 = average(4, 23.4, 6, "string", '.');     // the average is ??
printf("%.1f %.1f %.1f", avg2, avg3, avg4);    // print 27.3 ?? ??
```

Een praktisch voorbeeld met de variabele argumentenlijst

Er zijn situaties waarbij een variabele argumentenlijst veel gebruikt wordt. Een voorbeeld is de exit-functie, die voordat het programma afgesloten wordt eerst een geformatteerde tekst afdrukt.

Code 13.20 : De functie `exit_with_message` drukt eerst een geformatteerde tekst.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdarg.h>
4
5  void exit_with_message(const char *fmt, ...)
6  {
7    va_list vl;
8
9    va_start(vl, fmt);
10   vfprintf(stderr, fmt, vl);
11   fprintf(stderr, "\n");
12   va_end(vl);
13
14   exit(1);
15 }
```

De functie `exit_with_message` uit code 13.20 drukt de formatstring `fmt` af met de argumenten die achter deze formatstring staan. Bij de aanroep

```
exit_with_message("Usage: %s <file_in> [file_out]", argv[0]);
```

verschijnt deze tekst op de uitvoer `stderr`

```
Usage: prgname <file_in> [file_out]
```

als `prgname` de naam van het programma is.

13.7 Preprocessoropdrachten of *compiler directives*

In paragraaf 6.7 is de **`#define`** voor het maken van macro's en macrodefinities besproken. De term **`#define`** is — net als **`#include`** — een zogenoemde *compiler directive*. Deze opdrachten staan altijd op een aparte regel en worden meestal helemaal links uitgelijnd. Tabel 13.4 geeft de meest gebruikte *compiler directives*.

Tabel 13.4 : **De meest gebruikte *compiler directives*.**

Compilatie-opdracht	Betekenis met een voorbeeld
`#define`	definieert macro of macrodefinitie `#define MACRO 5`
`#include`	sluit headerbestand in `#include <studio.h>`
`#error`	stopt compiler en geeft foutmelding `#error "some message"`

`#line`	past regelnummer aan
	`#line 45 thisfile.c`
	`#line 23`

`#pragma`	implementatie van systeemafhankelijke opdrachten `#pragma vector=TIMER0_OVF_vect`
`#undef`	verwijderd macro of macrodefinitie `#undef MACRO`

De preprocessor kent ook twee operatoren **#** en **##**. De eerste zet een teken om naar een string en de tweede plakt twee tekens aan elkaar.

Een `#pragma` wordt gebruikt om compiler- en systeemafhankelijke opdrachten uit te voeren. De IAR-compiler gebruikt in tegenstelling tot de AVR GNU C-compiler pragma's bij de definities van interruptfuncties en interruptvectoren:

```
// IAR style
#pragma vector=TCE0_OVF_vect
__interrupt void functionName()
{
    // code
}
```

```
// AVR GNU style

ISR(TCE0_OVF_vect)
{
    // code
}
```

Voorwaardelijke preprocessoropdrachten

Naast de gewone preprocessoropdrachten kent C ook een aantal voorwaardelijke preprocessoropdrachten of *conditional compiler directives*. In tabel 13.5 staan deze voorwaardelijk preprocessoropdrachten. Met deze opdrachten compileert de compiler onderdelen uit de code alleen als er aan een bepaalde voorwaarde is voldaan. Een verschil met de gewone conditionele `if`-statements is dat niet de compiler maar de processor deze voorwaardelijke opdrachten interpreteert. Een ander onderscheid is dat deze opdrachten net als de andere *compiler directives* altijd op een aparte regel staan:

```
#ifdef MACRO
    printf("The value MACRO is %d\n", MACRO);
#else
    printf("The value MACRO doesn't exist\n");
#endif
```

Bovenstaande code drukt de waarde van de macro `MACRO` af als deze macro bestaat en een mededeling als `MACRO` niet gedefinieerd is.

Een ander voorbeeld is de test op de compiler die gebruikt wordt. De bibliotheken van Atmel met extra drivers voor de Xmega gebruiken vaak een bestand `avr_compiler.h` waarin definities staan die bestemd zijn voor de Imagecraft Com-

Tabel 13.5 : **Voorwaardelijke preprocessoropdrachten.**

Opdracht	Betekenis
#if *condition*	start voorwaardelijke opdracht
#else	sluit voorwaardelijke opdracht af en start alternatief
#endif	sluit voorwaardelijke opdracht af
#elif *condition*	sluit voorwaardelijke opdracht af en start nieuwe opdracht
#ifdef *definition*	start voorwaardelijke opdracht als definitie bestaat
#ifndef *definition*	start voorwaardelijke opdracht als definitie niet bestaat

De uitdrukking
  #if defined(condition)
is identiek met
  #ifdef condition
en
  #if !defined(condition)
is identiek met
  #ifndef condition

piler en definities die bestemd zijn voor de AVR GNU C-compiler. Dit headerbestand bevat deze voorwaardelijke **if**-elif-**else**-constructie:

```
#if defined(__ICCAVR__)
  // stuff for ICC (Imagecraft Compiler)
#elif defined(__GNUC__)
  // stuff for GNUC (GNU Compiler)
#else
  #error Compiler not supported.
#endif
```

Twee strepen __ bij een naam of definitie geeft aan dat deze standaard bij de compiler bekend is.

Een andere toepassing is de bescherming tegen het meerdere keren insluiten van een headerbestand. Een UART-bibliotheek heeft bijvoorbeeld een headerbestand `uart.h` met een aantal definities. Het headerbestand definieert een speciale macro `_UART_H_`:

```
#ifndef _UART_H_
#define _UART_H_ 1
```

De waarde van deze macro is niet relevant. De h-bestanden en c-bestanden, die van de bibliotheek gebruik maken, kunnen nu testen of het headerbestand `uart.h` al is ingesloten:

```
#ifndef _UART_H_
#include "uart.h"
#endif
```

Tabel 13.6 : **Vooraf gedefinieerde macro's.**

Opdracht	Betekenis
`__LINE__`	het huidig regelnummer
`__FILE__`	volledige naam van het huidige bronbestand
`__DATE__`	datum van de start van de compilatie
`__TIME__`	tijd van de start van de compilatie
`__STDC__`	geeft aan dat het een standaard C-compiler is
`__GNUC__`	geeft aan dat het een GNU C-compiler is

Vooraf gedefinieerde macro's

Bij voorwaardelijke opdrachten en bij de bijbehorende foutmeldingen gebruikt men vaak macro's, die de compiler standaard kent. Tabel 13.6 geeft een aantal van deze vooraf gedefinieerde macro's.

14

De Xmega

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

Onderwerpen

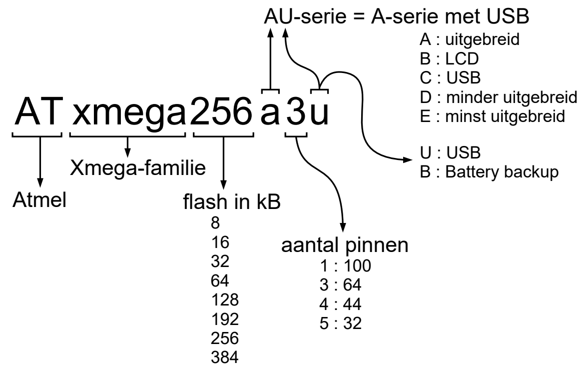
De behandelde onderwerpen zijn:

- De Xmega-familie van Atmel.
- De opbouw van de Xmega.
- De behuizing en de pinout van de Xmega.
- De geheugenorganisatie van de Xmega. Besproken worden het SRAM-, het EEPROM- en het flashgeheugen.
- De lock- en fusebits.
- Het kloksysteem van de Xmega.
- De methoden waarmee de Xmega kan worden geprogrammeerd, namelijk: via PDI, via de JTAG-interface en via USB.
- De ontwikkelomgeving voor de Xmega.

Dit boek is geschreven voor de Xmega256a3u van Atmel, een microcontroller met een AVR-architectuur uit de Xmega-familie. AVR staat volgens sommigen voor *advanced virtual RISC*. 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 van de Xmega256a3u staat in figuur 14.2. De instructieset is klein en door het gebruik van *pipelining* duren bijna alle instructies maar één klokslag.

De ATmega-familie kent een aantal varianten voor specifieke toepassingen, zoals LCD, PWM en CAN.

Atmel introduceerde in 1996 de 8-bits AVR-microcontrollers. Aanvankelijk waren er twee families: de ATtiny en de ATmega. De ATmega heeft meer pinnen, geheugen en mogelijkheden dan de ATtiny. In 2007 is daar de ATXmega-familie aan toegevoegd. Atmel noemt zelf de ATXmega een 8/16-bits microcontroller, hoewel het een gewone 8-bits microcontroller is met alleen meer 16-bits mogelijkheden. De ATXmega — of kortweg Xmega — heeft meer geheugen en meer mogelijkheden dan de ATmega, maar belangrijker is dat de opbouw meer gestructureerd is en dat de opzet van software daarop is aangepast. De manier van werken met een Xmega is duidelijk anders dan die met de ATmega of de ATtiny.



Figuur 14.1 : De naamgeving bij Xmega-familie. Het eerste getal geeft de grootte van het flashgeheugen in KB, de letter geeft de complexiteit van de component, het tweede getal beschrijft het aantal pinnen en de laatste letter geeft de variant aan.

De officiële naam is ATxmega256a3u, Atmel noemt de familie de AVR Xmega. Sommige spreken van een ATXmega, andere noemen het een Xmega. Dit boek spreekt van een Xmega en noemt de component Xmega256a3u.

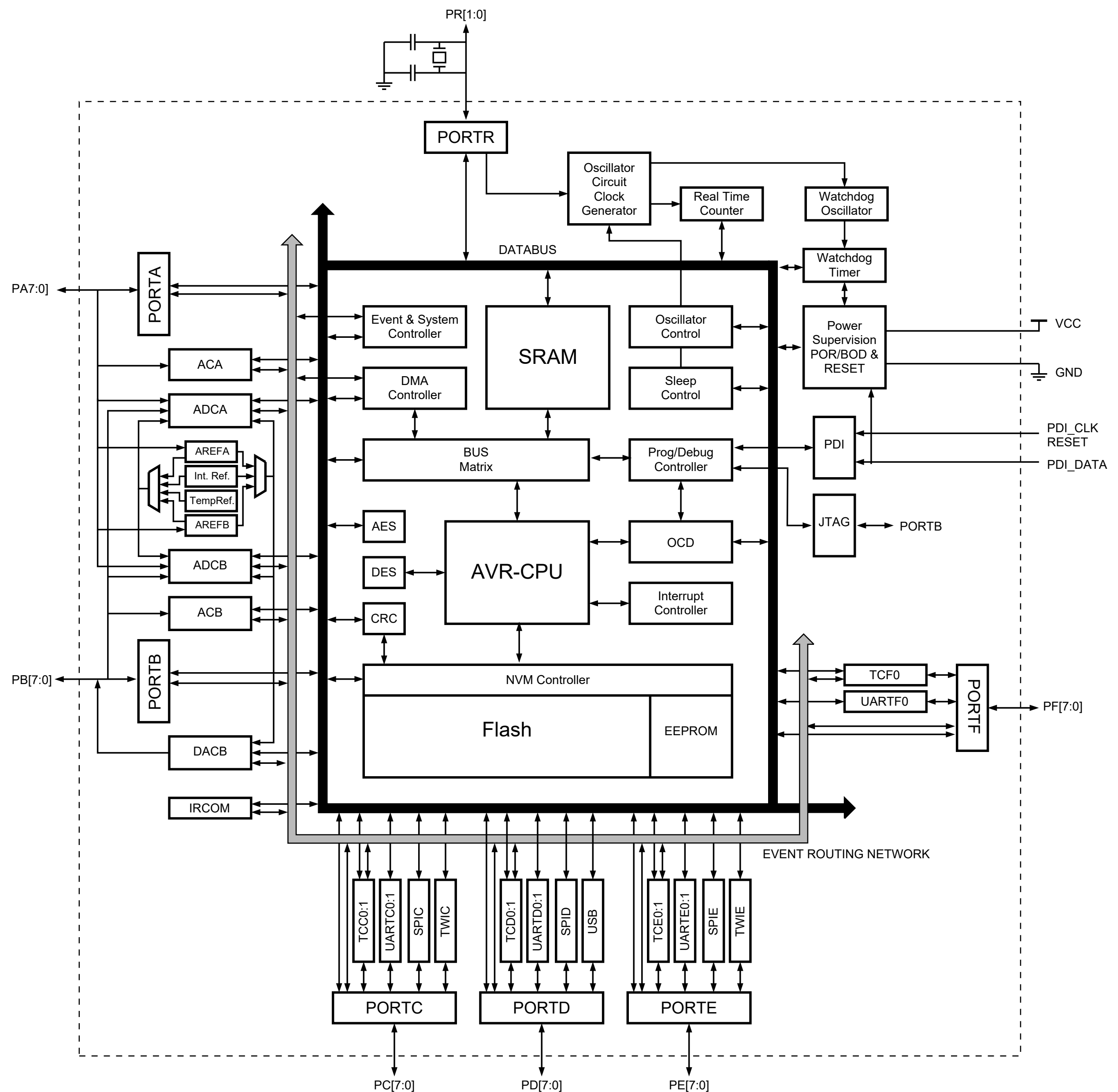
De Xmega-familie kent een aantal series: de A-, B- , C-, D-, E- en AU-serie. De naam van de component geeft ruwweg aan om wat voor soort microcontroller het gaat. Figuur 14.1 verklaart de naam ATXmega256a3u. De cijfers en letters beschrijven de complexiteit van de schakeling, de grootte van het flashgeheugen en het aantal pinnen.

De datasheets bij de Xmega bestaan altijd uit twee delen: een algemene manual voor de serie en een datasheet voor een specifieke component of een groep van componenten. Voor de Xmega256a3u bevat de zogenoemde AU-manual de gedetailleerde beschrijving van alle onderdelen en alle registers. De specifieke datasheet van de Xmega256a3u geeft een beknopte beschrijving van alle onderdelen en geeft gedetailleerde informatie over de elektrische en andere karakteristieken van de component.

De documentatie is — net als de datasheets van andere microcontrollers — zeer uitgebreid. Dit hoofdstuk vat de belangrijkste, algemene aspecten uit de datasheet samen. In latere hoofdstukken wordt aan de hand van praktische voorbeelden meer gedetailleerde informatie over bepaalde eigenschappen en functionaliteiten 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.

14.1 De opbouw van de Xmega

Het blokdiagram van figuur 14.2 geeft de opbouw van Xmega. In het centrum staat de AVR-CPU, de processor met de AVR-architectuur. De schematische opbouw van deze processor komt overeen met het blokschema uit figuur 1.9 van paragraaf 1.4.



Figuur 14.2 : **Het blokdiagram van de Xmega.** Het hart van de microcontroller is bedacht door Alf-Egil Bogen en Vegard Wollan en is aangegeven met AVR-CPU.

SRAM staat voor *static random access memory*.

EEPROM staat voor *electrical erasable programmable read only memory*.

De processor communiceert via een speciale busmatrix met het SRAM, en via de NVM-controller met het flashgeheugen en het EEPROM. NVM staat voor *non volatile memory* oftewel niet vluchtige geheugens. De processor heeft een verbeterde RISC-architectuur met een gescheiden programmabus en databus. In figuur 14.2 is de databus en het *event routing network* duidelijk zichtbaar. Het event-systeem is een verzameling eigenschappen waardoor perifere onderdelen met elkaar kunnen communiceren zonder tussenkomst van de CPU.

Het blokdiagram van figuur 14.2 is gebaseerd op een Xmega256a3u. Deze Xmega heeft 64 pinnen met 50 I/O-aansluitingen, die verdeeld zijn over de poorten A, B,

C, D, E, F en R. De poorten A tot en met F hebben 8 aansluitingen. Poort R heeft slechts 2 aansluitingen. Iedere poort heeft een aantal specifieke functies. Bij poort A en B bevinden zich de analoge functies en bij de poorten C, D, E en F de digitale interfaces. Andere componenten uit de Xmega-familie hebben meer poorten met een andere functionaliteiten of hebben soms 4 aansluitingen bij sommige poorten.

In het blokdiagram van figuur 14.2 is direct te zien dat de Xmega heel veel speciale functies kent. De belangrijkste features van de Xmega256a3u zijn:

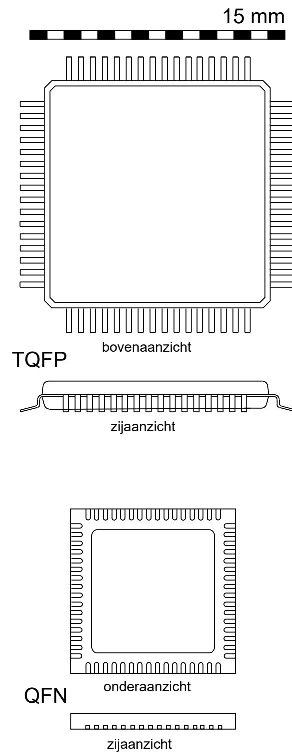
- Twee 12-bits ADC's, *analog-to-digital converters*;
- Twee blokken met twee analoge comparatoren;
- Een 12-bits DAC, *digital-to-analog converter*;
- Twee TWI's, *two-wire serial interfaces*, dat zijn tweedraads aansluitingen die gebruikt kunnen worden als I^{2}C-interfaces;
- Drie SPI's, *serial peripheral interfaces*, dat zijn verbindingen voor seriële communicatie;
- Zeven USART's, *universal, synchronous and asynchronous receiver and transmitter*, voor RS232-communicatie;
- Een USB-interface (Universal Serial Bus), die ook gebruikt kan worden om de microcontroller te programmeren en te debuggen;
- Zeven 16-bits real-time counters, die als timer gebruikt kunnen worden en waarmee ook PWM-signalen gemaakt kunnen worden;
- Per poort twee interrupts, die met iedere pin verbonden kunnen worden;
- Een JTAG-interface voor het debuggen en het programmeren;
- Een DMA-module voor *direct memory access*;
- Een event-system, waarmee direct tussen verschillende blokken gecommuniceerd kan worden zonder tussenkomst van de CPU;
- Een infrarood communicatiemodule;
- Een CRC module, *cyclic redundancy check generator*;
- Twee cryptografische modules voor AES, *advanced encryption standard*, en DES, *data encryption standard*;
- Een *watchdog timer*;
- Een *brown-out detection*;
- Verschillende *sleep modes*;
- Interne oscillatoren, extra aansluitingen voor externe oscillatoren en een PLL, *phase locked loop*, voor het instellen van een nauwkeurige klok;
- Een PDI, *program debug interface*, dat is een tweedraads aansluiting voor het programmeren en debuggen.

PWM staat voor *pulse width modulation* en betekent pulsbreedtemodulatie. Deze vorm van modulatie wordt veel gebruikt, bij bijvoorbeeld de aansturing van motoren.

Al deze mogelijkheden zijn zeer uitgebreid beschreven in de datasheet van de Xmega256a3u en de bijbehorende AU-manual. Daarnaast heeft Atmel een groot aantal *application notes* gepubliceerd met voorbeelden.
In de volgende hoofdstukken komen een groot aantal onderwerpen aan de orde. De belangrijkste functies worden aan de hand van praktische voorbeelden besproken. Dit boek pretendeert niet om volledig te zijn. Niet alle features komen aan bod en van de besproken features worden alleen de belangrijkste aspecten uitgediept.

Tabel H.1 in bijlage H geeft een overzicht van de belangrijkste handleidingen en *application notes* van Atmel.

14.2 De behuizing van de Xmega

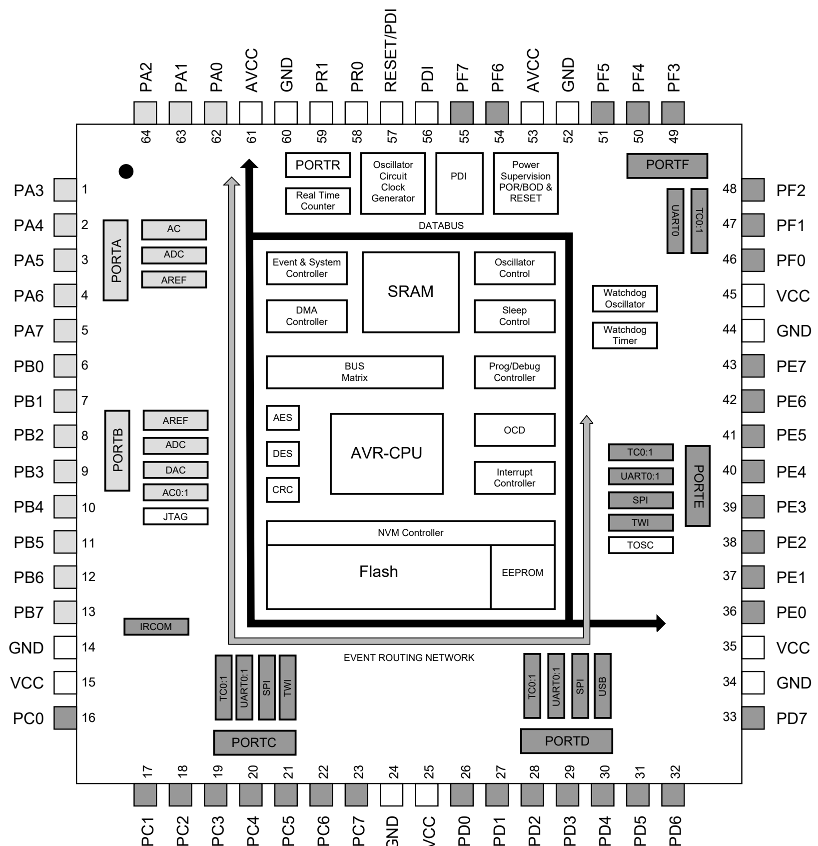


Figuur 14.3 : De behuizingen van de Xmega256a3u.

De Xmega-microcontrollers zijn alleen als SMD, *surface mounted devices*, verkrijgbaar. De Xmega256a3u is leverbaar met deze behuizingen:

TQFP : thin quad flat package
QFN : thin quad flat no-lead package

Figuur 14.3 laat deze behuizingen zien. Het grootste package, de TQFP, is slechts 16 bij 16 mm. Het kleinste package, de QFN, is ongeveer 9 bij 9 mm.



Figuur 14.4 : De pinout bij de 64-pins TQFP- of QFN-behuizing van de Xmega256a3u. De analoge interfaces bevinden zich bij de lichtgrijs gekleurde aansluitingen en de digitale interfaces bij de donkergrijs gekleurde aansluitingen.

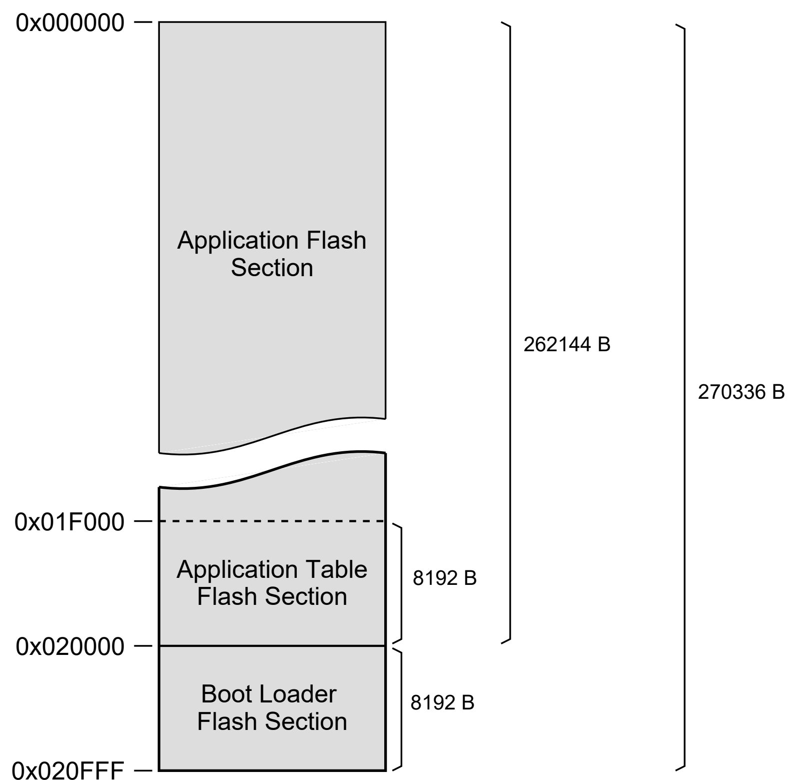
Figuur 14.4 is een interpretatie van figuur 2.1 uit de datasheet met de *pinout* van de TQFP- of VQFN-behuizing. De Xmega256a3u kent 50 bruikbare aansluitingen. De andere zijn de voedingspinnen en de aansluitingen voor de PDI, de *program debug interface*, waarmee de microcontroller geprogrammeerd en gedebugd kan worden. Elke bruikbare aansluiting kan naast een gewone in- of uitgang ook de in- of uitgang zijn van een speciale functie. Bij de poorten A en B bevinden zich de analoge interfaces en bij de poorten C, D, E en F bevinden zich de digitale interfaces.

14.3 De geheugenorganisatie bij de Xmega

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

Het flashgeheugen

In het flashgeheugen staat het programma van de applicatie die de microcontroller uitvoert. Dit geheugen wordt daarom ook het programmageheugen genoemd. Het flash is bij de Xmega256a3u ruim 256 KB groot en is bereikbaar via de PDI, *program debug interface* en vanuit de applicatie in de microcontroller.



Figuur 14.5 : De indeling van het programmageheugen bij de Xmega256a3u. Het applicatiedeel is $2^{17} = 262144$ bytes en de bootsector is $2^{13} = 8192$ bytes groot.

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.

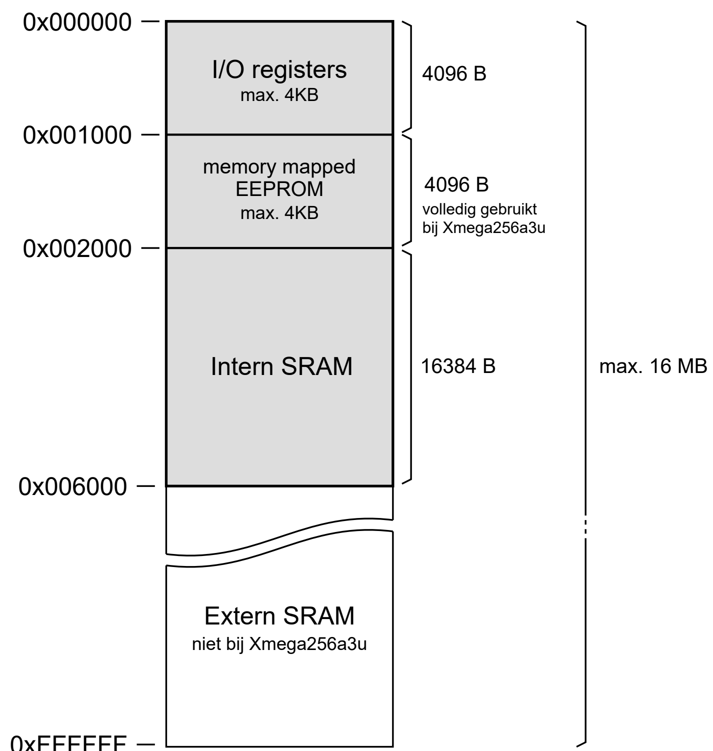
Figuur 14.5 geeft een overzicht van de indeling van het flashgeheugen voor de Xmega256a3u. Omdat de instructies 16-bits of 32-bits breed zijn, is de breedte van het flash 16-bits. Op ieder adres in het programmageheugen staat dus een *word*.

Het applicatiedeel van het flashgeheugen bevat een gedeelte — de *application table flash section* — dat gebruikt kan worden voor een veilige opslag van niet-vluchtige gegevens. Een deel van dit flashgeheugen kan gereserveerd worden als *boot sector*, daar is dan ruimte voor een *boot loader*. Als de bootsector niet gebruikt wordt, is het beschikbaar voor de applicatie van de microcontroller.

SRAM

In het SRAM worden de gegevens bewaard die de applicatie tijdens de uitvoering nodig heeft. Een SRAM-geheugen is vluchtig. Als de spanning wegvalt, gaan de gegevens in het SRAM verloren.

Het SRAM-geheugen en alle zogenoemde IO-registers, inclusief de laagste 16 *general purpose registers* worden op dezelfde manier geadresseerd. Figuur 14.6 geeft overzicht van alle datageheugens inclusief de adressering. Het EEPROM-geheugen kan op dezelfde manier als de andere datageheugens geadresseerd worden. Men spreekt dan van een memory-mapped geheugen.



Figuur 14.6 : De indeling van de datageheugens bij de Xmega en bij de Xmega256a3u in het bijzonder.

De dataregisters, het EEPROM-geheugen en het SRAM-geheugen zijn één byte breed. Op ieder adres staat bij deze geheugens dus één byte. De IO-registers staan op adressen `0x0` tot en met `0x0fff`. Het memory-mapped EEPROM en het SRAM beginnen bij iedere type Xmega op de adressen `0x1000` en `0x2000`. De Xmega256a3u heeft een SRAM van 16384 bytes.

Bij een aantal Xmega's is het mogelijk om direct via een EBI, *external bus interface*, externe geheugens, zoals een SDRAM, aan te spreken. De adressen van de externe geheugens komen na de adressen van het interne SRAM. De Xmega256a3u kent deze mogelijkheid niet.

EEPROM

Het EEPROM is 4096 bytes groot en Atmel garandeert dat het minimaal 80.000 keer geschreven en gewist kan worden. Het EEPROM kan op twee manieren geadresseerd worden: via een eigen, specifieke adressering en als een memory-mapped geheugen. Met de laatste mogelijkheid kan alleen uit het EEPROM worden gelezen. Om het geheugen leeg te maken of te beschrijven, is altijd de specifieke adressering nodig. Dit wordt gedaan via de NVM-controller en een aantal registers uit het IO-registerdeel van het datageheugen. In de datasheet wordt uitgelegd hoe deze registers gebruikt moeten worden om data in het EEPROM te schrijven en uit te lezen.

De lockbits en fusebits

De Xmega heeft een aantal lockbits waarmee onder andere ingesteld kan worden dat het flash of de bootsector niet meer geherprogrammeerd kan worden. Daarnaast zijn er een aantal fusebytes met fusebits. Met deze bytes worden de resetmogelijkheden van bijvoorbeeld de brownout-detector, de watchdog en de opstartconfiguratie ingesteld.

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 fuse- en lockbits worden apart geprogrammeerd. De meeste programmeerprogramma's hebben hier een apart menu voor. De fusebits kunnen door de applicatie worden gelezen, maar niet geschreven. De lockbits kunnen ook door de applicatie worden gewijzigd, maar alleen naar een strengere vorm van *locking*. Lockbits kunnen alleen worden gewist door een volledige *chip erase*.

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

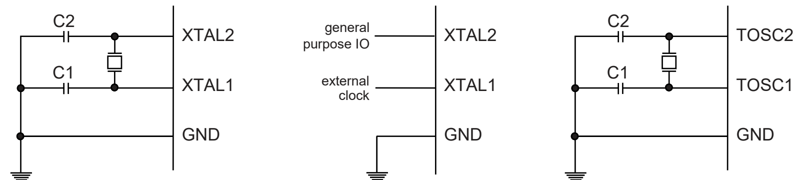
14.4 De systeemklok en klokopties

De Xmega kan gebruik maken van verschillende klokken. Het heeft een aantal interne oscillatoren en aansluitingen voor een externe kristaloscillator en resonator. Met een PLL, *phase locked loop*, en met prescalers zijn veel verschillende klokfrequenties mogelijk. De maximale klokfrequentie is 32 MHz bij 3,3 V.

Intern beschikt de Xmega over gekalibreerde RC-oscillatoren van 32 kHz, 2 MHz en 32 MHz. De 32-kHz-oscillator is bedoeld voor real-time counters. Extern kan er een kristaloscillator van 0,4 MHz tot 16 MHz worden aangesloten. De PLL kan de frequentie van het kloksignaal vermenigvuldigen en met de prescalers wordt de frequentie gedeeld. Bovendien is het mogelijk een extern kloksignaal aan te sluiten.

Bij de Xmega kan de klokbron in de applicatie worden aangepast. Standaard gebruikt de Xmega de interne klok van 2 MHz. Het programma start dan met 2 MHz en bij de initialisatie stelt het programma de klok in op bijvoorbeeld 32 MHz of schakelt het over op een externe oscillator.

In figuur 14.7 staan de schema's voor de verschillende klokconfiguraties. Externe kristallen worden aangesloten aan de pinnen `XTAL1` en `XTAL2` of de pinnen `TOSC2` en `TOSC1` voor de real-time counter. De twee condensatoren hebben dezelfde waarden en zijn nodig om de oscillator de juiste laadcapaciteit te geven.
Een extern kloksignaal wordt aangesloten op `XTAL1`. De andere aansluiting kan dan functioneren als een gewone in- of uitgang.



Figuur 14.7 : **De schema's voor de oscillatoren.** Links is de kristaloscillator voor de systeemklok aangesloten op pin `XTAL1` en `XTAL2`. In het midden staat de aansluiting voor een externe klok. Rechts is de kristaloscillator voor de real-time counter aangesloten op pin `TOSC1` en `TOSC2`.

14.5 Het programmeren van de Xmega

Er zijn drie methoden waarmee de Xmega geprogrammeerd kan worden:

- serieel via de PDI,
- serieel via de JTAG-interface.
- serieel via USB.

Al deze drie methoden zijn een vorm van ISP, *in-system programming*. In de manual en de datasheets staat de nodige informatie en er zijn ook speciale application notes over het programmeren van de Xmega.

PDI, Program Debug Interface

Alle Xmega's hebben een speciale module voor het serieel programmeren en het debuggen: de PDI oftewel de *program debug interface*. Er zijn twee aansluitingen voor de klok en de data. De klokpin is tegelijkertijd de resetpin. In figuur 14.4 is voor de Xmega256a3u pin 57 de klokpin en pin 56 de datapin.

Om de Xmega via de PDI te programmeren is een programmer nodig. Een programmer is een elektronische schakeling die de computer op de juiste manier, bijvoorbeeld via USB, verbindt met de programmeeraansluiting van de microcontroller. Bij de programmer hoort een softwareprogramma dat de microcontrollerapplicatie via de programmer in het programmageheugen van de microcontroller schrijft.

JTAG-interface

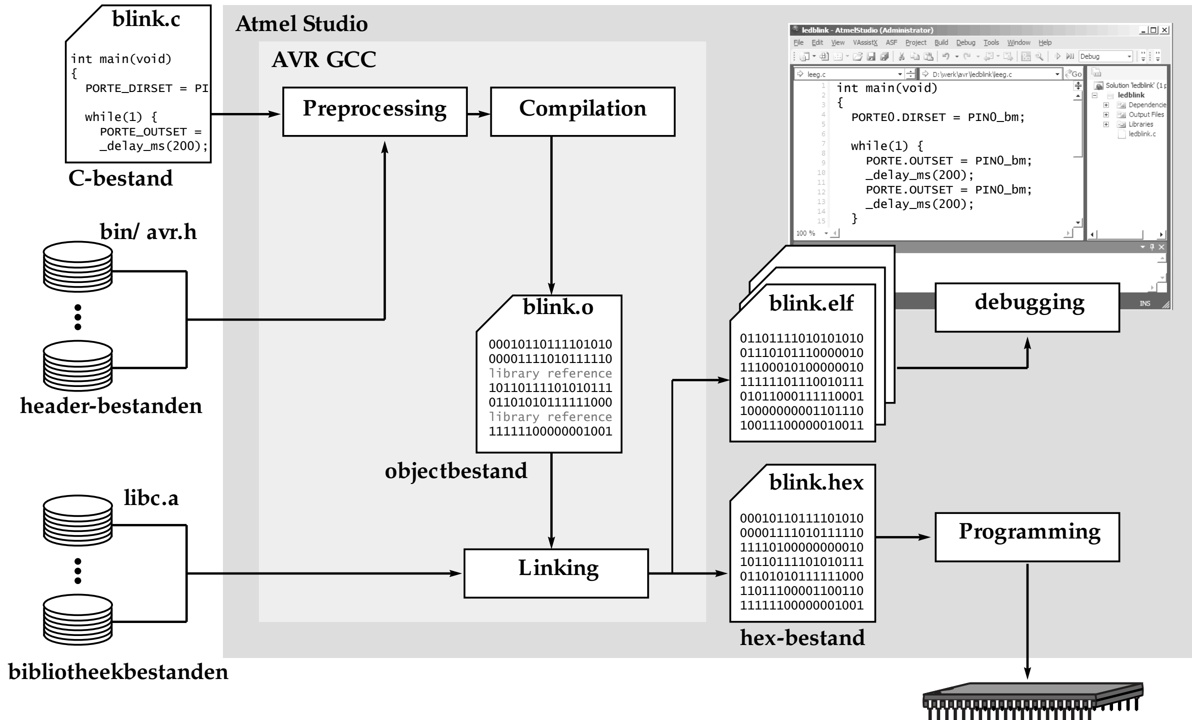
Een PCB is een *printed circuit board*.

Veel Xmega's hebben een JTAG-interface. 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. Veel digitale IC's hebben speciale logica en extra aansluitingen voor dit testprotocol.
De JTAG-interface is ook geschikt om componenten te programmeren en te debuggen. Bijlage C geeft meer achtergronden van JTAG. Voor de Xmega en de andere microcontrollers van Atmel bestaan speciale JTAG-programmers.
De Xmega256a3u heeft, net als alle andere 64-pins Xmega's, een JTAG-interface.

USB-interface en boot-loader

USB staat voor *universal serial bus*.

De Xmega's met een USB-interface kunnen worden geprogrammeerd met een boot-loader, die zich in de bootsector van het programmageheugen bevindt. Via de USB kan het programma in het applicatiedeel van het programmageheugen worden geprogrammeerd. Er is geen speciale programmer nodig. Wel is er een speciale applicatie nodig om de Xmega op deze manier te programmeren.



Figuur 14.8 : De ontwikkelomgeving met Atmel Studio en AVR-gcc. Het compilatietraject met AVR-gcc is vergelijkbaar met dat van de gewone GNU-compiler, zie ook 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 gebruikt Atmel Studio bij het debuggen en simuleren. Rechtsboven staat de gebruikersinterface van Atmel Studio.

14.6 De ontwikkelomgeving voor de Xmega

Voor het programmeren van de Xmega gebruikt dit boek Atmel Studio. Dat is de ontwikkelomgeving van Atmel en gebruikt de AVR GNU C-compiler als C-compiler. Met Atmel Studio kan de code worden gesimuleerd, geprogrammeerd en gedebugd.

De gecompileerde code van het programma staat in de vorm van hexadecimale getallen in het hex-bestand. Daarom noemt men dit ook wel de hex-code.

Figuur 14.8 geeft het compilatietraject met Atmel Studio en AVR-gcc. Het compilatietraject met AVR-gcc komt overeen met dat van de gewone GNU-compiler uit figuur 2.4. Het verschil tussen figuur 2.4 en figuur 14.8 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.

Met een PDI- of JTAG-programmer kan de microcontroller rechtstreeks vanuit Atmel Studio worden geprogrammeerd. Voor het programmeren via USB is een aparte applicatie nodig.

15

Generieke IO

Doelstelling	Dit hoofdstuk start met 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 een drukknop op Xmega aansluit en hoe je de ingang uitleest.

Onderwerpen

De behandelde onderwerpen zijn:

- De aansluiting van een led aan de microcontroller.
- De opbouw van de generieke IO van de Xmega: het `DIR`-, `OUT`- en `IN`-register.
- De adressering van het `DIR`-, `OUT`- en `IN`-register.
- Het maken van vertragingstijden met `_delay_us` en `_delay_ms`.
- De instelling van de klokfrequentie met de constante `F_CPU`.
- De bitwerkingen om een bit te setten en te clearen.
- De *read-modify-write*-methode.
- De Xmega-stijl om het IO-register te setten en te clearen.
- De aansluiting van een drukknop aan de microcontroller met een externe en interne pullupweerstand.
- Contactdender, antidenderschakelingen en antidenderalgoritmen.

Vier voorbeelden demonstreren het knipperen van één of meer leds:

- Een knipperende led met de Xmega-stijl.
- Een knipperende led met de *read-modify-write*-methode.
- De twee knipperende leds met de Xmega-stijl.
- Knipperende leds met de toggle-functie.

Vijf voorbeelden demonstreren het uitlezen van een drukknop:

- Een led die de status van een drukknop weergeeft.
- Een led aan en uit te zetten met een drukknop.
- Een functie die detecteert dat de drukknop ingedrukt is.
- Een functie die detecteert of de drukknop losgelaten wordt.
- Een functie met parameters, die detecteert of de drukknop losgelaten wordt.

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 micro-

controller 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.

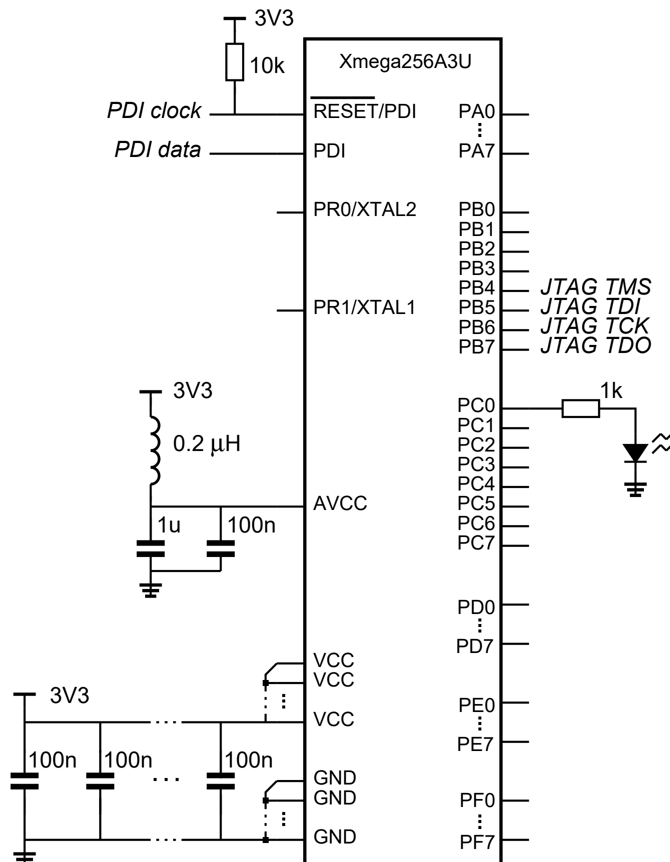
De *debugger* wordt ook wel simulator genoemd. Sommigen gebruiken het woord *debuggen* alleen bij het debuggen van een microcontroller via de JTAG-interface en spreken van *simuleren* als de microcontroller wordt gesimuleerd. Anderen gebruiken in beide gevallen hiervoor het woord debuggen. In beide gevallen 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.

Ieder 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.

Bij de voorbeelden komen meestal meerdere varianten van de code aan de orde. De eerste is daarbij niet altijd de beste of de meest logische keuze. Stapsgewijs wordt de code besproken van eenvoudig naar meer complex. Verschillende stijlen, notaties en methodieken worden toegepast.

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

Bij het Xmega-bord uit bijlage J is ook een led aangesloten op pin 0 van poort C. De stroom door de led wordt daar geregeld met een stroomspiegel.



Figuur 15.1 : Het schema voor het knipperen van een led.

15.1 De schakeling voor Led Blink

Figuur 15.1 geeft een minimale configuratie voor een Xmega256a3u om de software te testen. De led is aangesloten op pin `PC0`. De weerstand van 1 kΩ beperkt de stroom door de led en de microcontroller. De waarde van de weerstand hangt af van de voedingsspanning, de eigenschappen van de led en van de gewenste lichtsterkte. Omdat de Xmega een actief lage reset heeft, is deze met 10 kΩ weerstand aangesloten aan VCC.

Voor de weerstand R geldt:

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

Met U_{cc} 3,3 V, U_{led} 2,0 V en I_{led} 20 mA geeft dat R 65 Ω is. Voor de E12-reeks is dit afgerond 68 Ω.

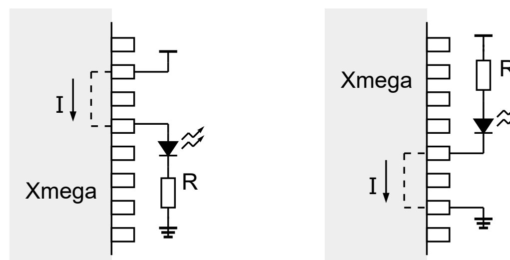
Dit is de minimale waarde voor de weerstand. Moderne leds, met een hoge lichtintensiteit, branden bij een stroom van 20 mA voor de meeste toepassingen te fel. Een weerstandswaarde van 470 Ω of 1 kΩ voldoet meestal.

In paragraaf 17.2 staat een aantal andere methoden om de led aan te sturen.

Er is geen kristal aangesloten; er wordt gebruik gemaakt van de interne oscillator. Alle digitale voedingspinnen, VCC, zijn verbonden met de voedingsspanning van 3,3 V. Via een kleine inductie — een zogenaamde ferrietkraal — is de analoge voeding, AVCC, eveneens verbonden met de voedingsspanning. Alle GND-pinnen zijn verbonden met de referentie. Bovendien is bij alle voedingspinnen een 100 nF capaciteit aangebracht om stoorsignalen te onderdrukken.

De Xmega256a3u kan geprogrammeerd worden met een PDI-programmer, die op de PDI- en de PDI/reset-pin is aangesloten, of via een JTAG-interface, die op poort B is aangesloten.

De led kan op twee manieren worden aangesloten, namelijk zodanig dat de microcontroller de stroom levert of dat de microcontroller de stroom afvoert. Figuur 15.2 toont deze twee methoden.



Figuur 15.2 : 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, *electromagnetic interference*. Dit is het versturen van en het ontvangen van ongewenste elektromagnetische straling. EMC, *electromagnetic 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 Xmega mag per pin maximaal 20 mA leveren of afvoeren. Vanwege allerlei EMI-aspecten blijft het nog steeds beter om de microcontroller stroom te laten afvoeren in plaats van te laten leveren.

15.2 De software voor Led Blink

In code 15.1 staat de software voor het laten knipperen van een led, die aangesloten is op pin `0` van poort C. In deze code zijn drie delen te onderscheiden:

- Het stuk voor de `main` — de preamble — met definities en de insluitingen van h-bestanden.
- Het deel in de `main` voor de oneindige **`while`** met de initialisatie.
- De oneindige **`while`** met de feitelijke functionaliteit.

Code 15.1 : De software voor het laten knipperen van een led.

```
 1 #define  F_CPU 2000000UL
 2
 3 #include <avr/io.h>
 4 #include <util/delay.h>
 5
 6 int main(void) {
 7   PORTC.DIRSET = PIN0_bm;       // bit 0 port C is set, it is an output
 8
 9   while (1) {
10     PORTC.OUTSET = PIN0_bm;     // bit 0 port C is high, led is on
11     _delay_ms(250);
12     PORTC.OUTCLR = PIN0_bm;     // bit 0 port C is low, led is off
13     _delay_ms(250);
14   }
15 }
```

De aansluitingen van de Xmega kunnen een ingang of een uitgang zijn. Standaard zijn alle aansluitingen ingang. Om de led aan te sturen, moet aansluiting `0` van poort C een uitgang zijn. Dit wordt in code 15.1 op regel 7 ingesteld.
De definities van `PORTC`, `DIRSET` en `PIN0_bm` staan in het headerbestand `avr/io.h`, dat op regel 3 is ingesloten.

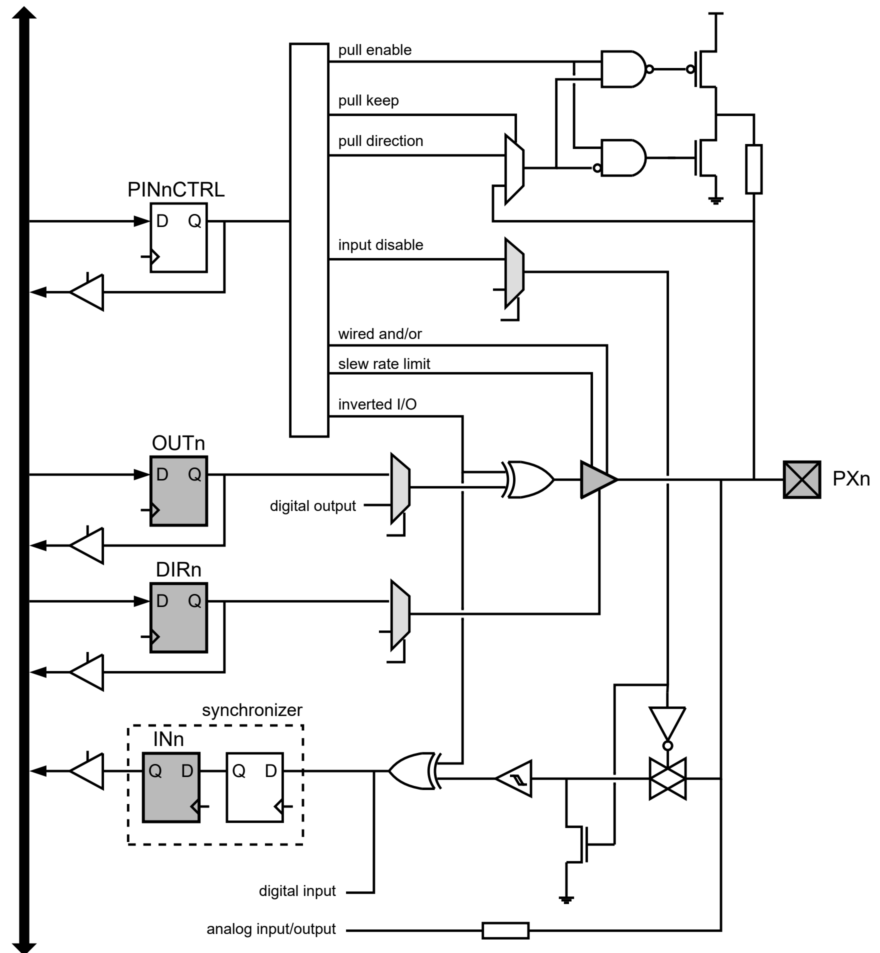
De oneindige **`while`** op regel 9 bevat de feitelijke functionaliteit van het programma. Eerst wordt op regel 10 de uitgang, aansluiting `0` van poort C, hoog gemaakt en wordt er 250 ms gewacht. Daarna wordt de uitgang weer laag gemaakt en wordt er weer 250 ms gewacht.
De macro `_delay_ms` is gedefinieerd in het headerbestand `util/delay.h` en berekent uit de klokfrequentie en de opgegeven vertragingstijd in milliseconden het aantal klokslagen dat gewacht wordt. De klokfrequentie `F_CPU` is gedefinieerd op regel 1 en is 2 MHz, de standaardwaarde van de Xmega. De definitie van `F_CPU` moet voor de insluiting van `delay.h` staan. De macrodefinitie `F_CPU` vertelt de compiler alleen wat de frequentie van de klok is. De klokfrequentie van de microcontroller wordt door deze definitie niet veranderd.

De software zorgt ervoor dat de uitgang steeds 250 ms hoog en 250 ms laag is, zodat de led met een frequentie van 2 Hz knippert.

15.3 De generieke IO van de Xmega

In figuur 15.3 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 uitgangseigenschappen van de IO-poort te maken heeft en een paar daaraan gelieerde functies, zoals het ingangscircuit en een pullup- en pulldownschakeling.

Afgebeeld is de IO van een willekeurige aansluiting. Bij de Xmega kan dat bit `0` tot en met 7 van poort A, B, C, D, E, F of R zijn. De `x` in de naam van de aansluiting `PXn` is de letter van de betreffende poort. De `n` in de naam bij de flipfloppen `DIRn`, `OUTn` en `INn` en bij `PXn` is het bitnummer van de aansluiting.

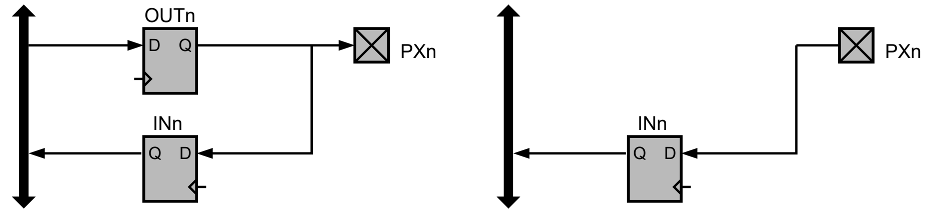


Figuur 15.3 : De generieke IO van de Xmega.
Deze figuur is een overdruk van figuur 13.10 uit de *Xmega-AU-manual* met een paar aanpassingen. De belangrijkste componenten voor de bespreking van de IO zijn donker grijs getint, namelijk: de aansluitpin `PXn`, de tristatebuffer en drie flipfloppen `DIRn`, `OUTn` en `INn`.
Bovenaan staat het `PINnCTRL`-register en de pullup- en pulldownschakeling. Rechtsonder bevindt zich een transmissiepoort en een schmitttrigger voor het ingangscircuit. De figuur toont alleen de generieke IO. Eventuele specifieke digitale of analoge aansluitingen zijn alleen genoemd, maar niet verder uitgewerkt.

`DIRn` is een flipflop uit het DIR-register van poort `X`. Als deze flipflop hoog is, is de grijsgetinte tristatebuffer ingeschakeld (*enabled*). De waarde van flipflop `OUTn` komt dan op de aansluitpin `PXn` te staan. De IO-poort werkt dan als uitgang. Als `DIRn` laag is, is de tristatebuffer gesperd (*disabled*). Er is dan geen directe verbinding tussen `OUTn` en de aansluitpin `PXn`; de IO-poort werkt dan als ingang. In figuur 15.4 zijn deze beide situaties sterk vereenvoudigd weergegeven.

`OUTn` is een flipflop uit het uitgangsregister van poort `X`. Alleen als `DIRn` hoog is, krijgt de aansluitpin `PXn` deze waarde van deze flipflop. De aansluitpin is via de transmissiepoort en de schmitttrigger verbonden met flipflop `INn`. Eén klokslag later staat de uitgangswaarde ook in flipflop `INn`, zie ook figuur 15.4.

`INn` 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.



Figuur 15.4 : De IO van de Xmega als ingang en als uitgang. De pullup- en pulldownschakeling, de synchronisatie flipflop, de transmissiegate en de schmitttrigger zijn hier weggelaten.
Links staat de configuratie van figuur 15.3 als `DIRn` hoog is. De aansluitpin is dan een uitgang en `INn` bevat dan het uitgangssignaal. Rechts is de configuratie van figuur 15.3 als `DIRn` laag is. De aansluitpin is dan een ingang.

Per poort zijn er maximaal acht aansluitpinnen. Bij de Xmega256a3u hebben alle poorten acht aansluitingen, alleen poort R heeft er twee. De Xmega-familie is zo opgezet dat het voor de beschrijving niet uitmaakt of er twee, vier of acht aansluitingen zijn. Sommige devices hebben bij poort B en poort E vier aansluitingen.

Tabel 15.1 : Een deel van de in- en uitgangsregisters van de Xmega. Voor poort C en D zijn de registers `DIR`, `OUT` en `IN` vermeld met de bitnummers. Tevens zijn de controlregisters `PIN0CTRL` en `PIN1CTRL` genoemd. Met bit `SRLEN` kan de *slew rate* van de uitgang worden aangepast en met bit `INVEN` kan de polariteit van een in- of uitgang worden omgekeerd. Met de drie bits `OPC[2:0]` wordt de pullup- en pulldown schakeling van de generieke IO ingesteld.

Adres	Registernaam		Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0640	PORTC	DIR	7	6	5	4	3	2	1	0
	...									
0x0644	PORTC	OUT	7	6	5	4	3	2	1	0
	...									
0x0648	PORTC	IN	7	6	5	4	3	2	1	0
	...									
0x0650	PORTC	PIN0CTRL	SRLEN	INVEN	OPC[2]	OPC[1]	OPC[0]	ISC[2]	ISC[1]	ISC[0]
0x0651	PORTC	PIN1CTRL	SRLEN	INVEN	OPC[2]	OPC[1]	OPC[0]	ISC[2]	ISC[1]	ISC[0]
	...									
0x0660	PORTD	DIR	7	6	5	4	3	2	1	0
	...									
0x0664	PORTD	OUT	7	6	5	4	3	2	1	0
	...									
0x0668	PORTD	IN	7	6	5	4	3	2	1	0
	...									
0x0670	PORTD	PIN0CTRL	SRLEN	INVEN	OPC[2]	OPC[1]	OPC[0]	ISC[2]	ISC[1]	ISC[0]
0x0671	PORTD	PIN1CTRL	SRLEN	INVEN	OPC[2]	OPC[1]	OPC[0]	ISC[2]	ISC[1]	ISC[0]

Er zijn per poort in ieder geval drie achtbits registers nodig voor `DIR`, `OUT` en `IN`. Daarnaast heeft iedere aansluiting een `PINnCTRL`-register. Bovendien zijn er nog een groot aantal registers die niet in figuur 15.3 zijn getekend, zoals de registers voor het instellen van de interrupts. In tabel 15.1 staat een aantal opeenvolgende registers van de Xmega. Ieder register heeft een geheugenadres, een naam en bestaat uit acht bits. De naam bestaat uit twee delen: de instantie `PORTC` waar het register bijhoort en de naam van het register `OUT`.

Bij het schrijven van software voor een microcontroller moeten de in- en uitgangsregisters en de registers van andere specifieke functies, zoals timers, digitale en analoge interfaces op de juiste wijze worden gemanipuleerd.

15.4 De organisatie van de registers bij de Xmega: de Xmega-stijl

De Xmega-familie is anders opgezet dan de populaire Mega-familie. De registers zijn anders georganiseerd en de software is anders opgezet. Naast de aparte macrodefinities voor de registers bevat `avr/io.h` ook typedefinities met speciale datastructuren. Bovendien hebben de generieke IO's bij de Xmega veel meer registers dan die bij de Mega. Er zijn registers toegevoegd, die het mogelijk maken om individuele bits in het `DIR`-register en in het `OUT`-register te wijzigen zonder de zogenoemd *read-modify-write* methode.

Het bestand `io.h` bevat geen definities. Het verwijst alleen verder naar `iox256a3u.h`, waarin de feitelijke definities van de Xmega256a3u staan. Bij een overstap naar een andere Xmega hoeft alleen de verwijzing te worden aangepast.

De datastructuur `port_t` voor de generieke IO uit `avr/io.h` staat in code 13.12 van paragraaf 13.4. Deze datastructuur is opgebouwd uit 24 bytes en is grafisch weergegeven in figuur 15.5.

0x00	DIR
0x01	DIRSET
0x02	DIRCLR
0x03	DIRTGL
0x04	OUT
0x05	OUTSET
0x06	OUTCLR
0x07	OUTTGL
0x08	IN
0x09	INTCTRL
0x0A	INT0MASK
0x0B	INT1MASK
0x0C	INTFLAGS
0x0D	reserved
0x0E	REMAP
0x0F	reserved
0x10	PIN0CTRL
0x11	PIN1CTRL
0x12	PIN2CTRL
0x13	PIN3CTRL
0x14	PIN4CTRL
0x15	PIN5CTRL
0x16	PIN6CRTL
0x17	PIN7CRTL

Figuur 15.5 :
De datastructuur `PORT_t`. De hexadecimale getallen geven de relatieve adressen van de registers.

De *strobe*-registers voor het `DIR`- en `OUT`-register

Naast het `DIR`-register zijn er drie zogenoemde *strobe*-registers, die de inhoud van `DIR` en daarmee de richting van de datastroom kunnen wijzigen: `DIRSET`, `DIRCLR` en `DIRTGL`. Bij een toewijzing aan `DIRSET`, `DIRCLR` of `DIRTGL` veranderen de bits in het `DIR`-register. Hieronder staan een aantal opeenvolgende toewijzingen aan `DIR`, `DIRSET`, `DIRCLR` en `DIRTGL` en het effect op de inhoud van `DIR`:

```
PORTC.DIR    = PIN0_bm;          // assign all bits      DIR is 00000001
PORTC.DIRSET = PIN2_bm;          // set bit 2,           DIR is 00000101
PORTC.DIRCLR = PIN0_bm;          // clear bit 0,         DIR is 00000100
PORTC.DIRTGL = PIN3_bm|PIN2_bm;  // toggle bit 3 and 2,  DIR is 00001000
PORTC.DIRSET = PIN2_bm|PIN0_bm;  // set bit 2 and 0,     DIR is 00001101
PORTC.DIRCLR = PIN3_bm;          // clear bit 3,         DIR is 00000101
```

De namen van de registers `DIRSET`, `DIRCLR` en `DIRTGL` geven aan wat de actie bij een toewijzing is. De bits die hoog zijn setten, clearen of toggelen de overeenkomstige bits van het `DIR`-register. De bits, die in `DIRSET`, `DIRCLR` of `DIRTGL` laag zijn, hebben geen invloed. Het setten, clearen of toggelen gebeurt dus alleen eenmalig bij de toewijzing aan `DIRSET`, `DIRCLR` of `DIRTGL`. De registers `DIRSET`, `DIRCLR` en `DIRTGL` kunnen niet worden uitgelezen. Als deze toch worden gelezen, is het resultaat altijd de inhoud van `DIR`.

`PIN0_bm`, `PIN2_bm` en `PIN3_bm` zijn gedefinieerd in `avr/io.h`. Dit zijn de bitmaskers voor pinnen `0`, 2 en 3. `PINi_bm` is het bitmasker voor bit `i`. De `i`^{e} bit van het masker is een 1 en de andere bits zijn `0`. Zo is `PIN3_bm` bijvoorbeeld gelijk aan `0b00001000` oftewel `0x08`.

Voor het `OUT`-register bestaan ook drie *strobe* registers, die de inhoud van `OUT` en uitgangswaarden kunnen wijzigen, namelijk: `OUTSET`, `OUTCLR` en `OUTTGL`. De werking van deze registers komt overeen met die van `DIRSET`, `DIRCLR` en `DIRTGL`.

De datastructuur voor de generieke IO

Hoewel het programma uit code 15.1 eenvoudig te lezen is, is het goed om de notatie bij de Xmega-stijl te analyseren. De stijl is gebaseerd op datastructuren. De gebruikelijke methode om een datastructuur te declareren kan bij de poorten niet worden gebruikt. Bij een gewone declaratie, zoals:

```
PORT_t poort;
```

alloceert de compiler ergens in het geheugen ruimte voor de datastructuur `PORT_t`. De variabelenaam `poort` is dan de naam van deze geheugenplaats. Het `OUTSET`-veld uit de datastructuur `poort` is dan te benaderen met:

```
poort.OUTSET = PIN0_bm;
```

De geheugenplaats van de registers is bij een microcontroller al bekend. De geheugenplaats van poort C uit code 15.1 ligt — net als de geheugenplaatsen van de anderen poorten — al vast. De registers van poort C bevinden zich bij de Xmega256a3u vanaf adres `0x0640`. Er is dus geen declaratie van de datastructuur nodig. Er is alleen een definitie nodig om deze geheugenplaats te benaderen. Deze definities staan in `iox256a3u.h` en in code 15.2 staat een fragment met de definities voor de generieke IO-poorten.

Code 15.2 : De macrodefinities voor de generieke io met de datastructuur `PORT_t`.

```
2877 #define PORTA    (*(PORT_t *) 0x0600)  /* I/O Ports */
2878 #define PORTB    (*(PORT_t *) 0x0620)  /* I/O Ports */
2879 #define PORTC    (*(PORT_t *) 0x0640)  /* I/O Ports */
2880 #define PORTD    (*(PORT_t *) 0x0660)  /* I/O Ports */
2881 #define PORTE    (*(PORT_t *) 0x0680)  /* I/O Ports */
2882 #define PORTF    (*(PORT_t *) 0x06A0)  /* I/O Ports */
2883 #define PORTR    (*(PORT_t *) 0x07E0)  /* I/O Ports */
```

Figuur 15.6 laat zien dat de betekenis van macro `PORTC` gelijk is aan de inhoud van de pointer die naar een datastructuur `PORT_t` wijst op het adres `0x0640`. De inhoud van een pointer naar een datastructuur is de datastructuur zelf. Bij `PORTC` is dat de datastructuur op locatie `0x0640`.

Figuur 15.6 : De betekenis van de macrodefinitie `PORTC`.

Het `OUTSET`-veld van `PORTC` is daarom in code 15.1 te benaderen met:

```
PORTC.OUTSET = PIN0_bm;
```

De punt tussen `PORTC` en `OUTSET` is de punt, die bij een datastructuur of een **`struct`** nodig is tussen het object (`PORTC`) en een veld (`OUTSET`) uit het object.

De overige registers uit `PORT_t`

Tot nu toe zijn alleen de registers gebruikt uit `PORT_t`, die de richting van de datastroom instellen en de waarden van het uitgangsregister bepalen. Het ingangsregister `IN` wordt in paragraaf 15.8 besproken. De registers `INTCTRL`, `INT0MASK` `INT1MASK` en `INTFLAGS` zijn nodig als de ingang een externe interrupt is en komen in hoofdstuk 16 aan de orde.
De bits in het register `REMAP` manipuleren de aansluitingen voor een aantal niet-generieke functies.

Voor iedere aansluiting van een poort, is er een pincontrolregister `PINnCTRL`. Deze registers, zie ook tabel 15.1, zijn nodig om onder andere de *slew rate* van een uitgang, de polariteit van de aansluiting en de pullup- en pulldownschakeling in te stellen. De onderstaande regels maken van aansluiting 3 van poort C een geïnverteerde uitgang:

```
PORTC.DIRSET    = PIN3_bm;         // pin 3 output
PORTC.PIN3CTRL |= PORT_INVEN_bm;   // pin 3 inverted
PORTC.OUTSET    = PIN3_bm;         // bit 3 high, output low
```

Bit 3 van het `OUT`-register wordt hoog gemaakt, waardoor de uitgang laag zal zijn. Het `PIN3CTRL`-register heeft — net als de meeste andere registers — geen set- en clearfuncties. Daarom is bij de toewijzing de *read-modify-write*-methode gebruikt, die in paragraaf 15.6 wordt behandeld.

15.5 De registers bij de verouderde ATmega-stijl

De ATmega, de voorganger van de Xmega, was minder gestructureerd opgebouwd dan de Xmega en de software was anders opgezet. Omdat deze stijl soms nog toegepast wordt bij de Xmega, wordt in deze paragraaf daar aandacht aan besteed. Het wordt afgeraden om deze stijl te gebruiken bij nieuwe ontwerpen.

Code 15.3 : Een knipperende led in de verouderde ATmega-stijl.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  int main(void) {
7    PORTC_DIR = _BV(0);
8
9    while (1) {
10     PORTC_OUT |= _BV(0);
11     _delay_ms(250);
12     PORTC_OUT &= ~(_BV(0));
13     _delay_ms(250);
14   }
15 }
```

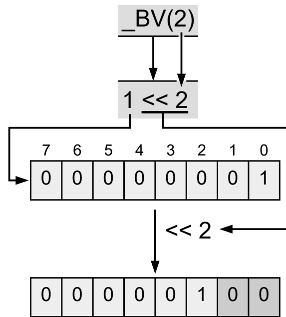
Een voorbeeld van een led blink in de verouderde ATmega-stijl staat in code 15.3. Er zijn drie belangrijke verschillen met code 15.1 De code gebruikt: geen *strobe*-registers, geen definities voor de pinnen en niet de datastructuur `PORTC`.

In plaats van de datastructuur `PORTC`, zijn er nu twee macrodefinities `PORTC_DIR` en `PORTC_OUT`, die direct naar de registers `DIR` en `OUT` van poort C wijzen. Deze macrodefinities staan in `iox256a3u.h`:

```
#define PORTC_DIR  _SFR_MEM8(0x0640)
#define PORTC_OUT  _SFR_MEM8(0x0644)
```

De macro `_SFR_MEM8` staat in `sfr_defs.h`, dat ook automatisch door `io.h` wordt ingesloten. De definities van `PORTC_DIR` en `PORTC_OUT` zijn na het invullen van de macro `_SFR_MEM8` als volgt te schrijven:

```
#define PORTC_DIR   ( * (volatile uint8_t *) (0x0640) );
#define PORTC_OUT   ( * (volatile uint8_t *) (0x0644) );
```



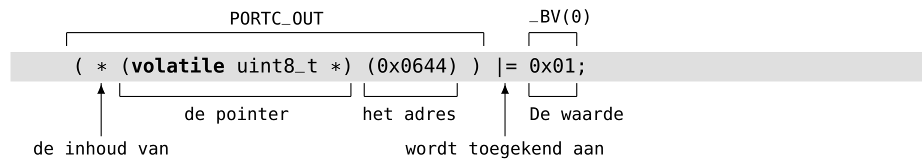
Figuur 15.7 : De betekenis van de macro `_BV(2)`. `_BV(2)` is hetzelfde als `1<<2`. Het getal 1 wordt 2 bits naar links geschoven en rechts aangevuld met nullen.

Het headerbestand `avr/sfr_defs.h` definieert ook de zogenoemde *bit value*:

```
#define _BV(bit) (1 << (bit))
```

Deze macro schuift de waarde 1 `bit` posities naar links. Deze uitdrukking maakt de betreffende bit `bit` hoog en de ander bits laag. De uitdrukking `_BV(0)` betekent `0b00000001` en de uitdrukking `_BV(2)` betekent `0b00000100`, zie ook figuur 15.7.

In figuur 15.8 is de betekenis van regel 10 uit code 15.3, die uitgang 0 van poort C hoog maakt, letterlijk uitgeschreven. Deze regel kent de waarde `_BV(0)`, oftewel `0x01`, toe aan de register `OUT` van `PORTC`. In woorden luidt deze regel dus als volgt: ken de waarde (`0x01`) toe aan de inhoud (∗) van het adres (`0x0644`) waar de pointer (`volatile uint8_t *`) naar wijst.



Figuur 15.8 : Met de macrodefinitie `PORTC_OUT` kunnen waarden aan het `OUT`-register van poort C worden toegekend.

De toekenning op regel 10 is niet gewoon een `=`, maar een `|=`. De Xmega heeft bij register `DIR` en bij register `OUT` drie *strobe*-registers om de individuele bits te manipuleren. De ATmega heeft deze niet en daarom moet bij deze stijl de *read-write-modify*-methode gebruikt worden om individuele bits te veranderen. De volgende paragraaf bespreekt deze methode en de bitbewerkingen die daar voor nodig zijn.

Op regel 7 staat bij de toewijzing aan `PORTC_DIR` wel alleen een `=` Dit betekent dat nu alle bits van `PORTC_DIR` worden overschreven. Aan het `DIR`-register wordt de waarde `0b00000001` toegekend.

15.6 Bitbewerkingen en de *read-write-modify*-methode

De Xmega heeft alleen extra *strobe*-registers bij de `DIR`- en `OUT`-registers van de poorten. Bij alle andere registers van de Xmega is om individuele bits te veranderen de *read-write-modify*-methode nodig. Deze methode wordt in deze paragraaf uitgelegd aan de hand van het `OUT`-register van poort C.

Als aan een register een waarde wordt toegekend, worden alle bits overschreven.

```
PORTC.OUT = PIN0_bm;
```

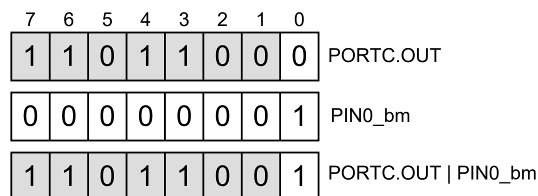
De bovenstaande toekenning maakt van register `OUT` bit `0` hoog en alle andere bits laag. In veel gevallen, zoals in code 15.3, is dat geen probleem. De aansluitingen 1 tot en met 7 worden hier niet gebruikt. In andere situaties moet alleen bit `0` hoog gemaakt worden, zonder de andere bits te veranderen. Dit kan door alle bits eerst te lezen en de bitsgewijze OF-functie te gebruiken.

```
PORTC.OUT = PORTC.OUT | PIN0_bm;
```

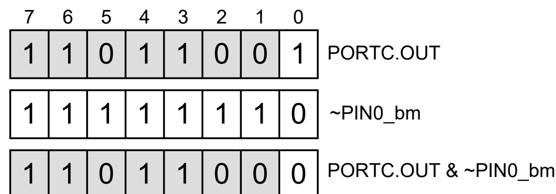
Bovenstaande toekenning leest de waarde van register `OUT`, verandert deze waarde met de bitsgewijze OF en kent deze uitkomst weer aan `OUT` toe. In figuur 15.9 is deze bitwerking gevisualiseerd. Bij toekenning wordt meestal de verkorte notatie uit paragraaf 9.13 gebruikt:

```
PORTC.OUT |= PIN0_bm;
```

De bewerking bestaat uit drie stappen: lezen van de huidige waarde, het veranderen van deze waarde en weer toekennen of terugschrijven van de nieuwe waarde. Deze methode van bewerken wordt daarom *read-modify-write* genoemd.



Figuur 15.9 : **De bitbewerking om één specifiek bit uit een register hoog te maken.** Omdat de bits 1 tot en met 7 van `PIN0_bm` `0` zijn, blijven bij de bewerking met de OF-functie de overeenkomstige bits van `PORTC.OUT` ongewijzigd. Alleen bit `0` verandert.



Figuur 15.10 : **De bitbewerking om één specifiek bit uit een register laag te maken.** Omdat de bits 1 tot en met 7 van `~PIN0_bm` hoog zijn, blijven bij de bewerking met de EN-functie de overeenkomstige bits van `PORTC.OUT` ongewijzigd. Alleen bit `0` verandert.

Op overeenkomstige wijze, zoals in figuur 15.10 is getekend, maakt de onderstaande bewerking bit `0` van register `OUT` laag.

```
PORTC.OUT = PORTC.OUT & ~PIN0_bm;
```

Bovenstaande bewerking is verkort te schrijven als:

```
PORTC.OUT &= ~PIN0_bm;
```

De bitsgewijze OF-functie `|` wordt gebruikt om een specifiek bit hoog te maken. De bitsgewijze EN-functie `&` en de inverse functie `~` worden gebruikt om een bit laag te maken.

Code 15.4 : **Twee knipperende leds in de Xmega-stijl.**

```
 1  #define  F_CPU 2000000UL
 2
 3  #include <avr/io.h>
 4  #include <util/delay.h>
 5
 6  int main(void) {
 7    PORTE.DIRSET = PIN3_bm|PIN2_bm;      // bit 2 and 3 port E are outputs
 8
 9    while (1) {
10      PORTE.OUTSET = PIN3_bm|PIN2_bm;    // outputs are high, leds are on
11      _delay_ms(250);
12      PORTE.OUTCLR = PIN3_bm|PIN2_bm;    // outputs are low, leds are off
13      _delay_ms(250);
14    }
15  }
```

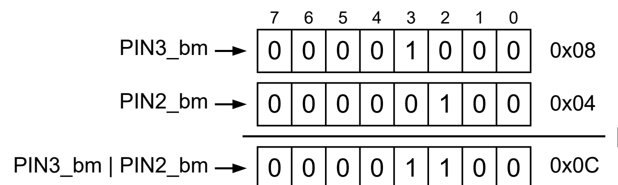
Code 15.4 laat twee leds knipperen die op pin 2 en pin 3 van poort E zijn aangesloten. Dit programma gebruikt bitmaskers voor de pinaansluitingen, die gedefinieerd zijn in `io.h` en in figuur 15.5 vermeld staan.

Code 15.5 : **Fragment uit `io.h` met de bitmaskers en bitposities voor de generieke IO.**

```
7044  #define PIN0_bm 0x01
7045  #define PIN0_bp 0
7046  #define PIN1_bm 0x02
7047  #define PIN1_bp 1
7048  #define PIN2_bm 0x04
7049  #define PIN2_bp 2
7050  #define PIN3_bm 0x08
7051  #define PIN3_bp 3
7052  #define PIN4_bm 0x10
7053  #define PIN4_bp 4
7054  #define PIN5_bm 0x20
7055  #define PIN5_bp 5
7056  #define PIN6_bm 0x40
7057  #define PIN6_bp 6
7058  #define PIN7_bm 0x80
7059  #define PIN7_bp 7
```

De bitpositie wordt in paragraaf 15.12 gebruikt en in paragraaf 16.7 besproken samen met het bitmasker, het groepsmasker, de groepspositie en de groepsconfiguratie.

De bitmaskers `PIN3_bm` en `PIN2_bm` zijn met de OF-functie gecombineerd tot één bitmasker waarvan bit 3 en bit 2 allebei hoog zijn, zie ook figuur 15.11.



Figuur 15.11 : **De samenstelling van het masker dat de bits 2 en 3 selecteert.**

In code 15.4 is op regel 7 het bitmasker `PIN3_bm|PIN2_bm` gebruikt om de pinnen 2 en 3 uitgang te maken en op regel 10 en regel 12 om deze uitgang hoog en laag te maken.

Er zijn veel verschillende methoden om een bitmasker op te schrijven. Hieronder staan zeven alternatieve bitmaskers voor de toewijzing van regel 10.

```
PORTE.OUTSET = 0x0C;
PORTE.OUTSET = 0b00001100;
PORTE.OUTSET = 12;
PORTE.OUTSET = (1<<3)|(1<<2);
PORTE.OUTSET = _BV(3)|_BV(2);
PORTE.OUTSET = 0x08|0x04;
PORTE.OUTSET = PIN3_bm|PIN2_bm;      // Xmega style
```

Sommigen menen dat de eerste drie beter zijn dan de andere vier, omdat er niets berekend hoeft te worden. De laatste vier alternatieven gebruiken de OF- en de schuiffuncties. Er is echter geen enkel verschil. De compiler berekent deze bitmaskers tijdens het compileren. Hooguit zal de compilatietijd iets langer duren. Al deze zeven toewijzingen doen precies hetzelfde. Dit boek gebruikt, vanwege de beste leesbaarheid, consequent de Xmega-stijl.

De bits in de registers kunnen ook één voor één worden veranderd.

```
PORTE.DIRSET = PIN2_bm;
PORTE.DIRSET = PIN3_bm;
```

Er zijn dan meer klokslagen nodig, maar bij het debuggen is dit vaak handig, omdat een aparte toewijzing eenvoudig weggecommentarieerd kan worden.

Code 15.4 gebruikt de *strobe*-registers `DIRSET` en `OUTSET` om bit 2 en 3 van register `DIR` en register `OUT` hoog te maken en register `OUTCLR` om de bits laag te maken. Er is daarom bij dit programma geen *read-modify-write* nodig.
Omdat bij gebruik van het *strobe*-register alleen een toewijzing wordt gedaan, is dit sneller dan de *read-modify-write*-methode. Er zijn dan twee in plaats van vijf klokslagen nodig voor het veranderen van een of meerdere bits.

Naast `OUTCLR` en `OUTSET` is er een derde *strobe*-register `OUTTGL`. Door een bit in dit register hoog te maken, verandert de betreffende bit in register `OUT` van waarde. Als de bit hoog was, wordt deze laag en als deze bit laag was, wordt deze weer hoog. Code 15.6 geeft een voorbeeld met twee leds.

Zonder `OUTTGL` kan dit ook gedaan worden met *read-modify-write* en de bitsgewijze XOR.
Regel 12 luidt dan:
`PORTF.OUT ^= PIN1_bm;`

Code 15.6 : Knipperende leds met de toggle-functie.

```
 1 #define  F_CPU 2000000UL
 2
 3 #include <avr/io.h>
 4 #include <util/delay.h>
 5
 6 int main(void) {
 7   PORTF.DIRSET = PIN1_bm|PIN0_bm;      // pin 1 and 0 port F output
 8   PORTF.OUTSET = PIN0_bm;              // pin 0 high
 9   asm volatile ("nop");
10
11   while (1) {
12     PORTF.OUTTGL = PIN1_bm;            // toggle bit 1
13     _delay_ms(500);
14     PORTF.OUTTGL = PIN0_bm;            // toggle bit 0
15     _delay_ms(500);
16   }
17 }
```

15.7 Vertragingstijden en de macrodefinitie F_CPU

Bij de bespreking van code 15.1 is al uitgelegd dat _delay_ms de macrodefinitie F_CPU gebruikt wordt om de vertragingstijd te bepalen. F_CPU moet voor de insluiting van util/delay.h gedefinieerd worden, anders gebruikt _delay_ms een standaardwaarde van 1 MHz en waarschuwt de gebruiker bij de compilatie:

```
Warning: F_CPU not defined for <util/delay.h>
```

De macro voor F_CPU hoeft niet per se in de code te staan. Deze kan ook ingesteld worden met de compileroptie -DF_CPU=2000000UL. Het voordeel is dat F_CPU bij een groot project dan eenmalig wordt ingesteld. Het nadeel is dat als F_CPU niet bekend is er fouten kunnen ontstaan. Dit is te ondervangen door de definitie voorwaardelijk toe te voegen.

De letters UL bij de frequentie geven aan dat het getal unsigned long is.

```
#ifndef F_CPU
  #define F_CPU 2000000UL
#endif
```

In bovenstaand codefragment is F_CPU 2 MHz als deze niet vooraf gedefinieerd is, anders behoudt F_CPU de originele waarde.

De macro F_CPU vertelt de compiler alleen wat de klokfrequentie is. De klokfrequentie van de microcontroller wordt door deze definitie niet gewijzigd. De klokfrequentie van de microcontroller hangt af van de instellingen van de interne oscillatoren en de instellingen en eventuele aanwezigheid van een externe oscillator. In paragraaf 23.5 staat beschreven hoe de klokconfiguratie van de Xmega256a3u aangepast kan worden.

Naast _delay_ms() definieert delay.h ook een macrofunctie _delay_us(), die een tijdvertraging in microseconden geeft. Beide functies verwachten een constante waarde als ingangsparameter. Bij een aanroep _delay_ms(x) zal de compiler — als x een variabele is — deze foutmelding geven:

```
Error: __builtin_avr_delay_cycles expects a compile time integer constant
```

Dit kan eventueel opgelost worden door een eigen vertragingsfunctie te maken. In code 15.7 staat een functie delay_MS(), die een 16-bits integer ms als ingangsparameter heeft.

Code 15.7 : Een vertragingsfunctie geschikt voor een variabele integer als parameter.

```
1  inline static void delay_MS(uint16_t ms)
2  {
3    for(uint16_t i=0; i<ms; i++) {
4      _delay_ms(1);
5    }
6  }
```

De systematische fout hangt af van het aantal extra klokslagen per iteratie, de stapgrootte t_{step} en de klokfrequentie F_{cpu}:

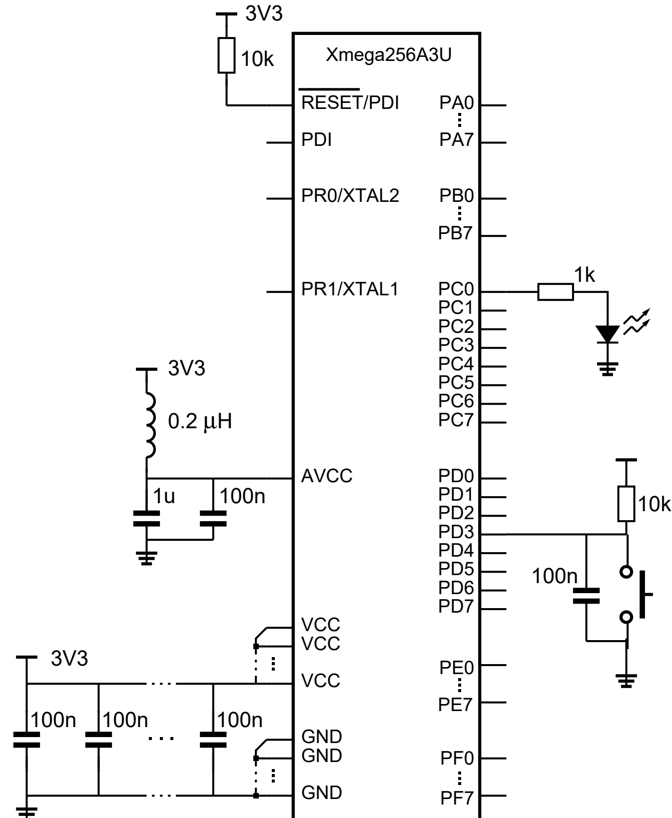
$$\frac{6}{t_{\text{step}} F_{\text{cpu}}} 100\%$$

De fout is bij een stapgrootte van 1 ms en een klok van 2 MHz 0,3%.

De vertragingstijd zal groter zijn, dan wordt opgegeven. Dit komt omdat er voor iedere iteratie van de **for**-lus zes extra klokslagen nodig zijn. De vertraging delay_MS(500) is bij een 2 MHz klok zodoende 501,5 ms. Een vergelijkbare functie voor microseconden is weinig zinvol omdat de fout dan veel te groot is.

15.8 De generieke IO als ingang gebruiken

De generieke IO kan ook gebruikt worden als ingang om signalen te lezen. Een basaal voorbeeld is een schakeling met een drukknop en een led. Als de drukknop ingedrukt wordt is de led aan en anders is de led uit.



Figuur 15.12 : De schakeling voor het aan- en uitzetten van een led met een drukknop.

De waarde van de condensator hangt af van de mate van contactdender.
Iedere type drukknop heeft een ander gedrag. Er bestaan ook drukknoppen, die nauwelijks last van contactdender hebben.

De schakeling van figuur 15.12 is de schakeling uit figuur 15.1 met daaraan toegevoegd een drukknop aan pin 3 van poort D. De weerstand en de condensator zijn optioneel. De condensator is toegevoegd om contactdender te voorkomen. Dit kan hard- of softwarematig ook op andere manieren worden voorkomen. In paragraaf 15.10 wordt contactdender besproken. De weerstand zorgt er voor dat als de knop niet ingedrukt is, de ingang van de microcontroller hoog is. Als de knop ingedrukt wordt, wordt de ingang laag.

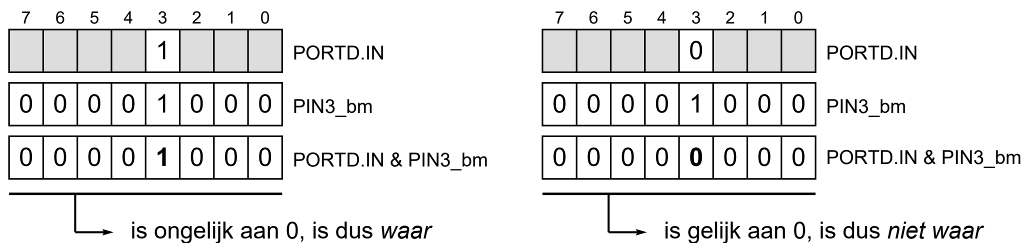
De datastructuur voor de generieke IO, de `struct PORT_t`, bevat het ingangsregister `IN`. Iedere klokslag wordt de inhoud van de ingangsregisters ververst met de waarden die er op dat moment op de aansluitpinnen staan.

In code 15.8 wordt op regel 4 aansluiting 3 van poort D expliciet gedefinieerd als ingang door bit 3 van het `DIR`-register van poort D laag te maken. Standaard zijn bij een reset alle bits van de registers laag. Zonder de toewijzing op regel 4 is deze aansluiting ook een ingang.

Code 15.8 : **Een led die de status van een drukknop weergeeft.**

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

int main(void) {
  PORTD.DIRCLR = PIN3_bm;  // input pin for button
  PORTC.DIRSET = PIN0_bm;  // output pin for led
  PORTC.OUTSET = PIN0_bm;  // led on

  while (1) {
    if ( PORTD.IN & PIN3_bm ) {
      PORTC.OUTCLR = PIN0_bm;   // button not pressed, led off
    } else {
      PORTC.OUTSET = PIN0_bm;   // button pressed, led on
    }
  }
}
```

In de oneindige `while`-lus wordt op regel 9 getest of de ingang hoog is. Voor deze test wordt de bitsgewijze EN-functie en het bitmasker voor pin 3 gebruikt.



Figuur 15.13 : **De test om te bepalen of bit 3 uit poort D hoog of laag is.** Alle bits, die in het masker `0` zijn, worden bij de bewerking `PORTD.IN & PIN3_bm` eveneens `0`. Afhankelijk van de waarde van bit 3 in `PORTD.IN` is de uitkomst gelijk of ongelijk aan `0`.

Figuur 15.13 laat zien dat de uitdrukking `PORTD.IN & PIN3_bm` waar is als de ingang hoog is en niet waar is als de ingang laag is. In het eerste geval is de drukknop *niet* ingedrukt. De uitgang is dan laag en de led is uit. De ingang is laag, zolang de drukknop ingedrukt wordt. De test is dan niet waar, zodat de toewijzing op regel 12 de uitgang hoog maakt en de led laat branden.

15.9 Het aan- en uitzetten van een led met een drukknop

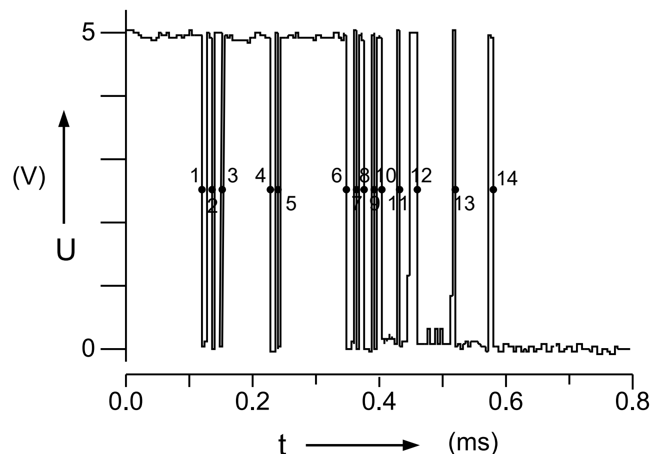
In code 15.8 gaat de led aan als de knop ingedrukt wordt en weer uit als de knop losgelaten wordt. Code 15.9 geeft een eerste aanzet van een programma dat de led laat aangaan als de knop wordt ingedrukt en weer laat uitgaat als er nogmaals op de knop wordt gedrukt.

Als de knop ingedrukt wordt, is de ingang laag. Op regel 12 staat bij de testconditie een `!`, waardoor de conditie waar is als de knop ingedrukt wordt. Op regel 13 wordt het register `OUTTGL` gebruikt om de uitgangswaarde te laten omklappen of *toggelen*. De tijdvertraging op regel 14 zorgt ervoor dat de led na één keer indrukken niet blijft toggelen. Alleen als de knop langer dan 200 ms wordt vastgehouden blijft de led aan- en uitgaan.

Code 15.9 : Een eerste aanzet om een led aan en uit te zetten met een drukknop.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  int main(void) {
7    PORTD.DIRCLR = PIN3_bm;  // input pin for button
8    PORTC.DIRSET = PIN0_bm;  // output pin for led
9    PORTC.OUTCLR = PIN0_bm;  // led off
10
11   while (1) {
12     if ( ! (PORTD.IN & PIN3_bm) ) {
13       PORTC.OUTTGL = PIN0_bm;   // toggles led
14       _delay_ms(200);
15     }
16   }
17 }
```

De oplossing uit code 15.9 is niet robuust. Als de schakelaar contactdender vertoont en er geen goede hardwarematige ontdenderschakeling wordt gebruikt, kan het gedrag van de schakelaar onvoorspelbaar worden.



Figuur 15.14 : De dender bij het indrukken van de drukknop. Het signaal gaat veertien keer van hoog naar laag voordat het definitief laag is geworden.

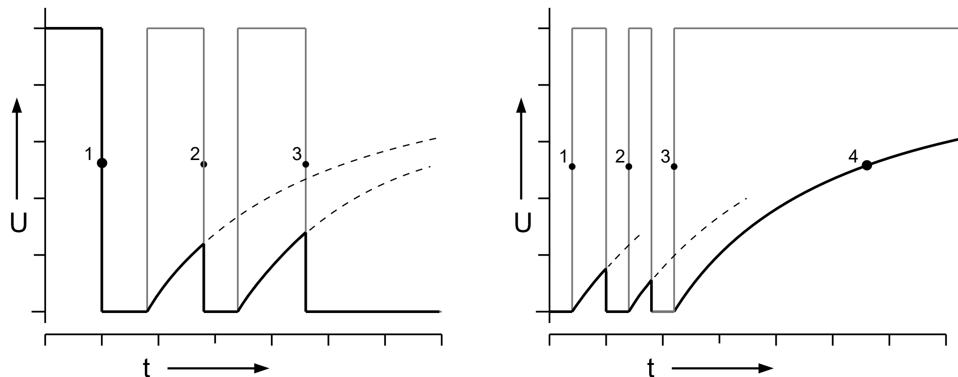
15.10 Contactdender

De meeste schakelaars en drukknoppen produceren contactdender. Het contact wordt niet in één keer gemaakt of in één keer verbroken. Figuur 15.14 toont de dender (*bouncing*) bij een maakcontact. In dit voorbeeld gaat het signaal niet één 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 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.

Als het signaal een oneven aantal keer van hoog naar laag gaat, is dat hetzelfde als dat het één keer van hoog naar laag gaat. Het signaal van figuur 15.14 gaat een even aantal keer, namelijk 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.

15.11 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 15.12 is het ingangssignaal gefilterd door een condensator parallel bij de drukknop te plaatsen.



Figuur 15.15 : 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.

Het effect van deze condensator is in figuur 15.15 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.

De schmitttrigger wordt besproken in bijlage F.9.

Omdat deze laatste overgang zo traag verloopt, is een schmitttrigger gewenst. In figuur 15.12 is geen schmitttrigger getekend, omdat er in de Xmega al een aanwezig is, zoals in figuur 15.3 te zien is.

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.

15.12 Softwarematige antidendermaatregelen

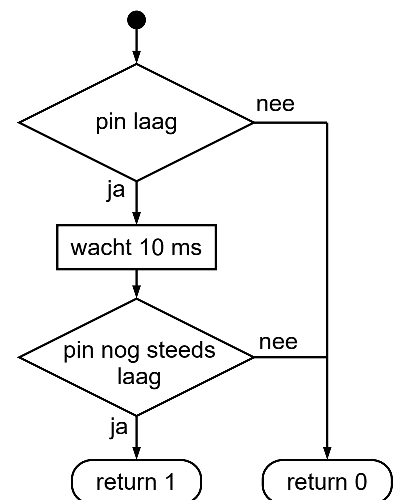
In paragraaf 16.6 staat een voorbeeld van een antidenderalgoritme met een timer en een interrupt. Het voordeel van deze methode is dat het hoofdprogramma ontlast wordt en de tijdplanning (*scheduling*) veel eenvoudiger is.

Dender (*bouncing*) is ook softwarematig te bestrijden. 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.

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 15.12 ook worden weggelaten.

Code 15.10 : De functie `button_pressed` voor het uitlezen van een drukknop.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define DEBOUNCE_PERIOD_MS   10
7  #define LOCK_PERIOD_MS       200
8
9  int button_pressed(void)
10 {
11   if ( bit_is_clear(PORTD.IN,PIN3_bp) ) {
12     _delay_ms(DEBOUNCE_PERIOD_MS);
13     if ( bit_is_clear(PORTD.IN,PIN3_bp) ) return 1;
14   }
15
16   return 0;
17 }
18
19 int main(void)
20 {
21   PORTD.DIRCLR   = PIN3_bm;           // input pin button
22   PORTD.PIN3CTRL = PORT_OPC_PULLUP_gc; // enable pull up
23   PORTC.DIRSET   = PIN0_bm;           // output pin led
24
25   while (1) {
26     if ( button_pressed() ) {
27       PORTC.OUTTGL = PIN0_bm;         // toggles output
28       _delay_ms(LOCK_PERIOD_MS);      // lock input
29     }
30   }
31 }
```



Figuur 15.16 : Het antidenderalgoritme om de drukknop uit te lezen. Als de ingang na 10 ms nog steeds laag is, is de knop ingedrukt.

In figuur 15.16 staat een antidenderalgoritme voor het uitlezen van een drukknop. Als de drukknop ingedrukt is, wordt er 10 ms gewacht. Mits deze tijd bij de knop past, zal het ingangssignaal stabiel zijn en geen dender meer vertonen. Als de pin dan nog steeds laag is, is de knop ingedrukt.

De functie `button_pressed` uit het programma van code 15.10 is een implementatie van antidenderalgoritme uit figuur 15.16. Het programma hoort bij figuur 15.12 waarbij de pullupweerstand en de condensator niet nodig zijn.

Het hoofdprogramma checkt voortdurend met de functie `button_pressed` of de knop ingedrukt is. Als dat het geval is, verandert de status van de led en wacht het programma 0,2 seconde. Als deze vertragingstijd wordt weggelaten, knippert de led een aantal keer. Een gebruiker houdt een drukknop altijd een fractie van een seconde ingedrukt. Na 0,2 seconde zal de knop losgelaten zijn. Als de gebruiker de knop langer ingedrukt houdt, zal de led blijven knipperen.

Uitleg code 15.10 regel 11
`bit_is_clear()`
`bit_is_set()`

Op regel 11 en op regel 13 is de macrodefinitie `bit_is_clear` gebruikt. Deze macro samen met de macro `bit_is_set` is gedefinieerd in `avr/sfr_defs.h`. Dit headerbestand wordt automatisch ingesloten als `avr/io.h` wordt gebruikt. De macrodefinitie `bit_is_set` test of een bit hoog is en `bit_is_clear` test of een bit laag is. Deze macrodefinities hebben twee ingangsparameters `sfr` en `bit`:

```
#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
#define bit_is_set(sfr, bit)   ( (_SFR_BYTE(sfr) & _BV(bit)))
```

Parameter `sfr` is de naam van het register en `bit` het nummer van de bit. De uitdrukking:

```
bit_is_clear(PORTD.IN,PIN3_bp)
```

is identiek met:

```
! (PORTD.IN & PIN3_bm)
```

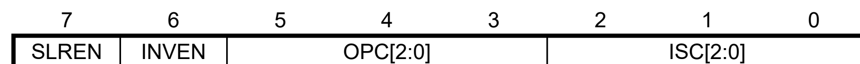
Bij `bit_is_clear` en `bit_is_set` wordt het bitnummer (`PIN0_bp`) gebruikt en niet het bitmasker (`PIN0_bm`).

Regel 11
`PIN3_bp`

De macro's `PINn_bp` zijn gedefinieerd in `avr/io.h` en representeren bitnummers of bitposities. De macro `PIN3_bp` komt overeen met de waarde 3. Verwar de bitpositie `PIN3_bp` niet met het bitmasker `PIN3_bm`, zie ook code 15.5.

Regel 22
`PORTD.PIN3CTRL`

Met het pincontrolregister kan een individuele pinaansluiting worden aangepast. `PORTD.PIN3CTRL` is het pincontrolregister van pin 3 van poort D. Figuur 15.17 toont de inhoud van dit register. `SRLEN` staat voor *slew rate limit enable*, hiermee wordt



Figuur 15.17 : Het pincontrolregister `PINnCTRL`.

de slew rate gelimiteerd. `INVEN` staat voor *inverted i/o enable*, hiermee wordt de polariteit van de in- of uitgang geïnverteerd. `OPC` staat voor *output and pull configuration*. Deze bits stellen onder andere de pullup- en pulldownschakeling van de generieke IO in. `ISC` staat voor *input/sense configuration*. Deze bits regelen de triggergevoeligheid van een ingang bij interrupts en events.

Regel 22
`PORT_OPC_PULLUP_gc`

De drie `OPC`-bits uit het `PINnCTRL`-register worden gebruikt om de aansluiting als wired-and of wired-or te configureren en om de pullup- en pulldownschakeling in te stellen. Tabel 15.2 geeft de verschillende instelmogelijkheden.
In code 15.10 hebben de `OPC`-bits de waarde `PORT_OPC_PULLUP_gc` en heeft ingang 3 van poort D een interne pullupweerstand.

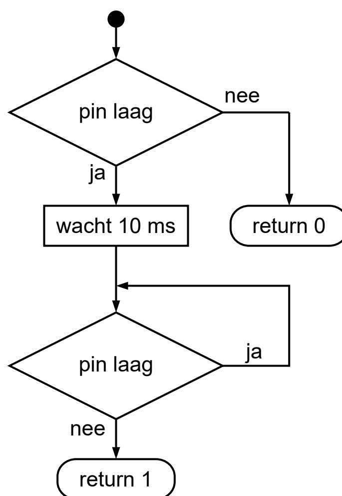
Alternatief antidenderalgoritme dat altijd maar één keer reageert

In code 15.10 is de ontdendertijd 10 ms en wacht het hoofdprogramma 0,2 s. De tijdvertragingen voor het ontdenderen en het wachten hangen sterk af van het type knop en van wat de applicatie moet doen. Als de knop langer dan 0,2 s wordt ingedrukt, wordt er opnieuw een druk op de knop geregistreerd. Dat kan

Tabel 15.2 : De uitgangs- en pullup/pulldown-configuratie.

OPC[2:0]	groepsconfiguratie	betekenis
000	PORT_OPC_TOTEM_gc	normaal
001	PORT_OPC_BUSKEEPER_gc	buskeeper
010	PORT_OPC_PULLDOWN_gc	pulldown
011	PORT_OPC_PULLUP_gc	pullup
100	PORT_OPC_WIREDOR_gc	wired-OR
101	PORT_OPC_WIREDAND_gc	wired-AND
110	PORT_OPC_WIREDORPULL_gc	wired-OR met pulldown
111	PORT_OPC_WIREDANDPULL_gc	wired-AND met pullup

ongewenst zijn. Natuurlijk kan de wachttijd langer gemaakt worden, maar dan bestaat juist de kans dat een korte indruk van de knop gemist wordt. Bij een toetsenbord is het juist plezierig dat de aanslag voortdurend herhaald wordt zolang de toets ingedrukt blijft.



Figuur 15.18 : Het antidenderalgoritme dat wacht op het loslaten van de drukknop.

Code 15.11 : Alternatief voor uitlezen drukknop. Deze functie `button_pressed` detecteert dat de drukknop losgelaten wordt.

```
 9 int button_pressed(void)
10 {
11   if ( bit_is_clear(PORTD.IN,PIN3_bp) ) {
12     _delay_ms(DEBOUNCE_PERIOD_MS);
13     while ( bit_is_clear(PORTD.IN,PIN3_bp) ) ;
14     return 1;
15   }
16   return 0;
17 }
```

Een alternatief is om de functie `button_pressed` te laten wachten totdat de drukknop losgelaten wordt. Het algoritme voor deze methode staat in figuur 15.18 en de alternatieve functie `button_pressed` staat in code 15.11. Het enige verschil met de functie uit code 15.10 is dat de **`if`** op regel 26 vervangen is door een **`while`**.

Het nadeel van de functie `button_pressed` uit code 15.11 is dat het programma in de functie 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 zal functioneren.

Alternatieve functie `button_pressed` met parameters

De functies `button_pressed` uit code 15.10 en code 15.11 werken alleen bij de aansluiting 3 van poort D. Voor een andere aansluiting moeten de functies worden aangepast. In code 15.12 staat een functie `button_pressed` met twee parameters: een pointer `p` die naar de poort wijst en een integer `bit` voor het bitnummer.

Code 15.12 : De functie `button_pressed` met parameters.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define LOCK_PERIOD_MS       200
7
8  int button_pressed(PORT_t *p, uint8_t bit)
9  {
10   if ( bit_is_clear(p->IN, bit) ) {
11     _delay_ms(10);
12     loop_until_bit_is_set(p->IN, bit);
13     return 1;
14   }
15   return 0;
16 }
17
18 int main(void) {
19   PORTD.DIRCLR   = PIN3_bm;            // input pin button
20   PORTD.PIN3CTRL = PORT_OPC_PULLUP_gc; // enable pull up
21   PORTC.DIRSET   = PIN0_bm;            // output pin led
22
23   while (1) {
24     if ( button_pressed(&PORTD, PIN3_bp) ) {
25       PORTC.OUTTGL = PIN0_bm;          // toggles output
26       _delay_ms(LOCK_PERIOD_MS);       // lock input
27     }
28   }
29 }
```

Om de betreffende poort te benaderen moet aan de functie het adres van de poort worden meegegeven. Pointer `p` wijst daarom naar het adres van een datastructuur `PORT_t`. Bij de aanroep van `button_pressed` op regel 24 staat voor de variabele `PORTD` de adresoperator `&`, zodat het adres van `PORTD` aan de functie wordt meegegeven.

In de functie is `p` een pointer naar de registers van de poort. Het scheidingsteken tussen het veld en een pointer naar een datastructuur is een pijltje `->`. Het ingangsregister `IN` van de poort wordt op regel 10 en regel 12 daarom gevonden met `p->IN`.

Op regel 12 is in plaats van een **`while`** de macrodefinitie `loop_until_bit_is_set` gebruikt. Het headerbestand `sfr_defs.h` bevat naast de definities `bit_is_clear` en `bit_is_set` nog twee macro's `loop_until_bit_is_clear` en `loop_until_bit_is_set`. Deze macro's wachten totdat de bit in het register respectievelijk laag is en hoog is. De uitdrukking:

```
loop_until_bit_is_set(p->IN, bit);
```

is identiek met:

```
while ( ( bit_is_clear(p->IN, bit) ) ;
```

Bij de Xmega-stijl op basis van datastructuren is het maken van algemene functies met parameters, zoals de functie `button_pressed` in code 15.12, relatief eenvoudig.

16

Interrupts

Doelstelling

In dit hoofdstuk leer je wat een interrupt is, hoe het interruptmechanisme werkt en hoe je de externe interrupt en de timeroverflowinterrupt van de Xmega gebruikt.

Onderwerpen

De behandelde onderwerpen zijn:

- Het verschil tussen polling en interrupts.
- Het interruptmechanisme bij de Xmega.
- Het gebruik van de externe interrupts, de Interrupt Service Routine `ISR` en de macro's `sei()` en `cli()`.
- De prioriteits- of interruptniveaus van de interrupts.
- De resetvector en de interruptvectoren.
- Een programma in machinecode en assembly.
- Meten van tijd en de begrippen timer en counter.
- De timer/counters bij de Xmega.
- Het gebruik van de overflowinterrupt van de timer/counter `0` van poort C.
- Een antidenderalgoritme op basis van interrupts.
- De groepsconfiguratie, het groepsmasker, de groepspositie, het bitmasker en de bitpositie.

De voorbeelden demonstreren het gebruik van:

- Led aan- en uitzetten met de externe interrupt `0`.
- Led laten knipperen met de timer/counter `0` met overflowinterrupt.
- Led laten knipperen met de timer/counter `0` zonder interrupt.
- Led laten knipperen met alleen timer/counter `0` van poort E.
- Antidenderalgoritme met externe interrupt en timeroverflowinterrupt.

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 de taak waar het mee bezig is. De microcontroller voert eerst de speciale interruptroutine uit en gaat daarna weer verder met het hoofdprogramma.

Polling is vergelijkbaar met deze situatie. Stel dat je op je studeerkamer bezig bent, dat je vrienden verwacht en dat de bel stuk is en je mobiel is weg. 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 dan niet veel terecht.

De voordelen van het gebruik van interrupts zijn evident. Het hoofdprogramma wordt niet belast met deels overbodige taken, waardoor de tijdplanning of *scheduling* eenvoudiger is. Interrupts zijn ook belangrijk bij vermogensbesparing. De processor kan worden uitgeschakeld om energie te besparen en weer worden aangezet door een interrupt.

Een interrupt is vergelijkbaar met de situatie dat de bel of je mobiel het wel doen. Je doet je gewone werk. 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.

Toch wordt *polling* ook veel gebruikt. Interrupts reageren snel, maar er is vanwege extra overhead meer processortijd nodig om een taak uit te voeren. Bij snelle communicatie met veel gebeurtenissen is *polling* sneller en eenvoudiger. Bij eenvoudige microcontrollers is *polling* een goed alternatief, doordat de mogelijkheden met interrupts dan te beperkt zijn.

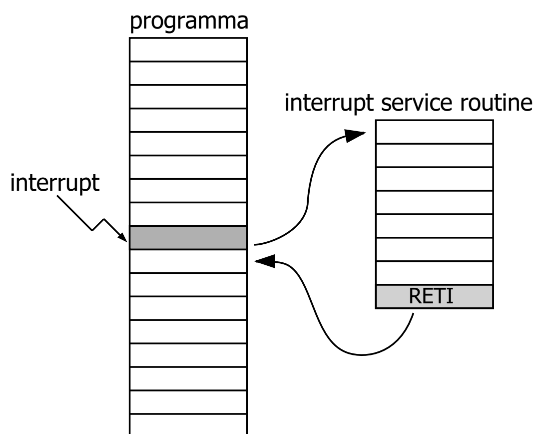
In paragraaf 15.8 en 15.11 is *polling* gebruikt om de status van de drukknop te bepalen. Dit hoofdstuk laat zien hoe een drukknop met een interrupt kan worden uitgelezen en bespreekt het gebruik van een timer en timerinterrupts.

16.1 Het interruptmechanisme

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 op het punt waar het onderbroken was.

Er zijn twee problemen die bij interrupts goed geregeld moeten worden: de ISR moet vanaf elke 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 na afloop van de ISR verder moet gaan, wordt onthouden door het huidige adres op de stack te zetten.



Figuur 16.1 : 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.

De verschillende handelingen, die voor een interrupt nodig zijn, hoeft de C-programmeur niet zelf te beschrijven. De compiler zorgt ervoor dat deze handelingen op de juiste manier worden uitgevoerd, mits de C-programmeur de interrupts op een correcte wijze configureert en het interruptmechanisme van de microcontroller op de juiste wijze instelt.

16.2 De externe interrupts en het interruptmechanisme bij de Xmega

In volgende paragraaf wordt een externe interrupt gebruikt om te demonstreren hoe een interrupt bij de Xmega toegepast wordt.

Iedere poort van de Xmega heeft twee externe interrupts: interrupt `0` en interrupt 1. Iedere pin van de poort kan als interruptingang gedefinieerd worden. De interrupt kan ook door meerdere pinnen getriggerd worden.

De interrupt kan gevoelig zijn voor verschillende signaalniveaus namelijk voor de opgaande flank, de neergaande flank of voor beide flanken en kan ook reageren op een laag signaalniveau.
Het interruptmechanisme van de Xmega kent verschillende niveaus. Een interrupt kan een laag, middelmatig of hoog prioriteitsniveau hebben. Een interrupt met een hoog prioriteitsniveau kan een interrupt met een medium of laag prioriteitsniveau interrumperen en een interrupt met een medium niveau kan een interrupt met een laag niveau interrumperen. Dus net als alle andere interrupts kunnen de externe interrupts ingesteld worden op een van deze drie niveaus.

De interruptniveaus uit tabel 16.1 worden door Atmel *interrupt levels* genoemd. Het begrip *interrupt priority* reserveert Atmel voor de verschillende prioriteiten bij een interrupt van het zelfde *level*. Interrupts met een lager interruptvectornummer hebben een hogere prioriteit.

Dit boek volgt dit onderscheid en gebruiktvanaf hier consequent het begrip interruptniveau in plaats van prioriteitsniveau.

Tabel 16.1 : De interruptniveaus of *interrupt levels* bij de Xmega.

Interrupt niveau	Naam	*Afkorting*[1]	Betekenis
`00`	off	`OFF`	interrupt uit
`01`	low	`LO`	interrupt laag niveau
`10`	medium	`MED`	interrupt middelmatig niveau
`11`	high	`HI`	interrupt hoog niveau

[1] Dit is de afkorting van de naam, zoals deze in de *group configurations* gebruikt wordt.

Om bij de Xmega een interrupt toe te passen, moeten er steeds zes aspecten bekend zijn en op de gewenste wijze ingesteld zijn:

1. Er is een interrupt service routine nodig.
2. De ISR moet de bijbehorende interruptvector hebben.
3. De interrupt moet op de gewenste manier geconfigureerd zijn.
4. De interrupt moet actief zijn op het gewenste interruptniveau.
5. De microcontroller moet gevoelig zijn voor het gewenste interruptniveau.
6. Het globale interruptmechanisme moet aangezet zijn.

Voor het toepassen van de interrupts bij de interfaces en andere perifere blokken heeft de Xmega een groot aantal registers met een scala aan mogelijkheden.

16.3 Een voorbeeld met externe interrupt 0

In deze paragraaf wordt de externe interrupt 0 van poort D gebruikt om te laten zien hoe het interruptmechnanisme van de Xmega en een externe interrupt gebruikt wordt. De schakeling, die nodig is, is het schema uit figuur 15.12 met een drukknop op pin 3 van poort D en een led op pin 0 van poort C. Bij de drukknop wordt een externe pullupweerstand gebruikt en een kleine capaciteit om contactdender tegen te gaan. Er is bewust gekozen om geen interne pullup en geen softwarematig antidenderalgortime te gebruiken om de software code zo eenvoudig mogelijk te houden.

Code 16.1 : Een led aan- en uitzetten met de externe interrupt 0 van poort D.

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  ISR(PORTD_INT0_vect)
5  {
6    PORTC.OUTTGL   = PIN0_bm;
7  }
8
9  int main(void)
10 {
11   PORTC.DIRSET   = PIN0_bm;
12
13   PORTD.INT0MASK = PIN3_bm;              // PD3 interrupt 0
14   PORTD.PIN3CTRL = PORT_ISC_FALLING_gc;  // falling edge
15   PORTD.INTCTRL  = PORT_INT0LVL_LO_gc;   // interrupt 0 low level
16
17   PMIC.CTRL |= PMIC_LOLVLEN_bm;          // set low level interrupts
18   sei();                                 // enable interrupts
19
20   while (1) {
21     asm volatile ("nop");                // do nothing
22   }
23 }
```

In code 16.1 is pin 3 van poort D een ingang. Dit staat nergens in de code. Dat is ook niet nodig, omdat alle pinnen standaard ingang zijn. Natuurlijk mag dit ook expliciet bij de initialisatie aangegeven worden met:
`PORTD.DIRCLR = PIN3_bm;`

Code 16.1 reageert op de externe interrupt 0. Iedere keer als de knop ingedrukt wordt, gaat het ingangssignaal omlaag. Deze neergaande flank zorgt ervoor dat de interrupt service routine wordt uitgevoerd en dat de status van de led verandert. De aspecten uit paragraaf 16.2, die nodig zijn voor een interrupt, zijn alle zes in code 16.1 terug te vinden. Figuur 16.2 toont een kopie van de code met daarbij aangegeven waar deze zes punten in de code staan.

Gedetailleerde bespreking van code 16.1

Uitleg code 16.1 regel 2
`#include <avr/interrupt.h>`

Het headerbestand `avr/interrupt.h` bevat de definities van een aantal macro's die bij interrupts nodig zijn, zoals `sei()` en `ISR()`.

Regel 4
`ISR(PORTD_INT0_vect)`

`ISR()` is de macrodefinitie, die bij AVR-gcc de interrupts afhandelt. De interrupt service routine is een functie met een body en een header met de interruptvector als enige parameter. Het programma gebruikt de interruptvector om de betreffende interruptfunctie te vinden. De body bevat, net als een gewone functie, de code die de routine bij een interrupt uitvoert.

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

ISR(PORTD_INT0_vect)
{
  PORTC.OUTTGL = PIN0_bm;
}

int main(void) {
{
  PORTC.DIRSET   = PIN0_bm;

  PORTD.INT0MASK = PIN3_bm;
  PORTD.PIN3CTRL = PORT_ISC_FALLING_gc;
  PORTD.INTCTRL  = PORT_INT0LVL_LO_gc;

  PMIC.CTRL      = PMIC_LOLVLEN_bm;
  sei();

  while(1) {
    asm volatile ("nop");
  }
}
```


Figuur 16.2 : De zes punten die nodig zijn voor de externe interrupt van code 16.1. Deze zijn te verdelen in drie groepen: de ISR met de interruptvector, de configuratie en het activeren van de interrupt en het instellen van het interruptmechnanisme van de microcontroller.

De macro ISR voert bij een interrupt de code uit de body uit en handelt daarbij het interruptmechanisme op juiste wijze af. Voordat de microcontroller de ISR uitvoert, zet het de waarde van de programcounter op de *stack*. Nadat de code is uitgevoerd, zet de microcontroller de oude waarde terug in de programcounter. Het hoofdprogramma kan dan verder gaan bij de instructie waar het gebleven was.

Regel 13
PORTD.INT0MASK

Iedere poort heeft twee *interrupt-mask*-registers, namelijk: INT0MASK voor interrupt 0 en INT1MASK voor interrupt 1. De ingangen, waarbij de bits van deze maskers hoog zijn, zijn aangesloten op de betreffende interrupt. In dit geval is pin 3 van poort D de interrupt en is bit 3 van INT0MASK dus hoog.

Regel 14
PORTD.PIN3CTRL

De interrupt kent diverse flankgevoeligheden voor de ingangssignalen. Deze gevoeligheid wordt ingesteld met de drie ISC-bits uit het PINnCTRL-register van de betreffende aansluiting. Tabel 16.2 geeft de betekenis van deze *input sense configuration*-bits. De toewijzing op regel 14 maakt de interrupt gevoelig voor de neergaande flank van het ingangssignaal.

Tabel 16.2 : De *input sense configuration*-bits.

ISC[2:0]	Naam groepsconfiguratie	Gevoeligheid van het ingangssignaal
000	PORT_ISC_BOTHEDGES_gc	gevoelig voor beide flanken
001	PORT_ISC_RISING_gc	gevoelig voor opgaande flank
010	PORT_ISC_FALLING_gc	gevoelig voor neergaande flank
011	PORT_ISC_LEVEL_gc	gevoelig voor een laag niveau
100		niet gebruikt
101		niet gebruikt
110		niet gebruikt
100	PORT_ISC_INPUT_DISABLE_gc	digitale ingangsbuffer uitschakelen

Uitleg code 16.1 regel 15
`PORTD.INTCTRL`

De interruptfuncties van de verschillende interfaces staan standaard uit. Iedere interrupt, die het programma gebruikt, moet expliciet worden aangezet. Bij de Xmega gaat dat tegelijkertijd met het instellen van het interruptniveau van de interrupt. In tabel 16.1 staan de verschillende interruptniveaus.
Het `INTCTRL`-register van iedere poort bevat de twee groepen met twee bits, waarmee de externe interrupts van de poort aangezet worden en ingesteld. De definitie `PORT_INT0LVL_LO_gc` is een zogenoemde groepsconfiguratie die de twee bits voor de externe interrupt `0` instelt op het lage niveau. In paragraaf 16.7 wordt het gebruik van groepsconfiguraties en groepsmaskers verder toegelicht.

Regel 17
`PMIC.CTRL`

De `PMIC`, *programmable multilevel interrupt controller* heeft een controlregister `CTRL` dat instelt voor welke interruptniveaus de microcontroller gevoelig is. Het bitmasker `PMIC_LOLVLEN_bm` zet de gevoeligheid voor het lage niveau aan.

Regel 18
`sei()`

Met de globale interrupt bit `I` uit het statusregister `SREG` wordt het hele interruptmechanisme aan- en uitgezet. Als deze bit laag is, is de microcontroller ongevoelig voor alle interrupts. Is de bit hoog dan is het interruptmechanisme actief. De AVR GNU C-compiler gebruikt de functie `cli()`, *clear interrupt bit*, om deze bit laag te maken en de functie `sei()`, *set interrupt bit* om deze bit hoog te maken.

Regel 21
`asm volatile ("nop");`

De opdracht `asm` geeft aan dat er een assembler instructie volgt. In dit geval is dat de no-operating instructie `nop`. Het sleutelwoord **`volatile`**, zorgt ervoor dat de compiler deze opdracht niet wegoptimaliseert. Het hoofdprogramma bestaat uit een oneindige **`while`**, die dus niets doet. Regel 21 met de `nop` mag ook weggelaten worden. De `nop`-instructie is toegevoegd, omdat dit handig is bij het debuggen en het simuleren. Bij deze regel kan dan een breekpunt, *breakpoint*, worden geplaatst.

Het interruptmechnanisme nader bekeken

Een interruptroutine is geen gewone functie. Het is een functie, die *nooit* in de programmacode wordt aangeroepen. De verschillende hardware-componenten van de microcontroller genereren de interruptsignalen waar het programma op moet reageren. Bij een normale functieaanroep is bekend waar de functie in het geheugen staat en is de positie bekend vanaf waar de functie is aangeroepen. Na afloop van de functie gaat het programma automatisch terug naar de plaats waar de functie werd aangeroepen.
Bij een interrupt is een mechanisme nodig dat de juiste interruptfunctie vindt en een mechanisme nodig om het programma na afloop te laten terugkeren naar de plaats waar het gebleven was. Voor het eerste mechanisme is de interruptvector nodig en voor het tweede mechanisme wordt de inhoud van de programcounter op de *stack* gezet.

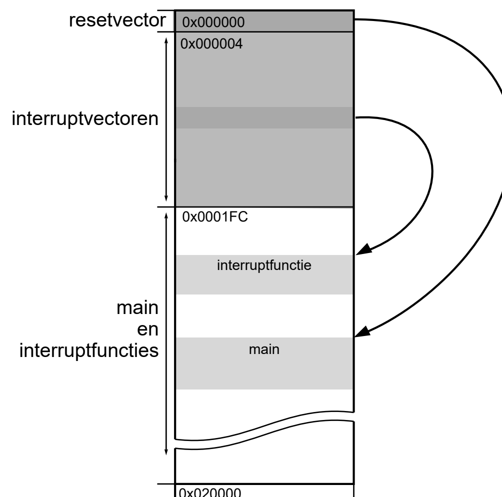
De interruptvector is in feite een nummer uit een opzoektabel met verwijzingen naar de verschillende interruptroutines. In tabel 16.3 staat een klein deel van de interruptvectoren van de Xmega256a3u. Deze vectoren zijn in het headerbestand `iox256a3u.h` gedefinieerd. Aangezien ieder type microcontroller zijn eigen features kent, is het aantal interruptvectoren voor elke Xmega anders of worden er andere nummers gebruikt.

Tabel 16.3 : De resetvector en de interruptvectoren van de Xmega256a3u.

Nummer	Adres (in bytes)	Naam bij AVR-gcc	Omschrijving
	0x0000	*resetvector*	*voor power-on-reset, brownout, watchdog, PDI*
1	0x0004	OSC_OSCF_vect	oscillator failure interrupt
2	0x0008	PORTC_INT0_vect	port C external interrupt 0
3	0x000C	PORTC_INT1_vect	port C external interrupt 1
4	0x0010	PORTR_INT0_vect	port R external interrupt 0
5	0x0014	PORTR_INT1_vect	port R external interrupt 1
6	0x0018	DMA_CH0_vect	DMA channel 0 interrupt
		⋮	
64	0x0100	PORTD_INT0_vect	port D external interrupt 0
65	0x0104	PORTD_INT1_vect	port D external interrupt 1
		⋮	
119	0x01DC	USARTF0_RXC_vect	UART F0 reception complete interrupt
120	0x01E0	USARTF0_DRE_vect	UART F0 data register empty interrupt
121	0x01E4	USARTF0_TXC_vect	UART F0 transmission complete interrupt
125	0x01F4	USB_BUSEVENT_vect	USB bus event interrupt
126	0x01F8	USB_TRNCOMPL_vect	USB transmission complete interrupt

De tweede kolom van tabel 16.3 geeft het adres waar de microcontroller automatisch naar toegaat bij een interrupt. Bij een externe interrupt 0 van poort D springt het programma dus automatisch naar adres `0x0100`.

De opzoektabel staat bij de Xmega256a3u aan het begin van het programmageheugen en voor iedere vector zijn vier bytes beschikbaar. Dat is genoeg plaats voor een sprongopdracht naar de betreffende interruptfunctie. In figuur 16.3 is het applicatiedeel van het flashgeheugen getekend met de adressen van de resetvector en de interruptvectoren. Op het adres van de resetvector staat een sprongopdracht naar functie `main` en op de adressen van de interruptvectoren staan sprongopdrachten naar de interruptfuncties.



Figuur 16.3 : Het applicatiedeel van het flashgeheugen met de interruptvectoren. Op het adres van de resetvector staat een sprongopdracht naar functie `main`. Op het adres van de interruptvector staat een sprongopdracht naar de interruptfunctie.

In werkelijkheid is het een beetje complexer. In figuur 16.4 is de C-code 16.1 vertaald naar machinecode en assembly. Machinecode of machinetaal is de taal die

```
geheugenadres   machinecode   assembly          commentaar
00000000:       fd c0         RJMP   .+506      ; 0x1FC
00000002:       00 00         NOP
00000004:       05 c1         RJMP   .+522      ; 0x210
00000006:       00 00         NOP
00000008:       03 c1         RJMP   .+518      ; 0x210
0000000a:       00 00         NOP
   :       resetvector met ...
00000100:       88 c0         RJMP   .+272      ; 0x212
00000102:       00 00         NOP
   :       ... 126 interruptvectoren
000001f8:       0b c0         RJMP   .+22       ; 0x210
000001fa:       00 00         NOP
000001fc:       11 24         EOR    r1, r1
   :b  initialisatieroutines door gcc toegevoegd
0000020c:       1a d0         RCALL  .+52       ; 0x242
0000020e:       2c c0         RJMP   .+88       ; 0x268
00000210:       f7 ce         RJMP   .-530      ; 0x0
00000212:       1f 92         PUSH   r1
   :3                  ISR
00000240:       18 95         RETI
00000242:       81 e0         LDI    r24,0x1
   :d                  hoofdprogramma
00000264:       00 00         NOP
00000266:       fe cf         RJMP   .-4        ; 0x264
00000268:       f8 94         CLI
0000026a:       ff cf         RJMP   .-2        ; 0x26a
```

Figuur 16.4 : **De machinecode en assembly, die hoort bij code 16.1 met de acties bij het starten van de microcontroller en bij een aanroep van de ISR.** In de figuur zijn drie scenario's getekend:

- Het programma begint bij het adres van de de resetvector en springt [a] naar een aantal initialisatieblok [b], dat `gcc` heeft toegevoegd. Vervolgens springt [c] het naar hoofdprogramma en voert dit programma 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.
- Op de adressen van de interruptvectoren die niet gebruikt worden, staat een sprongopdracht [A] naar adres `0x210`, waar een sprongopdracht [B] staat naar de adreslocatie van de resetvector.

De tekst uit figuur 16.4 is gebaseerd op het lss-bestand dat Atmel Studio bij de compilatie van code 16.1 genereert.

de processor begrijpt. Anders gezegd, dat is het programma in enen en nullen. Assembly of assembleertaal is een symbolische notatie van de machinecode, die voor de mens leesbaar is.

Op adres `0x0` staat bij de machinecode `0xFDC0`. Dat is een relatieve sprongopdracht `RJMP .+506`. De instructie `RJMP` is slechts twee bytes groot, daarom staat er op adres `0x02` nog een `NOP`-opdracht. De `RJMP` springt vanaf positie `0x02` precies 506 bytes verder naar adres `0x1FC`. Op dit adres begint een initialisatie die de compiler heeft toegevoegd. Op adres `0x20C` springt het programma verder naar het hoofdprogramma dat op adres `0x242` begint.

Op adres `0x100`, de locatie van de interruptvector van de externe interrupt `0` van poort D, staat een sprongopdracht naar de bijbehorende interruptroutine, die op adres `0x212` staat. Bij een interrupt zet de microcontroller de waarde van de pro-

grammateller op de *stack*, voert de feitelijke interruptfunctie uit en zet de waarde van de programmateller weer terug. Op adres `0x240` sluit `RETI` de interruptfunctie af en de microcontroller gaat verder waar deze in het hoofdprogramma gebleven was.

Atmel Studio kan de code disassembleren door in de debugmode de *disassembler* aan te roepen. Dit is het compilatieresultaat in 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.

16.4 Timer/counters

Tijd is bij microcontrollers belangrijk. Embedded systemen hebben vaak een beperkte taak, kennen een beperkt aantal instellingen, maar moeten wel op gezette tijden de juiste functies uitvoeren. Een 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 iedere microseconde de temperatuur te controleren.

De methode om met een digitaal systeem tijd te meten is klokpulsen tellen. Digitaal is de verstreken tijd t gelijk aan het aantal getelde klokpulsen m vermenigvuldigd met de klokperiode van het systeem T_{cpu}.

$$t = mT_{\text{cpu}} \tag{16.1}$$

Een 16-bits teller, die telt met een klokfrequentie van 32768 Hz, geeft elke twee seconde een overflow.

Microcontrollers hebben meestal meerdere tellers. Omdat een teller gebruikt kan worden om te tellen en om tijd te meten, wordt een teller de ene keer een *counter* en de andere keer een *timer* genoemd. Deze timer/counters bevatten naast een gewone digitale teller ook allerlei logica waarmee signalen gemeten en gegenereerd kunnen worden.

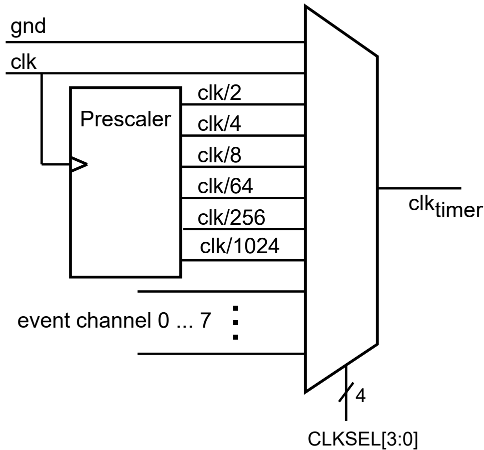
Timer/counters bij de Xmega

De Xmega heeft bij iedere digitale poort één of twee 16-bits timer/counters. De Xmega256a3u heeft vier digitale poorten. De poorten C, D en E hebben beide een timer/counter `0` en een timer/counter 1 en poort F heeft alleen een timer/-counter `0`. De namen van deze tellers zijn: `TCC0` — *`timer/counter c0`* —, `TCC1`, `TCD0`, `TCD1`, `TCE0`, `TCE1` en `TCF0`.

Naast de zeven genoemde 16-bits timer/counters heeft de Xmega256a3u ook nog een *real time counter* voor het maken van een *real time clock* met behulp van een laagfrequent kristaloscillator.

De timer/counter `0` en timer/counter 1 zijn grotendeels identiek en kennen beide diverse modi. Deze timer/counters kunnen gebruikt worden om:

- tijd te meten;
- de frequentie, de periodetijd en de duty cycle van een signaal te meten;
- een blokvormig signaal met een bepaalde frequentie te genereren;
- verschillende soorten PWM-signalen, pulsbreedtegemoduleerde signalen, te genereren.



Figuur 16.5 : De prescaler voor de timers. De klok van de timer kan op zeven gedeelde kloksignalen worden aangesloten. Als de klok met de referentie, GND, verbonden is, is de timer uit. De klok kan ook verbonden worden met een van de acht *event channels*.

Tabel 16.4 : De selectiebits voor het instellen van het kloksignaal van de timers.

CLKSEL[3:0]	Groepsconfiguratie	Betekenis
0000	TC_CLKSEL_OFF_gc	Timer is uit
0001	TC_CLKSEL_DIV1_gc	clk_{cpu}
0010	TC_CLKSEL_DIV2_gc	$clk_{cpu}/2$
0011	TC_CLKSEL_DIV4_gc	$clk_{cpu}/4$
0100	TC_CLKSEL_DIV8_gc	$clk_{cpu}/8$
0101	TC_CLKSEL_DIV64_gc	$clk_{cpu}/64$
0110	TC_CLKSEL_DIV256_gc	$clk_{cpu}/256$
0111	TC_CLKSEL_DIV1024_gc	$clk_{cpu}/1024$
1nnn	TC_CLKSEL_EVCH*n*_gc	Event channel $0 \cdots 7$

Dit hoofdstuk gebruikt de timer/counters alleen om tijd te meten. Het meten aan signalen en het genereren van PWM-signalen komt in de latere hoofdstukken aan bod.

Een 16-bits teller kan maximaal tot 65535 tellen. Bij een systeemklok van 2 MHz is de maximale meettijd korter dan 33 ms en bij een systeemklok van 32 MHz is dat net iets meer dan 2 ms. Om langere perioden te kunnen meten, hebben de tellers bij een microcontroller altijd een zogenoemde *prescaler* of klokdeler. Een prescaler genereert uit de systeemklok verschillende gedeelde kloksignalen, die als klok voor de teller kunnen functioneren. In figuur 16.5 is de prescaler van de Xmega getekend en in tabel 16.4 staat een overzicht met de verschillende instelmogelijkheden.

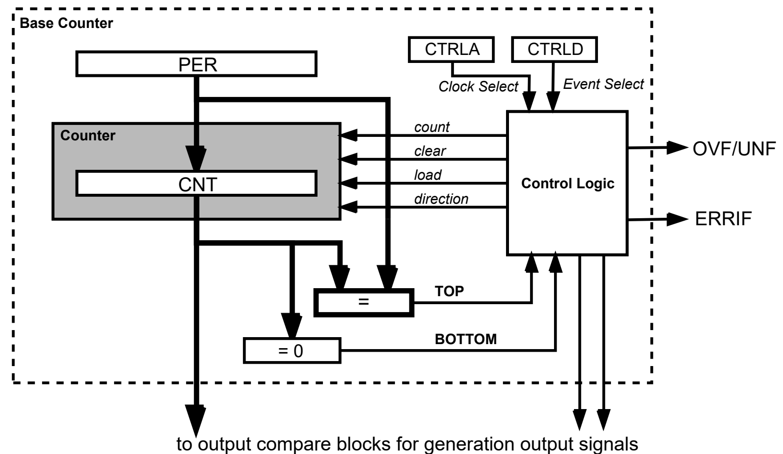
Met een prescaling hangt de tijd af van het aantal gedeelde klokslagen dat geteld wordt. Als N de prescaling is en m klokslagen geteld worden, komt dat overeen met Nm klokpulsen van de systeemklok. De verstreken tijd t is dan:

Bij de timer/counter van de Xmega stelt register PER het aantal klokslagen m in dat geteld wordt.
Hierbij is m gelijk aan PER + 1.

$$t = NmT_{\text{cpu}} = \frac{Nm}{f_{\text{cpu}}} \tag{16.2}$$

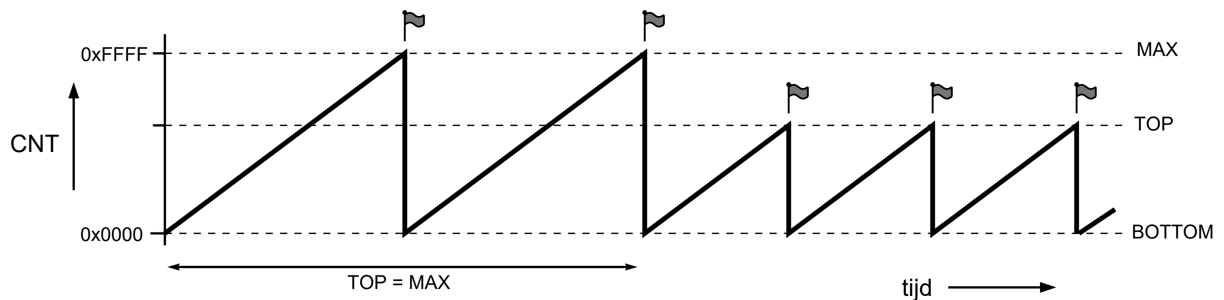
De maximale prescaling van de 16-bits timer/counters is 1024. Voor een frequentie van 2 MHz is de maximale tijd die gemeten kan worden bijna 33,6 s en voor een frequentie van 32 MHz is dat bijna 2,1 s.

Het blokschema van figuur 16.6 geeft het basisgedeelte van een timer/counter van de Xmega met de feitelijke teller. De inhoud van de teller is benaderbaar als register CNT. De 16-bits teller telt van 0 tot de waarde van het getal dat in het PER



Figuur 16.6 : Het basisgedeelte van een timer/counter van de Xmega met de teller. Het uitgangsdeel voor de generatie van PWM-signalen is niet getekend. De feitelijke teller is grijs getekend. De controllogica stuurt de teller aan en genereert de interruptsignalen.

register staat. Daarna begint de teller weer bij `0`. De maximale waarde van `PER` is `0xFFFF`. Signaal `BOTTOM` is actief wanneer de waarde van de teller gelijk is aan `0` en signaal `TOP` is actief als deze gelijk is aan de waarde in register `PER`. Het blok met de besturingslogica gebruikt de signalen `BOTTOM` en `TOP` en de status van de in het schema getekende registers of de teller telt, een nieuwe waarde inlaadt, op `0` begint en of de teller optelt of aftelt. Afhankelijk van de instelling kan de besturingslogica ook twee interruptsignalen genereren als `BOTTOM` of `TOP` actief zijn.



Figuur 16.7 : De waarde van de teller `CNT` als functie van de tijd in de normale modus. De teller telt van `BOTTOM` naar `TOP`. De waarde van `BOTTOM` is `0` en die van `TOP` kan worden ingesteld. Als de waarde van `TOP` bereikt is, wordt de teller weer `BOTTOM` en is er een interrupt. Dat laatste is steeds aangegeven met een vlaggetje.

In figuur 16.7 staat de waarde van de teller als functie van de tijd. De waarde waarbij de signalen `TOP` en `BOTTOM` actief zijn, worden ook `TOP` en `BOTTOM` genoemd. Iedere keer als de teller bij `TOP` is, springt deze terug naar `BOTTOM`. Mits de overflow-interrupt van de timer/counter ingesteld is, zal er dan ook een interrupt gegeven worden.

16.5 Een tijdvertraging maken met een timer/counter

Met een timer/counter kan tijd worden gemeten en dus ook gebruikt worden om een tijdvertraging te maken. In code 16.2 staat een programma dat een led laat knipperen met behulp van timer/counter 0 van poort E. De schakeling die nodig is, is die uit figuur 15.1.

De timer/counter wordt gebruikt in de normale modus. Voor het knipperen van de led is een tijdvertraging nodig van 250 ms. De klokfrequentie is 2 MHz. De tijdvertraging komt dan overeen met 500000 klokslagen. Voor een 16-bits teller moet de prescaling minimaal 8 zijn. Tabel 16.5 geeft het aantal gedeelde klokslagen m als functie van de prescaling voor de frequenties 2 en 32 MHz. Dit aantal klokslagen is de waarde die het register moet krijgen, opdat — bij deze prescaling — de tijdvertraging 250 ms is.

Tabel 16.5 : De vertraging hangt af van register PER, van de prescaling N en van de klokfrequentie. Deze tabel toont de meest geschikte waarde van PER+1 voor een tijdvertraging van 250 ms.

	2 MHz		**32 MHz**	
prescaling N	m = PER+1	**tijd (t)**	m = PER+1	**tijd (t)**
1				
2				
4				
8	62500	0,25		
64	7812	0,249984		
256	1953	0,249984	31250	0,25
1024	488	0,249856	7812	0,249984

Bij de normale modus is het aantal klokslagen m gelijk aan PER + 1. De periodetijd t is dan gelijk aan:

$$t = \frac{N\,(\text{PER}+1)}{f_{cpu}}$$

De nauwkeurigheid van de tijdvertraging is in het algemeen groter bij een lage prescaling. In tabel 16.5 is ook de tijdvertraging t vermeld voor de betreffende prescaling en de bijbehorende waarde van PER. Bij 16-bits tellers is de afrondingsfout in de meeste gevallen relatief klein. Bij 8-bits tellers kan de afrondingsfout wel te groot zijn. De applicatie bepaalt of een afrondingsfout acceptabel is. In dit geval —- het laten knipperen van de led — maakt het niet uit. Een fout van 1% kan dan bijvoorbeeld acceptabel zijn.

In code 16.2 is timer/counter 0 van poort E ingesteld op de normale modus met een prescaling van 8 en de waarde van PER gelijk aan 62499. De overflowinterrupt is geactiveerd met het lage interruptniveau en de interruptfunctie verandert de aansluiting 0 van poort C. Iedere keer als de waarde van de teller gelijk is aan die van PER is er een overflowinterrupt en wordt de interruptfunctie uitgevoerd, waardoor de led van status verandert.

Gedetailleerde bespreking van code 16.2

Uitleg code 16.2 regel 6
`TCE0_OVF_vect`

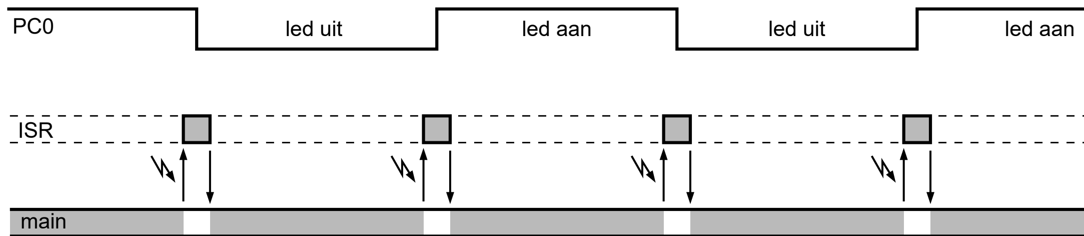
Vector `TCE0_OVF_vect` is de interruptvector voor de overflow van timer/counter 0 van poort E. Deze vector bevat de naam van de timer/counter `TCE0` en de naam van het soort interrupt `OVF`.

Code 16.2 : Led laten knipperen met timer/counter 0 van poort E.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6  ISR(TCE0_OVF_vect)
7  {
8    PORTC.OUTTGL = PIN0_bm;
9  }
10
11 int main(void)
12 {
13   PORTC.DIRSET   = PIN0_bm;
14
15   TCE0.CTRLB     = TC_WGMODE_NORMAL_gc;  // Normal mode
16   TCE0.CTRLA     = TC_CLKSEL_DIV8_gc;    // prescaling 8
17   TCE0.INTCTRLA  = TC_OVFINTLVL_LO_gc;   // enable overflow interrupt low level
18   TCE0.PER       = 62499;                // t = N*(PER+1)/FCPU = 8*62500/2000000 = 0.25 s
19
20   PMIC.CTRL     |= PMIC_LOLVLEN_bm;      // set low level interrupts
21   sei();                                 // enable interrupts
22
23   while (1) {
24     asm volatile ("nop");
25   }
26 }
```

Regel 15 `TCE0.CTRLB`	De drie minst significante bits van register `CTRLB` stellen de modus van de timer-counter in. Voor de normale modus moeten deze bits laag zijn. De groepsconfiguratie voor deze modus is `TC_WGMODE_NORMAL_gc`. Omdat standaard alle bits laag zijn, mag regel 15 — in dit geval — ook worden weggelaten.
Regel 16 `TCE0.CTRLA`	De vier minst significante bits van register `CTRLA` stellen de klok van de teller in. In tabel 16.4 staat de groepsconfiguratie. In dit geval is de prescaling 8 en is `TC_CLKSEL_DIV8_gc` toegekend aan `CTRLA`.
Regel 17 `TCE0.INTCTRLA`	Bit 1 en 0 uit register `INTCTRLA` stellen de overflowinterrupt in. Net als alle interrupts kan deze interrupt vier verschillende niveaus hebben, namelijk: uit, laag, middelmatig en hoog. Tabel 16.1 geeft de groepsconfiguratie. In dit geval wordt aan `CTRLINTA` de waarde `TC_OVFINTLVL_LO_gc` toegekend. Hierbij betekent `TC` timer/-counter en staat `OVFINTLVL` voor *overflow interrupt level*.
Regel 18 `TCE0.PER`	Register `PER` is een 16-bits register. De hoge en lage byte zijn ook apart benaderbaar als `PERH` en `PERL`. Register `PER` bevat de waarde `TOP` van de teller. Als de teller gelijk is aan `TOP`, springt de teller terug naar 0 en is er een interrupt, mits de overflowinterrupt actief staat.

De toekenning op regel 20 maakt de microcontroller gevoelig voor interrupts met het lage interruptniveau en regel 21 zet het globale interruptmechanisme aan. Het hoofdprogramma doet in de oneindige lus niets. Alleen op het moment dat er een overflowinterrupt is, wordt de interruptfunctie uitgevoerd en verandert de status van de led.



Figuur 16.8 : De tijdlijnen van het hoofdprogramma `main` en de interruptfunctie `ISR`. Iedere 250 ms is er een interrupt, waardoor het hoofdprogramma stopt en de interrupt wordt uitgevoerd.

Feitelijk bestaat de code dus uit twee programma's: een hoofdprogramma dat niets doet en een interruptfunctie die de led laat knipperen. In figuur 16.8 zijn de tijdlijnen van het hoofdprogramma en de interruptfunctie getekend.

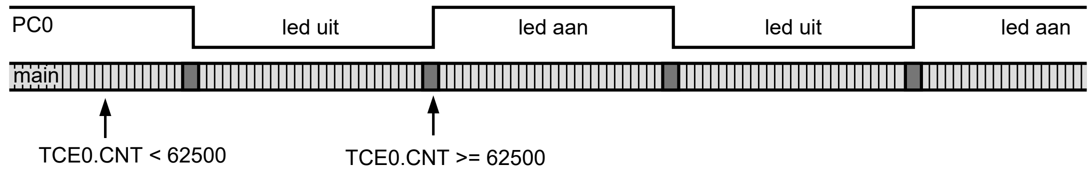
Een tijdvertraging met een timer/counter zonder interrupt

Met een timer/counter kan ook een tijdvertraging gemaakt worden zonder het interruptmechanisme. Code 16.3 beschrijft een teller die tot de maximale waarde telt met een prescaling van 8. Op regel 13 wordt in de oneindige **`while`**-lus getest of de waarde van de teller groter is dan 62500. Als dat het geval is, verandert de status van uitgang `PC0` en wordt de teller nul gemaakt.

Code 16.3 : Led laten knipperen met `TCE0` zonder interrupt.

```
 1  #define  F_CPU 2000000UL
 2
 3  #include <avr/io.h>
 4
 5  int main(void)
 6  {
 7    PORTC.DIRSET  = PIN0_bm;
 8
 9    TCE0.CTRLA    = TC_CLKSEL_DIV8_gc;    // prescaling 8
10    TCE0.PER      = 0XFFFF;               // maximal value
11
12    while (1) {
13      if ( TCE0.CNT >= 62500 ) {          // 8*62500/2000000 = 0.25 s
14        PORTC.OUTTGL = PIN0_bm;
15        TCE0.CNT = 0;
16      }
17      asm volatile ("nop");               // do nothing
18    }
19  }
```

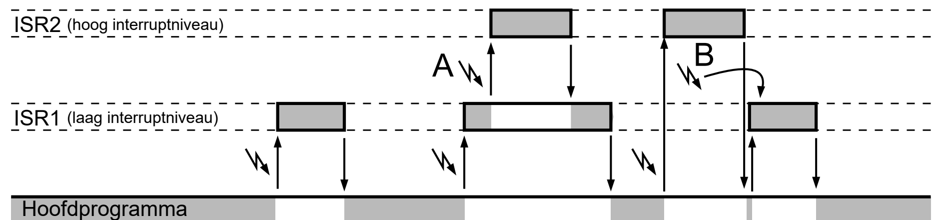
Het verschil met de interruptmethode van code 16.2 is dat nu het hoofdprogramma voortdurend bezig is met het testen `CNT`. Het programma *pollt* continue de waarde van de teller. Als de test negatief is, wordt er 11 klokslagen later weer een test uitgevoerd. Slechts één keer per 250 ms, oftewel één keer per 500000 klokslagen, is de test positief.
De tijdlijn uit figuur 16.9 laat zien dat dit programma continue bezig is met het testen van de teller. Bij *polling* is dit altijd het geval. Zeker bij een programma dat



Figuur 16.9 : De tijdlijn van code 16.3. Het hoofdprogramma vergelijkt voortdurend de waarde van `CNT` met 62500. Meestal is het resultaat negatief. Slechts één keer per 250 ms is `TCE0.CNT` groter.

meerdere taken uitvoert, kan de tijdplanning of *scheduling* lastig zijn. Interruptfuncties ontlasten het hoofdprogramma en maken dit eenvoudiger en overzichtelijker.

Daartegenover staat dat het interruptniveau van de interruptfuncties goed gekozen moet worden. Als er meerdere interrupts zijn, hangt de volgorde af van de prioriteit. Interruptfuncties met een gelijk of lager interruptniveau worden na elkaar uitgevoerd. Sommige interruptfuncties mogen niet onderbroken worden en moeten hun taak ononderbroken uitvoeren. Figuur 16.10 laat het tijdsgedrag zien bij een programma met twee interruptfuncties. De interruptfunctie met het hoge interruptniveau wordt niet onderbroken. De interrupt met het lagere interruptniveau wordt uitgevoerd nadat de interrupt met het hogere interruptniveau is afgerond.



Figuur 16.10 : Het tijdsgedrag bij een programma met twee interruptfuncties. Bij A interrumpeert `ISR2` de interruptfunctie `ISR1`. Bij B wordt de interrupt van `ISR1` niet direct uitgevoerd, omdat `ISR1` een lager interruptniveau heeft. `ISR2` wordt afgerond en daarna wordt `ISR1` uitgevoerd.

Met name voor programma's, waarbij de interrupts zeer frequent voorkomen, kan het tijdsgedrag zeer complex worden. Dit wordt nog versterkt doordat voor het uitvoeren van een interrupt extra handelingen nodig zijn. Zo zijn er voor de interruptfunctie uit code 16.2 bijvoorbeeld meer dan 50 klokslagen nodig.

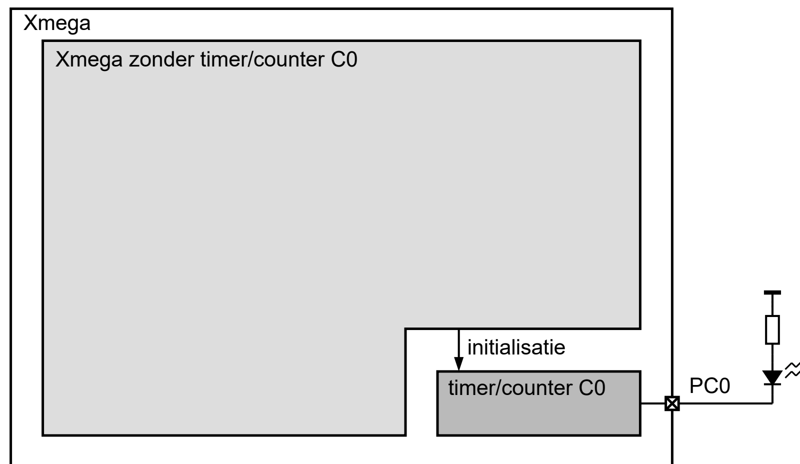
Het knipperen van een led met een timer/counter in de frequentiemodus

Het laten knipperen van een led kan nog efficiënter gedaan worden door een timer/counter in de frequentiemodus of in één van de PWM-modi te gebruiken. In code 16.4 is op regel 9 de timer/counter 0 van poort C ingesteld op de frequentiemodus en is de uitgang van de teller verbonden met uitgang `PC0`. De teller telt nu tot de waarde van het register `CCA`. Als deze waarde bereikt is, wordt de teller weer nul en klapt de waarde van de uitgang om.

Code 16.4 : Led laten knipperen met alleen `TCC0`.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4
5  int main(void)
6  {
7    PORTC.DIRSET  = PIN0_bm;
8
9    TCC0.CTRLB    = TC0_CCAEN_bm | TC_WGMODE_FRQ_gc;
10   TCC0.CTRLA    = TC_CLKSEL_DIV8_gc;
11   TCC0.CCA      = 62500;
12
13   while (1) {
14     asm volatile ("nop");   // do nothing
15   }
16 }
```

De frequentiemodus komt in paragraaf 22.2 aan de orde bij de bespreking van de verschillende PWM-mogelijkheden. Dit voorbeeld is hier toegevoegd om te laten zien dat het mogelijk is een led te laten knipperen zonder interrupts en met een hoofdprogramma dat niets doet. Bij deze aanpak wordt de timer/counter gebruikt om de led aan te sturen en blijft de rest van de microcontroller beschikbaar voor andere activiteiten. Het microcontrollerprogramma is — afgezien van de initialisatie — geen tijd kwijt voor het laten knipperen van de led. In figuur 16.11 is dit gevisualiseerd.



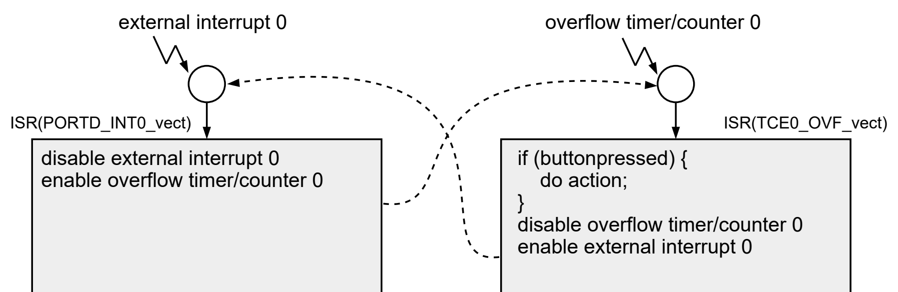
Figuur 16.11 : De timer/counter `0` van poort C stuurt de led aan. De rest van de Xmega is beschikbaar voor andere activiteiten.

Afhankelijk van het soort applicatie en de complexiteit van het tijdsgedrag zal er de ene keer een oplossing worden gebruikt met een interrupt en de andere keer een oplossing op basis van *polling* of voert een van de interfaces van de microcontroller de taak autonoom uit.

16.6 Een antidenderalgoritme met een externe interrupt en TCE0

In hoofdstuk 15 is het denderprobleem van een schakelaar of drukknop met een hardware- en softwarematige methode opgelost. Er zijn vele manieren om dat te doen. Hier wordt nog een oplossing gegeven die gebruikt maakt van een externe interrupt en de overflow van een timer/counter.

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 wachten kan prima met een timer/counter worden gedaan. Er zijn dan twee interruptfuncties nodig : één voor de externe interrupt en één voor de timer.



Figuur 16.12 : **Het antidenderalgoritme.** Aanvankelijk is alleen de `ISR(PORTD_INT0_vect)` actief. Als er op de knop wordt gedrukt, wordt `ISR(TCE0_OVF_vect)` ingeschakeld en `ISR(PORTD_INT0_vect)` uitgeschakeld. Na de timer `0` interrupt wordt `ISR(TCE0_OVF_vect)` uitgeschakeld en `ISR(PORTD_INT0_vect)` weer ingeschakeld.

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/counter start. De interruptfunctie van de timer/counter wordt na enige tijd actief en kijkt of de knop nog steeds ingedrukt is. Als dat zo is, wordt de gewenste activiteit uitgevoerd. Vervolgens wordt de timer/counter uitgezet en de externe interrupt weer aangezet. Figuur 16.12 geeft dit algoritme met de twee interrupt service routines weer.

De interruptfunctie voor de externe interrupt schakelt zichzelf uit. Dit is gedaan om te voorkomen dat deze ISR weer opnieuw aangeroepen wordt. Op deze manier wordt alleen de eerste neergaande flank gedetecteerd. De timer interrupt schakelt zichzelf eveneens uit. Dit zorgt er voor dat de timer interrupt alleen aangeroepen wordt na het indrukken van de knop.

In code 16.5 staat het programma dat hoort bij het antidenderalgoritme van figuur 16.12. Timer/counter `E0` is ingesteld op de normale modus. Bij de initialisatie krijgt op regel 37 register `PER` van timer/counter `E0` de waarde `PERIOD_TIMER`. Deze macrodefinitie staat op regel 7 en berekent de periodetijd `PER` uit de klokfrequentie, de prescaling en de gewenste antidendertijd. De timer staat na de initialisatie uit: er is geen klok aangesloten en de interruptoverflow is eveneens uit. De registers `CTRLA` en `INTCTRLA` ontbreken bij initialisatie. Alle bits van deze registers zijn daarom nog laag. De externe interrupt `0` van poort D is aangesloten op pin 3, is gevoelig voor de neergaande flank en heeft een laag interruptniveau.

Code 16.5 : Antidenderalgoritme met een externe interrupt en een timer-overflow interrupt.

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  #define F_CPU              2000000UL
5  #define DEBOUNCE_PERIOD_MS  2.0
6  #define PRESCALING          8
7  #define PERIOD_TIMER        ( (uint16_t) \
8                                (( (double) (DEBOUNCE_PERIOD_MS) * 0.001 * (F_CPU)) / (PRESCALING)) )
9
10 ISR(PORTD_INT0_vect)
11 {
12   PORTD.INTCTRL  = (PORTD.INTCTRL & ~PORT_INT0LVL_gm) | PORT_INT0LVL_OFF_gc;  // ext. interrupt 0 off
13   TCE0.CNT       = 0;                                                         // reset timer/counter
14   TCE0.INTCTRLA  = (TCE0.INTCTRLA & ~TC0_OVFINTLVL_gm) | TC_OVFINTLVL_LO_gc;  // timer interrupt on
15   TCE0.CTRLA     = TC_CLKSEL_DIV8_gc;                                         // start timer/counter
16 }
17
18 ISR(TCE0_OVF_vect)
19 {
20   if ( bit_is_clear(PORTD.IN, PIN3_bp) ) {
21     PORTF.OUTTGL   = PIN1_bm;
22   }
23   TCE0.CTRLA     = TC_CLKSEL_OFF_gc;                                          // stop timer/counter
24   TCE0.INTCTRLA  = (TCE0.INTCTRLA & ~TC0_OVFINTLVL_gm) | TC_OVFINTLVL_OFF_gc; // timer interrupt off
25   PORTD.INTCTRL  = (PORTD.INTCTRL & ~PORT_INT0LVL_gm) | PORT_INT0LVL_LO_gc;   // ext. interrupt 0 on
26 }
27
28 int main(void)
29 {
30   PORTF.DIRSET    = PIN1_bm;
31
32   PORTD.INT0MASK  = PIN3_bm;               // PD3 interrupt 0
33   PORTD.PIN3CTRL  = PORT_ISC_FALLING_gc;   // falling edge
34   PORTD.INTCTRL  |= PORT_INT0LVL_LO_gc;    // interrupt 0 low level
35
36   TCE0.CTRLB      = TC_WGMODE_NORMAL_gc;   // normal mode
37   TCE0.PER        = PERIOD_TIMER;
38
39   PMIC.CTRL |= PMIC_LOLVLEN_bm;            // set low level interrupts
40   sei();                                   // enable interrupts
41
42   while (1) {
43     asm volatile ("nop");                 // do nothing
44   }
45 }
```

Na een druk op de knop start de interruptfunctie van de externe interrupt `0`. Deze functie zet eerst op regel 12 de externe interrupt uit en maakt daarna achtereenvolgens register `CNT` nul, geeft de interrupt van de timer het lage interruptniveau en zet tenslotte de teller aan door de prescaling 8 te maken.

De interruptfunctie voor de externe interrupt 0 wordt afgesloten en het programma blijft in de oneindige **while**-lus totdat na 2,0 ms de overflow van timer/-counter 0 optreedt en de interruptfunctie van de timer/counter wordt uitgevoerd.

De interruptfunctie van de timer/counter 0 test op regel 20 of de knop nog steeds ingedrukt is. Als dat het geval is, verandert de status van de led. Vanaf regel 23 zet de functie achtereenvolgens de interrupt van de timer uit, de timer zelf uit en zet de externe interrupt 0 weer aan door het interruptniveau weer laag te maken.

16.7 Groepsconfiguratie, groepsmasker, groepspositie, bitmasker en bitpositie

De toekenningen in code 16.5 op de regels 12, 14, 24 en 25 zijn tamelijk complex. Dit is nodig omdat er op deze regels slechts een paar bits uit de registers `PORTD.INTCTRL` en `TCE0.INTCTRLA` gewijzigd moeten worden. De andere bits moeten ongewijzigd blijven. Deze bitmanipulaties maken gebruik van de zogenoemde groepsconfiguraties en groepsmaskers.

Bij de Xmega heeft AVR GNU C-compiler een zeer systematische opzet. In paragraaf 15.4 is al uitgelegd dat de registers te benaderen zijn via datastructuren. Op verschillende plaatsen zijn bitmaskers toegepast en in paragraaf 15.12 is de bitpositie gebruikt. In het headerbestand `avr/io.h` eindigen de namen van de macrodefinities vaak met de extensies: `_bm`, `_bp`, `_gc`, `_gm` en `_gp`. Deze extensies betekenen respectievelijk *bit mask*, *bit position*, *group configuration*, *group mask* en *group position* oftewel: bitmasker, bitpositie, groepsconfiguratie, groepsmasker en groepspositie. In code 16.5 komen de eerste vier van deze extensies voor.

De compiler sluit bij de Xmega256a3u bij het insluiten van `avr/io.h` automatisch ook het bestand `avr/iox256a3u.h` in met de feitelijke beschrijving van de registers.

De bitpositie `_bp`

De bitpositie geeft altijd de positie of het nummer van een specifiek bit in een register aan. Enkele voorbeelden van macrodefinities uit `avr/io.h` zijn:

```
#define PIN2_bp          2
#define PIN3_bp          3
#define PIN4_bp          4
#define PORT_INT0LVL0_bp 0
#define PORT_INT0LVL1_bp 1
#define PORT_INT1LVL0_bp 2
#define PORT_INT1LVL1_bp 3
```

De bitpositie is altijd een nummer en is bedoeld om de leesbaarheid van de code te verbeteren. Op regel 20 is direct duidelijk dat de tweede parameter van de functie `bit_is_clear` een pinnummer is:

```
if ( bit_is_clear(PORTD.IN, PIN3_bp) ) {          // zoals het hoort
```

Zonder macrodefinitie kan het minder duidelijk zijn dat 3 een pinnummer is:

```
if ( bit_is_clear(PORTD.IN, 3) ) {                // minder duidelijk
```

Overigens geeft iedere macrodefinitie met de waarde 3 hetzelfde eindresultaat:

```
if ( bit_is_clear(PORTD.IN, PORT_INT1LVL1_bp) ) {   // zeer onduidelijk
```

Ondanks dat het feit de code prima zal functioneren, is deze laatste notatie zeer verwarrend en slecht leesbaar.

Het bitmasker `_bm`

Het bitmasker is altijd een 8-bits getal waarvan alle bits nul zijn op één na. Enkele voorbeelden van macrodefinities uit `avr/io.h` zijn:

```
#define PIN2_bm          0x04
#define PIN3_bm          0x08
#define PIN4_bm          0x10
#define PORT_INT0LVL0_bm  (1<<0)
#define PORT_INT0LVL1_bm  (1<<1)
#define PORT_INT1LVL0_bm  (1<<2)
#define PORT_INT1LVL1_bm  (1<<3)
```

Bitmaskers zijn in dit hoofdstuk en in hoofdstuk 15 al regelmatig gebruikt. Met name in paragraaf 15.6 en 15.4 is al uitgelegd hoe deze maskers bij de bitmanipulaties gebruikt kunnen worden.

De groepsconfiguratie `_gc` en het groepsmasker `_gm`

Bij veel instellingen van de microcontroller gaat het niet om één specifieke bit uit een register, maar om een groep van bits. Voor de interrupt zijn bijvoorbeeld steeds twee bits nodig om de vier interruptniveaus *off*, *low*, *medium* en *high* te definiëren.

Figuur 16.13 : Het interruptcontrolregister `INTCTRL` van de poorten. De bits 1 en 0 bepalen het interruptniveau van interrupt 0 en de bits 3 en 2 bepalen het interruptniveau van interrupt 1. Bit 4 tot en met 7 worden niet gebruikt.

In figuur 16.13 staat het interruptcontrolregister `INTCTRL` met de bits voor de interruptniveaus van interrupt 0 en interrupt 1. Bit 1 en 0 bepalen het interruptniveau van interrupt 0 en bit 3 en 2 het niveau van interrupt 1.

Bij de Xmega legt de AVR GNU C-compiler deze bitgroepen met enumeraties vast. Deze bitgroepen worden groepsconfiguraties genoemd. De groepsconfiguraties voor de externe interrupt 0 en 1 van de poorten zijn als volgt gedefinieerd:

```
typedef enum PORT_INT0LVL_enum
{
    PORT_INT0LVL_OFF_gc = (0x00<<0),  /* Interrupt Disabled */
    PORT_INT0LVL_LO_gc  = (0x01<<0),  /* Low Level */
    PORT_INT0LVL_MED_gc = (0x02<<0),  /* Medium Level */
    PORT_INT0LVL_HI_gc  = (0x03<<0),  /* High Level */
} PORT_INT0LVL_t;

typedef enum PORT_INT1LVL_enum
{
    PORT_INT1LVL_OFF_gc = (0x00<<2),  /* Interrupt Disabled */
    PORT_INT1LVL_LO_gc  = (0x01<<2),  /* Low Level */
    PORT_INT1LVL_MED_gc = (0x02<<2),  /* Medium Level */
    PORT_INT1LVL_HI_gc  = (0x03<<2),  /* High Level */
} PORT_INT1LVL_t;
```

In code 16.1 is bij de initialisatie van de externe interrupt 0 van poort D de groepsconfiguratie aan het interruptcontrolregister toegekend:

```
PORTD.INTCTRL  = PORT_INT0LVL_LO_gc;
```

De bits INT1LVL[1:0] van de externe interrupt 1 zijn nu ook automatisch nul gemaakt. Dit is eigenlijk identiek met deze toewijzing:

```
PORTD.INTCTRL  = PORT_INT1LVL_OFF_gc | PORT_INT0LVL_LO_gc;
```

Als de externe interrupt 1 gebruikt wordt met een hoog interruptniveau en interrupt 0 op een laag niveau, kan dat tegelijkertijd gedaan worden bij de initialisatie:

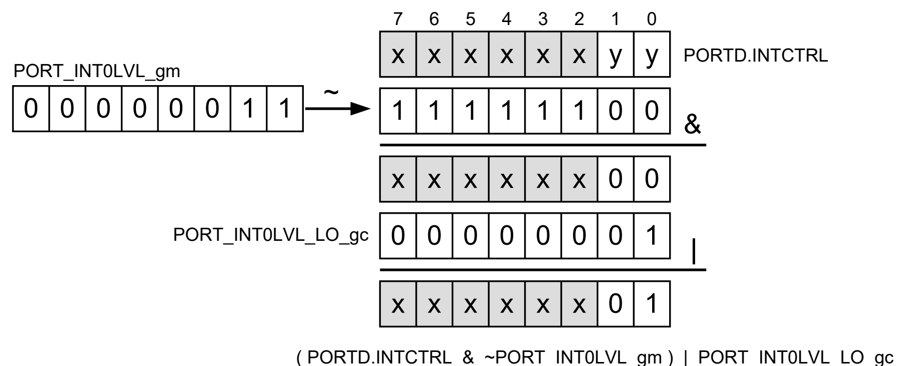
```
PORTD.INTCTRL  = PORT_INT1LVL_HI_gc | PORT_INT0LVL_LO_gc;
```

In veel gevallen worden de bitgroepen niet bij de initialisatie maar ergens anders in de code gewijzigd. In code 16.5 veranderen de INT0LVL-bits bij beide interruptfuncties. In dit voorbeeld worden de INT1LVL-bits niet veranderd en voldoet een directe toewijzing. Toch is er in code 16.5 voor gekozen om expliciet alleen de INT0LVL-bits te wijzigen. Dit voorkomt problemen bij eventuele latere uitbreidingen van de software.

Om een bitgroep te manipuleren is een masker nodig om de betreffende bits af te zonderen. Dit masker wordt het groepsmasker genoemd. In avr/io.h zijn de groepsmaskers voor de interruptniveaus van de externe interrupt 0 en 1 als volgt gedefinieerd:

```
#define PORT_INT0LVL_gm  0x03
#define PORT_INT1LVL_gm  0x0C
```

Bij iedere groepsconfiguratie hoort ook een groepsmasker. De bits van het groepsmasker, die overeenkomen met de groepsconfiguratie, zijn hoog. De andere bits zijn laag. Voor het groepsmasker PORT_INT0LVL_gm, dat hoort bij de enumeratie PORT_INT0LVL_t, zijn de bits 0 en 1 hoog.



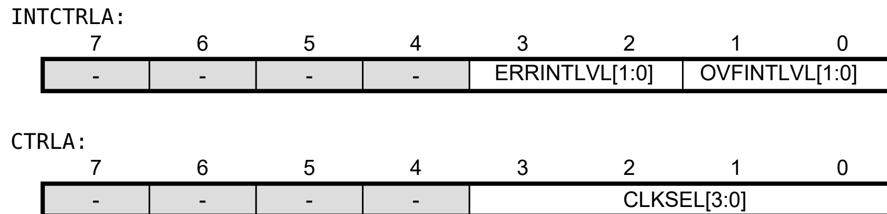
Figuur 16.14 : De bitbewerkingen om alleen de INT0LVL-bits van register INTCTRL te veranderen.

Het groepsmasker PORT_INT0LVL_gm is in code 16.5 op regel 25 gebruikt om bit 0 en bit 1 laag te maken terwijl de overige bits hun waarde behouden. De of-functie voegt de nieuwe groepsconfiguratie, PORT_INT0LVL_LO_gc, toe.

```
PORTD.INTCTRL  = (PORTD.INTCTRL & ~PORT_INT0LVL_gm) | PORT_INT0LVL_LO_gc;
```

Figuur 16.14 licht deze toekenning stapsgewijs toe.

In code 16.5 is dezelfde methode gebruikt bij het in- en uitschakelen van de timer-overflowinterrupt. Register `INTCTRLA` van de timer/counter heeft immers naast de twee `OVFINTLVL`-bits voor het interruptniveau van de overflowinterrupt ook nog twee `ERRINTLVL`-bits, zoals in figuur 16.15 te zien is.



Figuur 16.15 : De registers `INTCTRLA` en `CTRLA` van de timer/counter.

Deze methode is daarentegen bij het in- en uitschakelen van de timer/counter met behulp van de `CLKSEL`-bits uit register `CTRLA` niet gebruikt. Dat is niet nodig omdat — zoals in figuur 16.15 te zien is — register `CTRLA` alleen `CLKSEL`-bits bevat en geen andere bits. De toekenning in code 16.5 op regel 15:

```
TCC0.CTRLA     = TC_CLKSEL_DIV8_gc;
```

verandert in alle gevallen alleen de `CLKSEL`-bits.

In de besproken voorbeelden is het niet per se nodig bij het toewijzen van de `INT0LVL`- en `OVFINTLVL`-bits de methode met het groepsmasker te gebruiken. De andere bits, `INT1LVL[1:0]`- en `ERRINTLVL[1:0]`, uit de registers worden immers niet gebruikt. Als de schakeling wordt uitgebreid met een extra drukknop, die gebruik maakt van externe interrupt 1, is de methode met het groepsmasker voor de `INT0LVL`-bits wel noodzakelijk.

De groepspositie `_gp`

De groepspositie geeft altijd de positie of het bitnummer van het minst significante bit van een groepsconfiguratie in een register. Enkele voorbeelden, zoals deze gedefinieerd zijn in `avr/io.h`, zijn:

```
#define PORT_INT0LVL_gp   0
#define PORT_INT1LVL_gp   2
#define TC0_OVFINTLVL_gp  0
#define TC0_ERRINTLVL_gp  2
#define AC_WSTATE_gp      6
```

Een voorbeeld, waarbij de groepspositie nuttig is, is het bepalen van de status bij de windowmodus bij de analoge comparator. Bit 6 en 7 van het statusregister definiëren deze status.

```
status = (ACA.STATUS & AC_WSTATE_gm) >>  AC_WSTATE_gp;
```

Het masker `AC_WSTATE_gm` maskeert bit 6 en 7 en de uitkomst schuift vervolgens `AC_WSTATE_gp` naar rechts. De variabele `status` krijgt afhankelijk van de status de waarde `0`, 1, 2 of 3. De bitmaskering is hier strikt genomen niet nodig, omdat bij het helemaal naar rechts schuiven van bit 6 en 7 alle andere bits ook automatisch nul zijn.

17

Displays

Doelstelling	Dit hoofdstuk behandelt de aansturing van verschillende eenvoudige displays. Daarbij leer je hoe je meerdere leds kunt aansturen, leer je wat een opzoektabel is en hoe je deze gebruikt. Verder leer je hoe meerdere ingangen kunt uitlezen door middel van *polling* en met externe interrupts.

Onderwerpen

De behandelde onderwerpen zijn:

- 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()`.
- Het uitlezen van meerdere drukknoppen met polling en met een interrupt.

Diverse voorbeelden demonstreren de aansturing van meerdere leds.

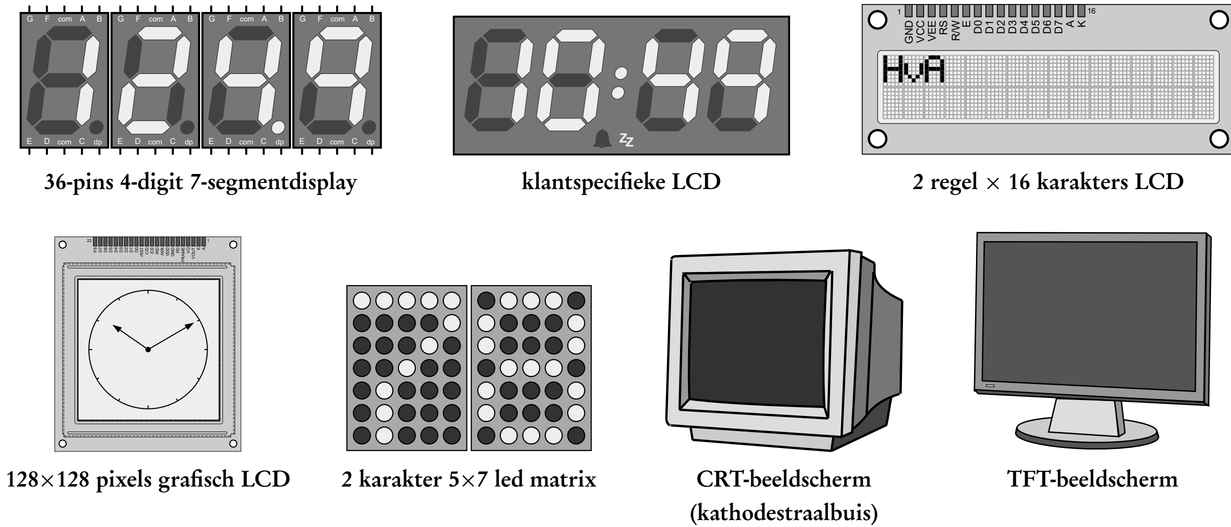
- Verschillende alternatieven om een 10-bits ledbar aan te sturen.
- Cijfers tonen op een dotmatrix.
- Cijfers tonen op een dotmatrix met een interruptfunctie en een timer.
- Cijfers tonen op een dotmatrix met een opzoektabel in flash.
- Een willekeurige getal tonen op een 7-segmentdisplay.

Twee voorbeelden demonstreren het gebruik van meerdere drukknoppen:

- Het uitlezen van zes drukknoppen met behulp van polling.
- Het uitlezen van zes drukknoppen met een externe interrupt.

Een complicatie bij het werken met een microcontroller is, dat er standaard geen toetsenbord en beeldscherm aanwezig is. Voor het testen is altijd een opstelling nodig met extra elektronica om informatie aan de microcontroller door te geven en om de resultaten zichtbaar te maken.
Het weergeven van informatie kan met leds, ledbars, ledarrays, ledmatrices, 7-segmentdisplays, een karaktergeoriënteerd display, een eenvoudige grafische display of met een standaard TFT- of CRT-beeldscherm. Figuur 17.1 toont diverse typen displays.

Leds, ledarrays, 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.



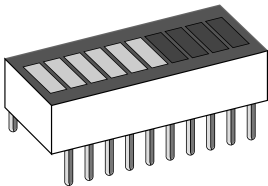
Figuur 17.1 : Diverse typen displays.
Displays zijn te verkrijgen in veel soorten, maten en prijzen. De schaal, waarop deze displays zijn afgebeeld, is willekeurig. Voor microcontrollersystemen zijn vooral de karaktergeoriënteerde en kleinere grafische displays interessant.

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.

Dit hoofdstuk behandelt een aantal eenvoudige displays: de eendimensionale ledarray of ledbar, de ledmatrix of dotmatrix en de 7-segmentdisplays. In dit hoofdstuk worden meerdere algoritmes en oplossingen om de displays aan te sturen gepresenteerd. Gewone functies en interruptfuncties zijn daarbij belangrijke hulpmiddelen net als de zogenoemde opzoektabellen of *look-up tables*.

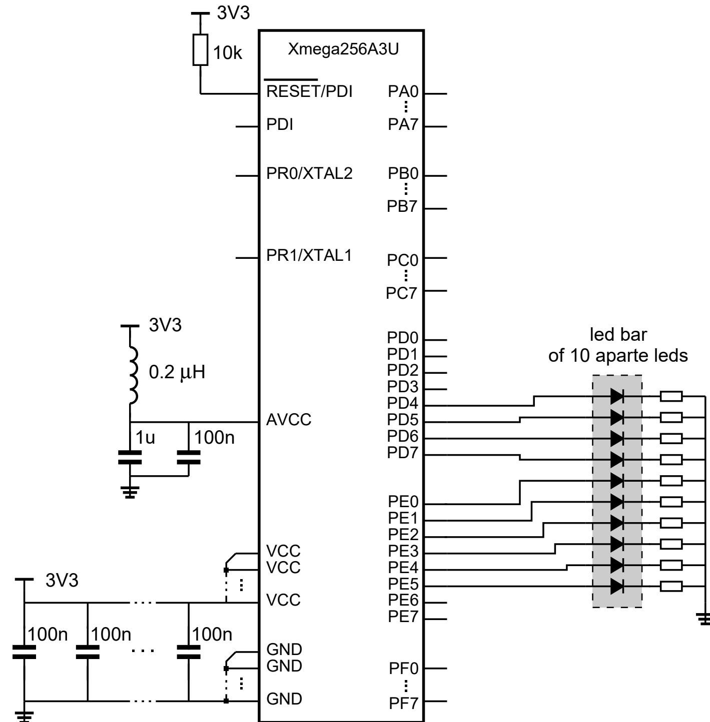
17.1 De ledbar

In hoofdstuk 15 is uitgelegd hoe een led kan worden aangestuurd. Door meerdere leds te gebruiken zijn interessantere applicaties te maken. Met tien leds kan bijvoorbeeld het niveau van een voorraadvat worden weergegeven. Als het vat leeg is, zijn alle leds uit en als het vat vol is, zijn alle leds aan. Voor dit soort toepassingen zijn speciale ledbars beschikbaar met acht of tien leds. In figuur 17.2 staat een ledbar met tien leds in een DIP-uitvoering. Bij een SMD-ontwerp wordt vaak met losse leds een dergelijke ledbar gemaakt.



Figuur 17.2 : Een ledbar met tien leds in DIP-uitvoering.
Als zes leds branden geeft dat bijvoorbeeld aan dat het vat voor 60% is gevuld.

In figuur 17.3 staat de schakeling die tien leds aanstuurt. Tien leds passen niet op één enkele poort van de microcontroller. De leds zijn daarom verdeeld over poort D en poort E. Vier leds zijn aangesloten op de pinnen 4 tot en met 7 van poort D en de andere leds op de pinnen `0` tot en met 5 van poort E. Pin 6 en 7 blijven zo beschikbaar voor een UART-aansluiting. De tien weerstanden zijn nodig om de stroom te begrenzen en de leds de juiste felheid te geven.



Figuur 17.3 : De schakeling voor het aansturen van een ledbar met tien leds.

Met tien leds kunnen elf verschillende niveaus worden weergegeven, namelijk het niveau waarbij alle leds uit zijn en de tien niveaus waarbij één of meer leds branden. Stel dat het bereik van de gemeten waarde loopt vanaf `0` tot en met 1023, moet dit bereik verdeeld worden over de elf niveaus.

Eerste opzet: waarde vergelijken met de verschillende niveaus

Het verdelen van de elf niveaus over het bereik van de meetwaarden kan op veel verschillende manieren worden gedaan. Een eerste mogelijkheid is om de gemeten waarde te vergelijken met de verschillende niveaus. In pseudocode zou dat er als volgt uit kunnen zien:

```
als waarde > LEVEL_LED_10
    alle leds aan
anders als waarde > LEVEL_LED_09
    leds 1 tot en met 9 aan en led 10 uit
anders als ···
    ⋮
anders als waarde > LEVEL_LED_01
    led 1 aan en de andere leds uit
anders
    alle leds uit
```

De verschillende niveaus hangen volgens vergelijking 17.1 zowel af van het bereik van de te meten waarden, als van het aantal leds `NUM_LEDS` en het betreffende niveau- of lednummer *i*.

$$\texttt{LEVEL_LED_}i = \frac{i}{\texttt{NUM_LEDS} + 1}\, bereik \qquad (17.1)$$

Het bereik wordt bepaald door de maximale waarde `MAX_VALUE`. Deze tien macro's definiëren dan de verschillende niveaugrenzen:

```
#define LEVEL_LED_10   ( 10 * (MAX_VALUE + 1) / (NUM_LEDS + 1) )
#define LEVEL_LED_09   (  9 * (MAX_VALUE + 1) / (NUM_LEDS + 1) )
    ⋮
#define LEVEL_LED_01   (  1 * (MAX_VALUE + 1) / (NUM_LEDS + 1) )
```

Code 17.1 : Het aansturen van een ledbar met tien leds volgens de derde opzet.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define NUM_LEDS   10
7  #define MAX_VALUE  1023
8
9  void display_level(uint8_t level)
10 {
11   PORTD.OUTCLR = PIN7_bm|PIN6_bm|PIN5_bm|PIN4_bm;
12   PORTE.OUTCLR = PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
13   if ( level >=  1 )  PORTD.OUTSET = PIN4_bm;
14   if ( level >=  2 )  PORTD.OUTSET = PIN5_bm;
15   if ( level >=  3 )  PORTD.OUTSET = PIN6_bm;
16   if ( level >=  4 )  PORTD.OUTSET = PIN7_bm;
17   if ( level >=  5 )  PORTE.OUTSET = PIN0_bm;
18   if ( level >=  6 )  PORTE.OUTSET = PIN1_bm;
19   if ( level >=  7 )  PORTE.OUTSET = PIN2_bm;
20   if ( level >=  8 )  PORTE.OUTSET = PIN3_bm;
21   if ( level >=  9 )  PORTE.OUTSET = PIN4_bm;
22   if ( level >= 10 )  PORTE.OUTSET = PIN5_bm;
23 }
24
25 int main(void)
26 {
27   uint8_t   level;
28   uint16_t  value = 0;
29
30   PORTD.DIRSET = 0xF0;
31   PORTE.DIRSET = 0x3F;
32
33   while(1) {
34     level = value * (NUM_LEDS + 1)  / (MAX_VALUE + 1);
35
36     display_level(level);
37
38     _delay_ms(10);
39
40     value++;
41     if (value > MAX_VALUE) {
42       value = 0;
43     }
44   }
45 }
```

De aansturing van de leds is ook niet zo eenvoudig. Bij alle elf niveaus zijn er steeds andere leds aan en uit en bovendien zijn de leds verdeeld over twee poorten. Voor het negende niveau, waarbij alle leds, behalve het tiende led, aan zijn, moeten alle uitgangen hoog zijn behalve pin 5 van poort E:

```
PORTD.OUTSET = PIN7_bm|PIN6_bm|PIN5_bm|PIN4_bm;
PORTE.OUTSET = PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
PORTE.OUTCLR = PIN5_bm;
```

Al met al levert deze aanpak relatief veel code op, maar het is in ieder geval een programma dat wel een gestructureerde opzet heeft.

Tweede opzet: eerst alle leds uit en daarna één voor één aan

Een andere aanpak, die de aansturing van de leds behoorlijk vereenvoudigt, maakt eerst alle uitgangen laag en zet daarna de leds één voor één aan als de waarde, `value`, groter is dan die van het betreffende niveau:

```
PORTD.OUTCLR = PIN7_bm|PIN6_bm|PIN5_bm|PIN4_bm;
PORTE.OUTCLR = PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
if ( value > LIMIT_LED_10 )  PORTE.OUTSET = PIN5_bm;
if ( value > LIMIT_LED_09 )  PORTE.OUTSET = PIN4_bm;
    ⋮
if ( value > LIMIT_LED_01 )  PORTD.OUTSET = PIN4_bm;
```

Derde opzet: eerst het niveaunummer berekenen

Een nog eenvoudiger variant is om de gemeten waarde niet met de niveaugrenzen te vergelijken, maar om uit de gemeten waarde eerst het niveau te berekenen volgens formule 17.2 en daarna dit getal met de niveaunummers of lednummers te vergelijken.

$$\mathtt{niveaunummer} = \frac{\mathtt{NUM_LEDS} + 1}{\mathtt{bereik}}\,\mathtt{waarde} \tag{17.2}$$

In code 17.1 staat een programma dat dit algoritme gebruikt om de tien leds van een ledbar aan te sturen. Er is geen ingangssignaal. De gemeten waarde `value` is een 16-bits getal dat in de oneindige lus iedere 10 ms wordt opgehoogd tot een maximale waarde `MAX_VALUE`. De waarde van `value` wordt met de bovenstaande formule lineair afgebeeld op de ledbar. Als de maximale waarde is bereikt, wordt `value` weer nul gemaakt.

Op regel 34 van code 17.1 wordt met behulp van formule 17.2 het niveaunummer `level` berekend en daarna met de functie `display_level` afgebeeld op de tien leds. Deze functie zet op regel 11 en regel 12 alle leds uit, en vanaf regel 13 worden de leds weer aangezet, als het niveaunummer hoger als dan het nummer van de led.

Vierde opzet: bij het vergelijken een `for`-lus gebruiken

In code 17.1 staat vanaf regel 13 tien keer bijna hetzelfde statement. Het ligt voor de hand om een **`for`**-lus te gebruiken. In code 17.1 gebruiken de eerste vier statements poort D en de volgende zes poort E. In de functie `display_level` van code 17.2 zijn de tien statements uit code 17.1 vervangen door een **`for`**-lus.

Code 17.2 : De functie `display_level` voor het afbeelden op een ledbar met een `for`-lus.

```
 9 void display_level(uint8_t level)
10 {
11   PORTD.OUTCLR = PIN7_bm|PIN6_bm|PIN5_bm|PIN4_bm;
12   PORTE.OUTCLR = PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
13
14   for (int i = 0; i < NUM_LEDS; i++) {
15     if ( level > i ) {
16       if (i < 4) {
17         PORTD.OUTSET = (1 << (i+4));
18       } else {
19         PORTE.OUTSET = (1 << (i-4));
20       }
21     }
22   }
23 }
```

De index `i` varieert van `0` tot en met 9. Omdat deze index één lager is dan de niveaunummers uit code 17.1, is bij het vergelijken met `level` de `>=` vervangen door een `>`-teken. Voor de leds die aan moeten, maakt de schuifoperatie de betreffende aansluiting hoog. Voor de indices `0` tot en met 3 gaat het om de pinnen 4 tot en met 7 van poort D en voor de indices 4 tot en met 9 om de pinnen `0` tot en met 5 van poort E.

Vijfde opzet: bij het afbeelden een hulpvariabele gebruiken

De functie `display_level` uit code 17.2 smeert het aansturen van de verschillende leds over relatief veel klokslagen uit. Eerst worden steeds alle leds uitgezet en daarna worden de benodigde leds één voor één weer aangezet. Bij een hoge meetwaarde is het tiende led relatief lang uit.

Code 17.3 : De functie `display_level` met een hulpvariabele `bar` voor het afbeelden.

```
 9 void display_level(uint8_t level)
10 {
11   uint16_t bar;
12
13   bar = 0x0000;
14   for (int i = 0; i < NUM_LEDS; i++) {
15     if ( i >= level )  {
16       break;
17     }
18     bar = (bar << 1) | 0x0001;
19   }
20
21   PORTD.OUT = (PORTD.OUT & 0x0F) | ( (bar<<4) & 0xF0);
22   PORTE.OUT = (PORTE.OUT & 0xC0) | ( (bar>>4) & 0x3F);
23 }
```

Bij de functie `display_level` uit code 17.3 is een 16-bits hulpvariabele `bar` gebruikt, die de af te beelden waarden van de uitgangen bevat. De tien minst significante bits van `bar` representeren de leds. Als de bit 1 is, is de uitgang hoog en is de led aan. Als de bit `0` is, is de uitgang laag en is de led uit. Op regel 13 worden alle leds

uitgezet. De **for**-lus op regel 14 zet de leds één voor één weer aan. Als de index van de led groter of gelijk is aan het gemeten niveau blijft de led aan en blijven de volgende leds uit. Na de test op regel 15 schuiven op regel 18 de bits van bar één positie naar links en wordt de minst significante bit hoog gemaakt.

De vier minst significante bits van bar worden op regel 21 aan de pinnen PD4 tot en met PD7 toegekend en de zes andere bits worden op regel 22 aan de pinnen PE0 tot en met PE5 toegekend.

Code 17.4 : De functie display_level voor het afbeelden op een ledbar.

```
 9  void display_level(uint8_t level)
10  {
11    uint16_t  bar;
12    uint8_t   bar_d, bar_e;
13
14    bar = 0x0000;
15    for (int i = 0; i < NUM_LEDS; i++) {
16      if ( i >= level )  {
17        break;
18      }
19      bar = (bar << 1) | 0x0001;
20    }
21
22    bar_d = (PORTD.OUT & 0x0F) | ( (bar<<4) & 0xF0);
23    bar_e = (PORTE.OUT & 0xC0) | ( (bar>>4) & 0x3F);
24    PORTD.OUT = bar_d;
25    PORTE.OUT = bar_e;
26  }
```

Zesde opzet: bij het afbeelden drie hulpvariabelen gebruiken

De bitbewerkingen uit code 17.3, die nodig zijn om uit bar de waarden van de uitgangen te bepalen, zijn relatief complex. De uitgangen van poort E veranderen 14 klokslagen later dan die van poort D. De functie display_level uit code 17.4, kent de nieuwe waarden van de uitgangen aan twee hulpvariabelen bar_d en bar_e toe en kent daarna deze hulpvariabelen toe aan respectievelijk PORTE.OUT en PORTD.OUT. In dit geval verandert poort E een paar klokslagen nadat poort D is gewijzigd.

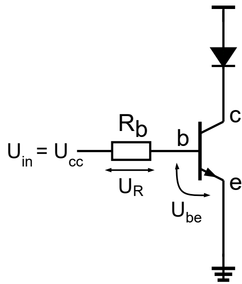
Conclusie

Er zijn bij ieder probleem veel verschillende oplossingen mogelijk. De ene oplossing hoeft niet beter te zijn dan de andere. Afhankelijk van de situatie is in het ene geval een bepaalde oplossing beter en in een andere geval kan een andere oplossing meer geschikt zijn. In dit voorbeeld maakt het niet uit welke oplossing er gebruikt wordt; alle zes alternatieven tonen de gemeten waarde value op de tien leds.

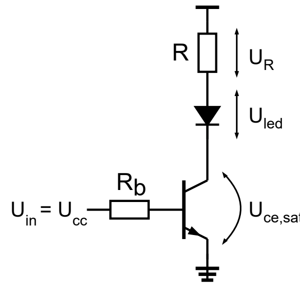
Het is altijd verstandig om verschillende alternatieven te bedenken, de voor- en nadelen te bestuderen en de gemaakte keuzes goed te documenteren.

17.2 Aansturing leds

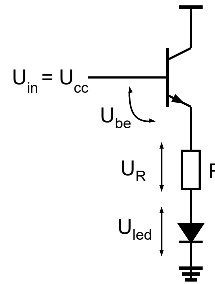
De stroom door de led van figuur 15.1 gaat via de microcontroller naar de referentie. Per pin kan de Xmega niet meer dan 20 mA leveren of afvoeren. Per VCC-aansluiting is de maximale stroom 100 mA of 200 mA. Het is altijd nuttig om de microcontroller geen of weinig stroom te laten leveren.
De figuren 17.4 tot en met 17.7 geven een aantal configuraties waarbij niet de microcontroller maar een transistor de stroom levert.



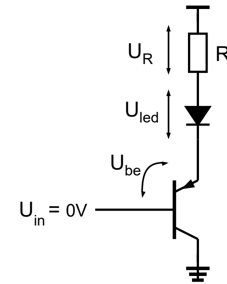
Figuur 17.4 : Aansturing led met een transistor en een basisweerstand.



Figuur 17.5 : Aansturing led met een transistor in verzadiging.



Figuur 17.6 : Aansturing led met een NPN-transistor als stroombron.



Figuur 17.7 : Aansturing led met een PNP-transistor als stroombron.

Figuur 17.4 gebruikt een NPN-transistor met een weerstand bij de basis. De uitgang van de Xmega 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:

$$R_b = \frac{U_R}{I_b} = \frac{\beta(U_{cc} - U_{be})}{I_c} \tag{17.3}$$

De voorbeelden in deze paragraaf gebruiken een gewenste stroom en spanning voor de led en een specifieke β 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.

Als de gewenste stroom door de led 15 mA is, dan is — met U_{cc} 3,3 V, U_{be} 0,7 V en β 200 — de basisweerstand R_b ongeveer 33 kΩ. De basisstroom, die door de microcontroller moet worden geleverd is slechts 75 μA.
Deze methode wordt vooral toegepast als de leds niet continu branden, bijvoorbeeld bij de multiplexing voor een ledmatrix.

Bovendien zijn de berekende waarden uiterste waarden. Een ledstroom van 15 mA is bij moderne leds erg groot. Bij 2 mA geven deze leds al voldoende licht.

In figuur 17.5 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:

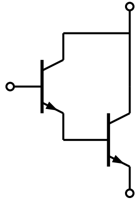
$$R = \frac{U_R}{I_c} = \frac{U_{cc} - U_{ce,sat} - U_{led}}{I_c} \tag{17.4}$$

Bij gemultiplext aansturen van dotmatrices en 7-segmentdisplays, is een hogere stroom door de leds mogelijk en kunnen de weerstandswaarden dus kleiner zijn.

Met $U_{ce,sat}$ gelijk aan 0,2 V, een ledspanning van 2 V en een ledstroom van 15 mA is R ongeveer 75 Ω. Voor de basisweerstand geldt:

$$R_b = \frac{U_{cc} - U_{be}}{I_b} = \frac{\beta(U_{cc} - U_{be})}{I_c} \tag{17.5}$$

In verzadiging is β vaak een factor 5 lager. De basisweerstand is dan ongeveer 7 kΩ.



Figuur 17.8 :
De darlingtontransistor. Deze transistor bestaat in feite uit twee bipolaire transistoren. De eerste transistor versterkt de stroom een factor β en de tweede transistor versterkt deze stroom met nog eens een factor β. De totale versterking is dan β^2.

De schakeling van figuur 17.6 lijkt op die van figuur 17.5. Er zijn echter 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:

$$R = \frac{U_R}{I_c} = \frac{U_{cc} - U_{be} - U_{led}}{I_c} \tag{17.6}$$

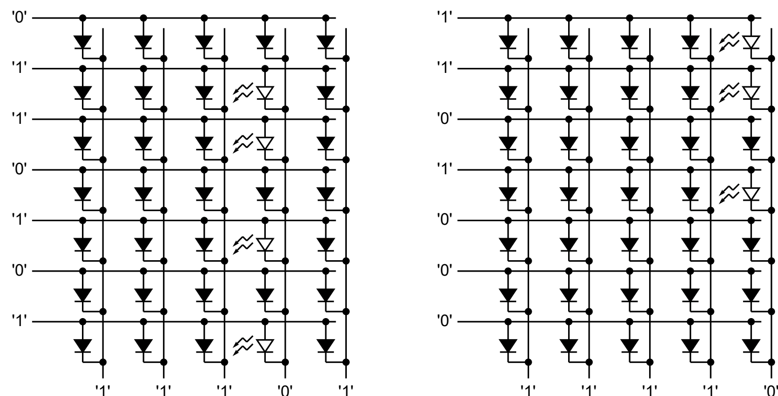
Met U_{be} gelijk aan 0,7 V, een ledspanning van 2 V en een ledstroom van 15 mA is R ongeveer 40 Ω.

Figuur 17.7 is het complement van figuur 17.6. 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 17.6 en dus is R weer 40 Ω.

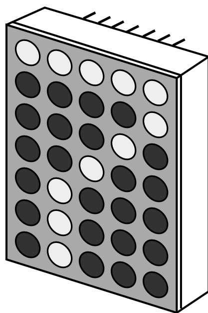
Duidelijk is dat met een transistor de led op vele manieren kan worden aangestuurd en dat de weerstandswaarden uitgerekend kunnen worden als de parameters van de gebruikte componenten bekend zijn. Als de simulatiemodellen beschikbaar zijn, kunnen de berekende waarden geverifieerd worden met een simulator. Ook kan met een proefopstelling de ledstroom worden bepaald.

Bij de aansturing van meerdere leds is een optie om een transistor-array toe te passen, zoals de ULN2803A of de TN0604/TP0604. De UN2803A bevat acht darlingtontransistoren. De TN0604 en de TP0604 bevatten respectievelijk vier NMOS- en vier PMOS-transistoren.

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.



Figuur 17.9 : 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.



Figuur 17.10 : Een 5x7-dotmatrix met een array van vijf bij zeven leds.

17.3 Een tweedimensionale ledarray of dotmatrix

Voor het weergeven van informatie wordt vaak een tweedimensionaal ledarray gebruikt. Een array met vijf bij zeven leds kan alle karakters zichtbaar maken. Een voorbeeld van een ledarray of dotmatrix staat in figuur 17.10 en figuur 17.9 toont het principe 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.

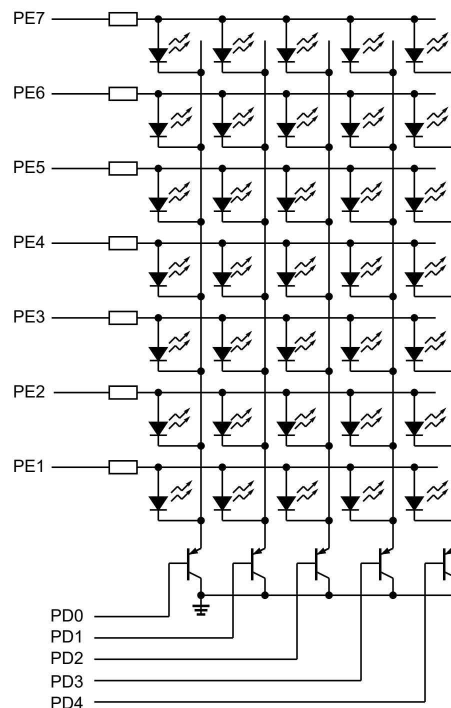
Dit effect wordt gebruikt bij het maken van een televisiebeeld. Tot voor kort was in Europa 50 Hz de standaard voor televisies en monitoren. Tegenwoordig is dat vaak 100 Hz.

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_{active}. In dit voorbeeld is t_{active} 4 ms.

Figuur 17.11 laat zien dat de leds rechtstreeks vanuit de microcontroller worden aangestuurd maar dat de kathodes via PNP-transistoren met poort D verbonden zijn. De stroom door de collector van de PNP-transistor hangt af van het aantal leds dat brandt. De weerstand zorgt ervoor dat de ledstroom niet te groot wordt. Deze stroom hangt volgens formule 17.6 af van de weerstand, de voedingsspanning, de ledspanning en de U_{be} van de transistor.

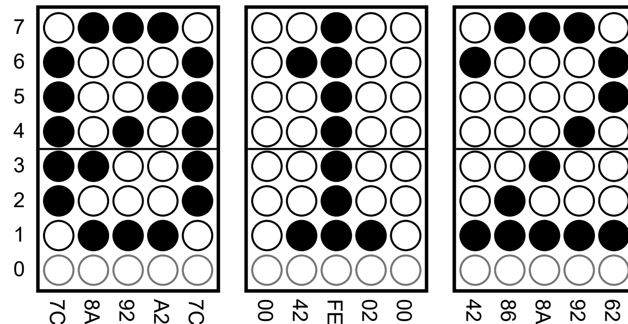
Er zijn twee soorten dotmatrices: met de kathodes aan de kolommen en met de anodes aan de kolommen. Dit wordt ook aangeduid met CC *column cathode* en CA *column anode*.
Hier is een dotmatrix van het type CC gebruikt met de kathodes aan de kolommen.



Figuur 17.11 : De aansturing bij een ledarray of een dotmatrix. De kathodes van de leds zijn per kolom verbonden via een PNP-transistor met de referentie. De basis van deze transistoren wordt aangestuurd vanuit poort D van de microcontroller. De anodes van de leds zijn via een weerstand verbonden met poort E.

Bij een gewenste stroom van 15 mA moet — met U_{cc} 3,3 V, U_{be} 0,7 V en U_{led} 2 V — de weerstand minimaal 40 Ω zijn. Moderne leds branden bij een stroom van 2 mA vaak voldoende fel. In dat geval is een weerstand van bijvoorbeeld 330 Ω geschikt.

Omdat de led maar een vijfde deel van de tijd aan is, kan de ledstroom groter zijn. Mits de gemiddelde stroom 15 mA blijft, is een maximale stroom van 75 mA mogelijk. De weerstanden moeten dan minimaal 8 Ω zijn. Bij moderne leds voldoet een waarde van minimaal 60 Ω.



Figuur 17.12 : 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. De 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.

17.4 Cijfers afbeelden op een dotmatrix

In figuur 17.12 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.

Een opzoektabel is een lijst van sleutels (*keys*) met een 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.

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 wordt gewacht:

```
doe voor elke kolom:
  zet juiste code op uitgangen voor de rij
  maak betreffende kolom laag
  wacht een tijd t_active
```

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 17.12 laat zien dat voor het karakter '0' de waarden 0x7C, 0x8A, 0x92, 0xA2 en 0x7C nodig zijn.

Het programma van code 17.5 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 PORTE en de kolommen aan de vijf minst significante bits van PORTD. Zie ook figuur 17.11.

Uitleg code 17.5 regel 9-20

De macro's MASK_ROW en MASK_COL definiëren de maskers voor de aansluitingen van de rijen en de kolommen. Een alternatief is om de maskers samen te stellen door de betreffende pinmaskers samen te voegen:

```
#define MASK_ROW  (PIN7_bm|PIN6_bm|PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm)
#define MASK_COL  (PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm)
```

Code 17.5 : De cijfers `0` tot en met 9 tonen met een dotmatrix of een ledarray.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define MASK_ROW   0xFE   // 1111 1110
7  #define MASK_COL   0x1F   // 0001 1111
8
9  const uint8_t lookup[][5] = {
10   {0x7C, 0x8A, 0x92, 0xA2, 0x7C}, // 0
11   {0x00, 0x42, 0xFE, 0x02, 0x00}, // 1
12   {0x42, 0x86, 0x8A, 0x92, 0x62}, // 2
13   {0x84, 0x82, 0xA2, 0xD2, 0x8C}, // 3
14   {0x18, 0x28, 0x48, 0xFE, 0x08}, // 4
15   {0xE4, 0xA2, 0xA2, 0xA2, 0x9C}, // 5
16   {0x3C, 0x52, 0x92, 0x92, 0x0C}, // 6
17   {0x80, 0x80, 0x9E, 0xA0, 0xC0}, // 7
18   {0x6C, 0x92, 0x92, 0x92, 0x6C}, // 8
19   {0x60, 0x92, 0x92, 0x94, 0x78}  // 9
20 };
```

```
22 int main(void)
23 {
24   uint8_t t = 0;
25   uint8_t digit = 0;
26
27   PORTE.DIRSET = MASK_ROW;
28   PORTD.DIRSET = MASK_COL;
29   PORTD.OUTSET = MASK_COL;
30
31   while(1) {
32     for (int col=0; col<5; col++) {
33       PORTE.OUT = lookup[digit][col];
34       PORTD.OUT = ~(1 << col);
35       _delay_ms(4);
36     }
37
38     t++;
39     if (t == 50) {
40       digit++;
41       if (digit == 10) digit = 0;
42       t = 0;
43     }
44   }
45 }
```

Uitleg code 17.5 regel 9-20

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 gedefinieerd als `uint8_t`. Het sleutelwoord **`const`** vertelt de compiler dat de array niet gewijzigd mag worden na de initialisatie.

Uitleg code 17.5 regel 32-36

Het algoritme, dat het karakter afbeeldt, kent op regel 33 aan het uitgangsregister van `PORTE` het arrayelement `lookup[digit][col]` toe. Variabele `digit` is de index van het karakter en `col` is de index voor de kolom. Op regel 34 wordt de betreffende kolom laag (`~(1<<col)`) gemaakt. 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

Een neveneffect van deze toekenning is dat de bits 7, 6 en 5 van het uitgangsregister ook hoog worden. De vertragingstijd t_{active} is 4 ms. Omdat er vijf kolommen zijn, wordt het karakter iedere 20 ms ververst.

Uitleg code 17.5 regel 38-41

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 `digit` opgehoogd.

Uitleg code 17.5 regel 29

Bij de initialisatie zijn de vijf kolomuitgangen hoog gemaakt. Dit zorgt er voor dat de leds aanvankelijk allemaal uit zijn.

Het neveneffect bij de toekenning aan het uitgangsregister van poort D, namelijk dat de andere bits van poort D ook wijzigen, treedt ook op bij de toekenning aan het uitgangsregister van poort E. In dat geval wordt bit `0` altijd laag gemaakt. Dit is te voorkomen door bitmaskering toe te passen. De volgende twee toekenningen laten bit `0` van `PORTE.OUT` en de bits 7, 6 en 5 van `PORTE.OUT` ongewijzigd:

```
PORTE.OUT = (PORTE.OUT & ~MASK_ROW) | lookup[digit][col];
PORTD.OUT = (PORTD.OUT & ~MASK_COL) | (~(1<<col) & MASK_COL);
```

De manier van toekennen uit code 17.5 hoeft geen probleem te zijn, als de uitgangsregisters van de andere aansluitingen niet gebruikt worden. Dat is het geval als de aansluitingen helemaal niet gebruikt worden of als ze voor een perifeer functieblok nodig zijn.

17.5 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. Dit voorbeeld kent drie taken: elke 4 ms moet er een kolom met leds worden aangestuurd, daarna moet variabele `t` worden gewijzigd en tenslotte verandert eens per seconde de variabele `digit`.

Een betere methode is om een interrupt te gebruiken. Interrupts zijn al in hoofdstuk 16 uitgebreid besproken. 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. Een timer kan op gezette tijden een interruptfunctie starten, die de informatie op de dotmatrix aanpast.

Tabel 17.1 :
De combinaties van N en `PER` om 4 ms te maken.

N	PER	t
1	8000	4,000
2	4000	4,000
4	2000	4,000
8	1000	4,000
64	125	4,000
256	31	3,968
1024	8	4,096

Code 17.6 gebruikt de *overflow* van timer/counter `0` van poort D. De tijd t tussen twee overflows hangt volgens formule 16.2 af van prescaling N, het aantal getelde klokslagen m en de klokfrequentie. Het aantal te tellen klokslagen wordt ingesteld met het `PER`-register. Tabel 17.1 geeft voor een klokfrequentie van 2 MHz bij alle mogelijke klokdelingen de meest geschikte waarde van `PER` om een tijd van 4 ms te maken.
Bij code 17.6 is gekozen voor een klokdeling van 64, zodat `PER` gelijk aan 125 moet zijn. De teller is dan niet erg actief en de nauwkeurigheid is redelijk hoog ($< 1\%$).

Gedetailleerde uitleg code 17.6

Uitleg code 17.6 regel 23
`volatile`

Het hoofdprogramma geeft via de pointervariabele `p` aan de interruptfunctie door welk karakter afgedrukt moet worden. Het hoofdprogramma laat pointer `p` op regel 54 naar één van de sets met waarden in de opzoektabel wijzen. De interruptfunctie gebruikt op regel 29 pointer `p` om de juiste kolom te selecteren. Om te voorkomen dat de compiler bij de optimalisatie de variabele `p` kwijt raakt, is bij de declaratie van `p` op regel 23 het woord **`volatile`** toegevoegd. De positie van **`volatile`** is heel belangrijk. In dit geval is `p` een volatile pointer naar een `uint8_t`:

```
volatile uint8_t *p;     // p is pointer to volatile uint8_t
uint8_t* volatile p;     // p is volatile pointer to uint8_t
```

Code 17.6 : Cijfers `0` tot en met 9 op dotmatrix met behulp van de timer `0` van poort D.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6
7  #define MASK_ROW   0xFE
8  #define MASK_COL   0x1F
9
10 const uint8_t lookup[][5] = {
11   {0x7C, 0x8A, 0x92, 0xA2, 0x7C}, // 0
12   {0x00, 0x42, 0xFE, 0x02, 0x00}, // 1
13   {0x42, 0x86, 0x8A, 0x92, 0x62}, // 2
14   {0x84, 0x82, 0xA2, 0xD2, 0x8C}, // 3
15   {0x18, 0x28, 0x48, 0xFE, 0x08}, // 4
16   {0xE4, 0xA2, 0xA2, 0xA2, 0x9C}, // 5
17   {0x3C, 0x52, 0x92, 0x92, 0x0C}, // 6
18   {0x80, 0x80, 0x9E, 0xA0, 0xC0}, // 7
19   {0x6C, 0x92, 0x92, 0x92, 0x6C}, // 8
20   {0x60, 0x92, 0x92, 0x94, 0x78}  // 9
21 };
22
23 uint8_t* volatile p;
```

```
25 ISR(TCD0_OVF_vect)
26 {
27   static int col = 0;
28
29   PORTE.OUT = *(p + col);
30   PORTD.OUT = ~(1 << col);
31   if ( col == 4 ) {
32     col = 0;
33   } else {
34     col++;
35   }
36 }
37
38 int main(void)
39 {
40   PORTE.DIRSET = MASK_ROW;
41   PORTD.DIRSET = MASK_COL;
42   PORTD.OUTSET = MASK_COL;
43
44   TCD0.CTRLB     = TC_WGMODE_NORMAL_gc;
45   TCD0.CTRLA     = TC_CLKSEL_DIV64_gc;
46   TCD0.INTCTRLA  = TC_OVFINTLVL_LO_gc;
47   TCD0.PER       = 125;      // t = 64*125/2M = 4 ms
48
49   PMIC.CTRL     |= PMIC_LOLVLEN_bm;
50   sei();
51
52   while(1) {
53     for (int digit=0; digit<=9; digit++) {
54       p = (uint8_t *) lookup[digit];
55       _delay_ms(1000);
56     }
57   }
58 }
```

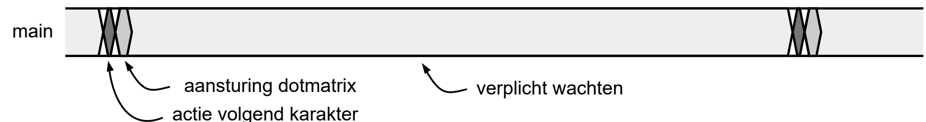
Regel 25-36 `ISR` **`static`**	De interruptfunctie (`ISR`) staat op regel 25 en zet steeds de informatie voor de leds klaar, maakt de betreffende kolom actief en bepaalt de volgende kolomindex `col`. De variabele `col` is **`static`**. Dit betekent dat na afloop van de functie de huidige waarde van de variabele bewaard blijft.
Regel 53-56	Het hoofdprogramma voert voortdurend een **`for`**-lus uit, die steeds `p` naar de volgende set gegevens in de opzoektabel laat wijzen en daarna 1 ms wacht.

Regel 44-47
`TCD0.CTRLB`
`TCD0.CTRLA`
`TCD0.PER`
`TCD0.INTCTRLA`
	Register `CTRLB` stelt de timer/counter in op de normale modus. De klokdeling is met register `CTRLA` ingesteld op 64 en de waarde van `PER` is 125. Register `INTCTRLA` zet het interruptmechanisme van de teller aan op het lage interruptniveau. De wijze van instellen van timer/counter `0` van poort D komt overeen met de instelling van timer/counter `0` van poort E in paragraaf 16.5.

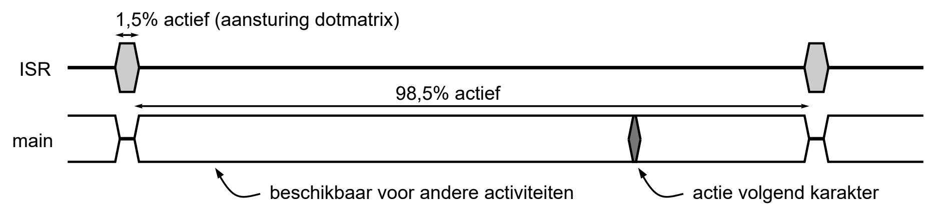
Regel 49-50
`PMIC.CTRL`
`sei()`

De toekenning aan `PMIC.CTRL` maakt het interruptmechanisme gevoelig voor interrupts met een laag interruptniveau. De macro `sei()` zet het interruptmechanisme van de microcontroller aan en is gedefinieerd in `interrupt.h`, die op regel 4 is ingesloten.

Het voordeel van de methode met de timer en de interrupt is dat het hoofdprogramma ontlast wordt. Figuur 17.13 visualiseert de tijdplanning (*scheduling*) voor de oplossing zonder interrupt en figuur 17.14 laat de tijdplanning (*scheduling*) zien voor de oplossing met interrupt.



Figuur 17.13 : De tijdplanning bij de oplossing van code 17.5 zonder interrupt. Het hoofdprogramma wacht voortdurend op de volgende keer dat er een kolom geschreven moet worden.



Figuur 17.14 : De tijdplanning bij de oplossing van code 17.6 met interrupt. De interruptfunctie regelt de aansturing van de dotmatrix en het hoofdprogramma selecteert alleen zo nu en dan het volgende karakter.

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.

Bij de oplossing met de interrupt en de timer doet het hoofdprogramma bijna niets. Voor de interruptfunctie zijn ruim 120 klokslagen nodig. Iedere 8000 klokslagen wordt een volgende kolom geschreven. Dit betekent dat er voor de interruptfunctie ongeveer 1,5% 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.

Code 17.7 : Cijfers afbeelden op een dotmatrix met de gegevens in flash.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6
7  #define MASK_ROW   0xFE
8  #define MASK_COL   0X1F
9
10 const __flash uint8_t lookup [] = {
11   0x7C, 0x8A, 0x92, 0xA2, 0x7C, // 0
12   0x00, 0x42, 0xFE, 0x02, 0x00, // 1
13   0x42, 0x86, 0x8A, 0x92, 0x62, // 2
14   0x84, 0x82, 0xA2, 0xD2, 0x8C, // 3
15   0x18, 0x28, 0x48, 0xFE, 0x08, // 4
16   0xE4, 0xA2, 0xA2, 0xA2, 0x9C, // 5
17   0x3C, 0x52, 0x92, 0x92, 0x0C, // 6
18   0x80, 0x80, 0x9E, 0xA0, 0xC0, // 7
19   0x6C, 0x92, 0x92, 0x92, 0x6C, // 8
20   0x60, 0x92, 0x92, 0x94, 0x78  // 9
21 };
22
23 const __flash uint8_t* ptr = lookup;
24 volatile int show_digit= 0;
25
26 ISR(TCD0_OVF_vect)
27 {
28   static int col = 4;
29   static int row = 0;
30
31   if (col==4) {
32     col = 0;
33     row = show_digit;
34   } else {
35     col++;
36   }
37   PORTE.OUT = *(ptr+5*row+col);
38   PORTD.OUT = ~_BV(col);
39 }
```

```
41 int main(void)
42 {
43   PORTE.DIR    = MASK_ROW;
44   PORTD.DIR    = MASK_COL;
45   PORTD.OUTSET = MASK_COL;
46
47   TCD0.CTRLB     = TC_WGMODE_NORMAL_gc;
48   TCD0.CTRLA     = TC_CLKSEL_DIV64_gc;
49   TCD0.INTCTRLA  = TC_OVFINTLVL_LO_gc;
50   TCD0.PER       = 125;
51
52   PMIC.CTRL     |= PMIC_LOLVLEN_bm;
53   sei();
54
55   while(1) {
56     for (int digit=0; digit<=9; digit++) {
57       show_digit = digit;
58       _delay_ms(1000);
59     }
60   }
61 }
```

17.6 Cijfers afbeelden op een dotmatrix met de gegevens in flash

De opzoektabel uit code 17.6 staat net als de rest van het programma en alle andere globale variabelen in het programmageheugen. Bij de initialisatie plaatsen de opstartroutines, die de compiler aan het programma toevoegt, alle globale variabelen — en dus ook de opzoektabel — in het datageheugen. Dit vindt plaats voordat de hoofdroutine `main` begint.

De opzoektabel uit code 17.6 staat dus zowel in het programmageheugen als in het datageheugen, dus zowel in het flashgeheugen als in het RAM-geheugen. In dit geval zijn er vijftig bytes voor de tabel nodig. Het RAM-geheugen van de

Xmega256a3u is 16 KB. Zeker als er meer karakters of karakters met meer pixels nodig zijn, kan dit een grote aanslag zijn op de beschikbare dataruimte.

Het flashgeheugen van de Xmega256a3u is ruim 256 KB groot. Het is daarom vaak verstandig om grote opzoektabellen niet in het RAM te plaatsen, maar direct in het programmageheugen te benaderen. In code 17.7 is er voor gezorgd dat de opzoektabel alleen in het flashgeheugen staat en niet naar het RAM-geheugen wordt gekopieerd. Het programma haalt de gegevens van de opzoektabel nu direct uit het flashgeheugen.

In dit geval is er voor gekozen om de tabel 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 17.7 past alleen de index van het cijfer dat getoond moet worden aan. Deze index is in feite het rijnummer uit de tabel. Deze index wordt opgeslagen in de globale variabele `show_digit`.
De interruptfunctie past het rijnummer aan als alle kolommen geschreven zijn. Dus als kolomnummer 4 is, wordt `show_digit` toegekend aan `row`.

Uitleg code 17.7 regel 10
`__flash`

De *qualifier* `__flash` zorgt ervoor dat de opzoektabel `lookup` alleen in flash staat en niet ook nog in het RAM-geheugen wordt geplaatst. Een *qualifier* is een toevoeging aan een typedefinitie. Andere voorbeelden van *qualifiers* zijn **`const`** en **`volatile`**. De *qualifier* `__flash` is pas geïntroduceerd bij versie 4.7 van *avr-gcc*. Eerdere versies gebruiken een macro `PROGMEM` en een includebestand `pgmspace.h`. Door de macro `PROGMEM` bij een typedeclaratie te plaatsen, wordt het attribuut `__progmem__` aan de declaratie toegevoegd. Dit attribuut zorgt er — net als de *qualifier* `__flash` — voor dat bij het starten van het programma de gegevens niet naar het datageheugen worden gekopieerd.

```
const __flash int16_t x;
const int8_t  y __attribute__((__progmem));
const int32_t z PROGMEM;
```

Bovenstaande declaraties plaatsen de variabelen alleen in het programmageheugen en niet in het datageheugen. De declaraties van `y` en `z` zijn identiek en zijn functioneel anders dan de declaratie van `x`.
Het grote verschil is dat er bij het gebruik van `PROGMEM` speciale functies nodig zijn uit `pgmspace.h` om het flashgeheugen te lezen. Bij het gebruik van `__flash` is dat niet nodig. Daarom heeft de nieuwe methode met `__flash` de voorkeur. In paragraaf 23.8 staat meer informatie over het lezen uit het flashgeheugen en het gebruik van `__flash` en `PROGMEM`.

Regel 10
`const`

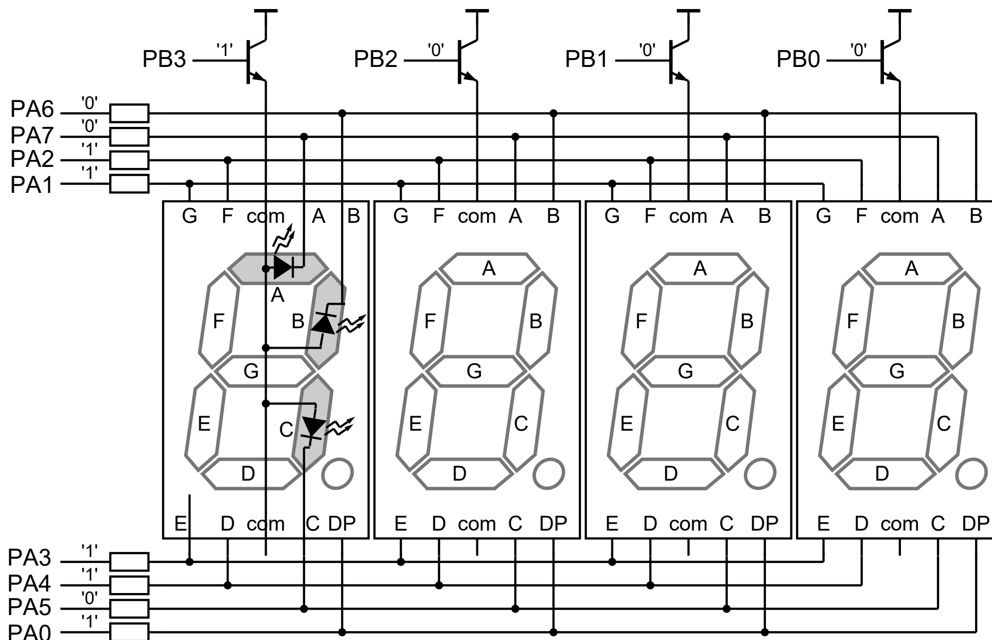
In het algemeen is het niet mogelijk om in het flashgeheugen te schrijven. Dat kan alleen via een bootloader. Omdat een gewone applicatie alleen gegevens uit het flash leest, zijn deze onveranderlijk. Bij de declaratie van deze variabelen moet daarom het sleutelwoord **`const`** worden geplaatst.

Regel 23
`ptr`

Bij de declaratie van de pointer `ptr` is ook `__flash` toegevoegd, omdat deze naar de opzoektabel `lookup` in het programmageheugen wijst.

Regel 37
*
dereferentie-operator

De dereferentie-operator * geeft de inhoud van pointer `ptr + 5*row + col`. Bij de toewijzing wordt de inhoud van het geheugen waar deze pointer naar wijst toegekend aan de uitgang `PORTE.OUT`.



Figuur 17.15 : **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 basis van iedere transistor is aangesloten op één van de aansluitingen van `PORTB`.

17.7 Een 4-digit 7-segmentdisplay aansturen

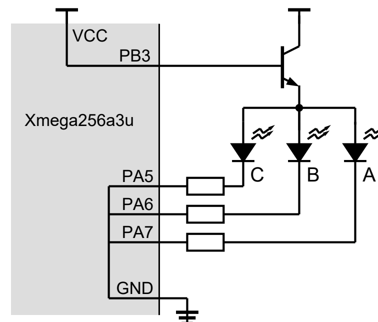
Een 7-segmentdisplay bestaat uit zeven leds die samen een patroon vormen waarmee cijfers en sommige karakters weergegeven kunnen worden. De segmenten (leds) hebben een rechthoekige vorm en worden aangeduid met de letters `A` tot en met `G`. Meestal heeft het display een achtste led voor een decimale punt (`DP`). De leds van de displays hebben een gemeenschappelijke anode of een gemeenschappelijke kathode.

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 17.15 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`.

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.

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 17.16 toont het elektrische 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 door de weerstanden. Elke led heeft een eigen afvoer (*sink*) via de microcontroller. De totale stroom die door `PORTA` en `PORT`B gaan is 100 mA. Iedere led mag zodoende maximaal 14 mA afvoeren.



Figuur 17.16 : Het elektrische gedrag van het 4-digit 7-segmentdisplay voor de situatie dat PB3 hoog en PA5, PA6 en PA7 laag zijn.

De displays worden één voor één 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_{active} dat een display actief is dus 5 ms.

De software voor de aansturing van een 4-digit 7-segmentdisplay

Voor het algoritme om een getal op het 4-digit 7-segmentdisplay te zetten moet alle 7-segmentdisplays achtereenvolgens even actief maken en het juiste cijfer op het display zetten:

```
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 modulusoperator (`%`). Zo geeft 6807 `%` 10 bijvoorbeeld het cijfer 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 17.8 is dit algoritme geïmplementeerd. Daarbij is — net als bij de codes voor de dotmatrix – een opzoektabel gebruikt voor de af te beelden waarden.

Net als in code 17.5 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.

Code 17.8 : Willekeurige getallen tonen op een 4-digit 7-segmentdisplay.

```
 1 #define F_CPU 2000000UL
 2
 3 #include <avr/io.h>
 4 #include <util/delay.h>
 5 #include <stdlib.h>
 6
 7 const uint8_t lookup[] = {
 8   0x03,  //  0000 0011  = 0
 9   0x9F,  //  1001 1111  = 1
10   0x25,  //  0010 0101  = 2
11   0x0D,  //  0000 1101  = 3
12   0x99,  //  1001 1001  = 4
13   0x49,  //  0100 1001  = 5
14   0x41,  //  0100 0001  = 6
15   0x1F,  //  0001 1111  = 7
16   0x01,  //  0000 0001  = 8
17   0x09   //  0000 1001  = 9
18 };
```

```
20 int main(void)
21 {
22   uint8_t   t     = 200;
23   uint16_t  value = 0;
24   uint16_t  tmp, digit;
25
26   PORTA.DIRSET = 0xFF;
27   PORTB.DIRSET = 0x0F;
28
29   while(1) {
30     if (t >= 200) {
31       value = rand() % 10000;
32       t = 0;
33     }
34
35     tmp = value;
36     for (int i=0; i<4; i++) {
37       digit     = tmp % 10;
38       tmp       = tmp / 10;
39       PORTA.OUT = lookup[digit];
40       PORTB.OUT = (PORTB.OUT & 0xF0) |
41                                  (_BV(i) & 0x0F);
42       _delay_ms(5);
43     }
44
45     t++;
46   }
47 }
```

Het getal dat `rand()` geeft, varieert van `0` tot en met `RAND_MAX`. De Xmega gebruikt 16-bits integers en daarom loopt het bereik van 0 tot en met 32767. Modulus 10000 zorgt er voor dat de waarde altijd maximaal 9999 is.

Omdat het resultaat van `rand()` van 0 tot en met 32767 loopt, is de uitkomst na bewerking met de modulus niet meer willekeurig. Waarden tot en met 2767 zullen vaker voorkomen.
Een alternatief is om in plaats van de modulusoperator de uitkomst van `rand()` te vermenigvuldigen met een schaalfactor. Als de schaalfactor 9999/32767 is, lopen de waarde van 0 tot en met 9999. Het headerbestand `stdlib.h` bevat ook een macro `RAND_MAX`, die de hoogste waarde van de randomfunctie definieert, zodat regel 31 ook vervangen kan worden door:

```
value = (uint32_t) 9999 * rand() / RAND_MAX;
```

Met deze berekening bevat `value` een willekeurige waarde, die in het bereik van 0 tot en met 9999 ligt.

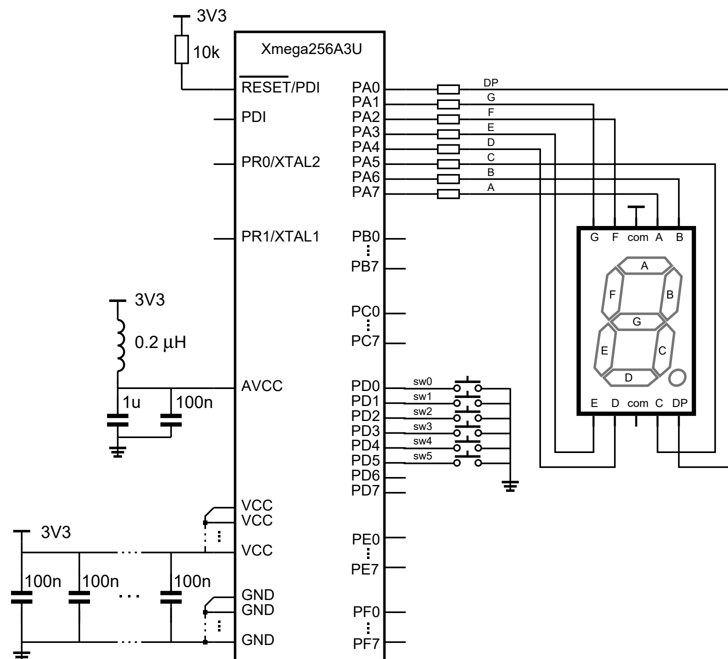
Een andere aspect is dat `rand` een pseudorandomgenerator is en deze altijd dezelfde reeks getallen geeft. Bij een microcontrollerprogramma is het vaak handiger om een timer/counter te gebruiken. Als de gebruiker van het programma op een knop drukt, zal de teller op dat moment een willekeurige waarde bevatten.

17.8 Het uitlezen van zes drukknoppen

Net als het aansturen van meerdere leds, zijn er bij het uitlezen van meerdere drukknoppen verschillende oplossingen mogelijk. Dit kan bijvoorbeeld met behulp van polling en met interrupts.

Het uitlezen van zes drukknoppen met behulp van polling

In de schakeling van figuur 17.17 zijn zes 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 zo gekozen dat de maximale stroom die de poort moet afvoeren ruim onder de 100 mA blijft. 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 17.17 : Schakeling met zes drukknoppen en een 7-segmentdisplay.

De applicatie voor de schakeling van figuur 17.17 zet op het display een 0, 1 of 5 als er respectievelijk op drukknop `PD0`, `PD1` of `PD5` wordt gedrukt.

Het programma staat in code 17.9. De waarden voor het 7-segmentdisplay staan in de opzoektabel `lookup`. 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 `button_pressed` leest eerst `PORTD.IN` en bewaart de geïnverteerde waarde van de zes bits in integer `x`. Als er op een knop gedrukt is, is één bit van `x` hoog. De **`for`**-lus gaat na op welke van de zes knoppen gedrukt is. De uitdrukking `x == _BV(i)` vergelijkt de waarde van `x` met respectievelijk 2^i. Als deze uitdrukking

Code 17.9 : Het uitlezen van zes drukknoppen met behulp van polling.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6
7  #define SW_MASK    0x3F   // 0011 1111
8  #define NUM_LEDS   6
9
10 const uint8_t lookup[] = {
11   0x03, 0x9F, 0x25, 0x0D, 0x99, 0x49, 0xFF};
12
13 int button_pressed(void)
14 {
15   uint8_t x;
16   int i, bitnumber;
17
18   x = ~(PORTD.IN) & SW_MASK;
19   bitnumber = -1;
20   for(i=0; i<NUM_LEDS; i++) {
21     if ( x == _BV(i) ) {
22       bitnumber = i;
23       break;
24     }
25   }
26   if (bitnumber < 0) return bitnumber;
27   _delay_ms(10);
28   while ( (PORTD.IN & SW_MASK) != SW_MASK );
29
30   return bitnumber;
31 }
32
33 int main(void)
34 {
35   int b;
36
37   PORTA.DIR    = 0xFF;
38   PORTA.OUT    = lookup[NUM_LEDS];
39
40   PORTD.DIRCLR    = SW_MASK;
41   PORTCFG.MPCMASK = SW_MASK;
42   PORTD.PIN0CTRL  = PORT_OPC_PULLUP_gc;
43
44   while (1) {
45     if ( (b = button_pressed()) >= 0 ) {
46       PORTA.OUT = lookup[b];
47     }
48   }
49 }
50
51
```

Code 17.10 : Het uitlezen van zes drukknoppen met behulp van een interrupt.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5  #include <avr/interrupt.h>
6
7  #define SW_MASK    0x3F
8  #define NUM_LEDS   6
9
10 const uint8_t lookup[] = {
11   0x03, 0x9F, 0x25, 0x0D, 0x99, 0x49, 0xFF};
12
13 ISR(PORTD_INT0_vect)
14 {
15   uint8_t x;
16   int i, bitnumber;
17
18   x = ~(PORTD.IN) & SW_MASK;
19   bitnumber = -1;
20   for(i=0; i<NUM_LEDS; i++) {
21     if ( x == _BV(i) ) {
22       bitnumber = i;
23       break;
24     }
25   }
26   if (bitnumber < 0) return;
27   _delay_ms(10);
28   while ( (PORTD.IN & SW_MASK) != SW_MASK );
29
30   PORTA.OUT = lookup[bitnumber];
31 }
32
33 int main(void)
34 {
35   PORTA.DIR  = 0xFF;
36   PORTA.OUT  = lookup[NUM_LEDS];
37
38   PORTD.DIRCLR     = SW_MASK;
39   PORTD.INT0MASK   = SW_MASK;
40   PORTCFG.MPCMASK  = SW_MASK;
41   PORTD.PIN0CTRL   = PORT_OPC_PULLUP_gc |
42                      PORT_ISC_FALLING_gc;
43   PORTD.INTCTRL   |= PORT_INT0LVL_LO_gc;
44
45   PMIC.CTRL |= PMIC_LOLVLEN_bm;
46   sei();
47
48   while (1) {
49     asm volatile ("nop");
50   }
51 }
```

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. In het geval er op geen enkele knop gedrukt is, is het bitnummer eveneens −1.
Als `bitnumber` negatief is, geeft de functie `button_pressed` −1 terug. Als `bitnumber` positief is, wacht de functie eerst 10 ms totdat de knop losgelaten is en geeft de functie het gevonden bitnummer terug.

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.

Bij de initialisatie van poort D is gebruik gemaakt van *multi-pin configuration*. Bij alle zes ingangen moet de interne pullupweerstand worden aangezet. Iedere ingang heeft zijn eigen PIN*n*CTRL-register. Dat kan gerealiseerd worden met de volgende zes toekenningen:

```
PORTD.PIN0CTRL  = PORT_OPC_PULLUP_gc;
PORTD.PIN1CTRL  = PORT_OPC_PULLUP_gc;
PORTD.PIN2CTRL  = PORT_OPC_PULLUP_gc;
PORTD.PIN3CTRL  = PORT_OPC_PULLUP_gc;
PORTD.PIN4CTRL  = PORT_OPC_PULLUP_gc;
PORTD.PIN5CTRL  = PORT_OPC_PULLUP_gc;
```

In code 17.9 is dit op regel 41 opgelost door de betreffende zes bits van het `MPCMASK`, *multi-pin configuration mask*, hoog te maken en daarna voor één van de registers PIN*n*CTRL de pullup aan te zetten:

```
PORTCFG.MPCMASK = SW_MASK;
PORTD.PIN0CTRL  = PORT_OPC_PULLUP_gc;
```

Het `MPCMASK`-register zorgt er voor dat de zes PIN*n*CTRL-registers gelijktijdig worden aangepast.

Voor alle bits uit het `MPCMASK`-register die hoog zijn, veranderen de overeenkomstige PIN*n*CTRL-registers als een van deze registers wordt aangepast. Direct na de toekenning aan één van de PIN*n*CTRL-registers wordt het `MPCMASK`-register automatisch leeggemaakt. In dit voorbeeld zijn de zes minst significante bits van het `MPCMASK`-register hoog en krijgen de zes ingangen PD0 tot en met PD5 een pullup.

Het uitlezen van zes knoppen met externe interrupt

De Xmega kent per poort twee interrupts. De zes ingangen uit figuur 17.17 kunnen dus niet alle zes ingangen een eigen interrupt krijgen. De ingangen kunnen echter wel alle zes een externe interrupt 0 of 1 veroorzaken. Het `INT0MASK`-register geeft de ingangen aan waar externe interrupt 0 gevoelig voor is.

In code 17.10 staat het programma dat de externe interrupt 0 van poort D gebruikt. Op regel 43 is externe interrupt 0 ingesteld op het lage niveau. Regel 39 maakt de laagste zes bits van het `INT0MASK`-register hoog en daarna krijgen de zes aansluitingen PD0 tot en met PD5 een pullup en wordt de interrupt gevoelig voor deze zes ingangen.

Op regel 40 en 41 krijgen de zes ingangen met behulp van de *multi-pin configuration*-methode een pullup en zijn ze gevoelig voor de neergaande flank.

In plaats van een functie `button_pressed` heeft het programma van code 17.10 een interruptfunctie `ISR(PORTD_INT0_vect)`. Deze functie lijkt op `button_pressed`, maar heeft net als alle andere interruptfuncties geen retourwaarde. Daarom is de toekenning aan het uitgangsregister van `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 het gedurende deze tijd niet op andere *low-level* interrupts reageren. Dit probleem kan worden opgelost door voor de tijdvertraging een timer te gebruiken, zoals dat bij code 16.5 uit paragraaf 16.6 is gedaan.

17.9 Conclusie

Bij het gebruik van een display met meerdere leds is het verstandig om goed naar de aansturing te kijken. Er zijn heel veel soorten leddrivers te koop. Gebruik bij de software voor de displays opzoektabellen voor het vastleggen van de informatie, die op het display getoond moet worden.

De aansturing van de displays en het uitlezen van meerdere knoppen kan softwarematig op veel verschillende manieren gedaan worden. Afhankelijk van de situatie zal de ene oplossing meer geschikt zijn dan de andere. Het is altijd verstandig om verschillende alternatieven te bedenken, de voor- en nadelen te bestuderen en de gemaakte keuzes goed te documenteren.

18

Liquid Crystal Display

Doelstelling

In dit hoofdstuk leer je hoe een karaktergeoriënteerd LCD is opgebouwd, wat HD44780 compatibel betekent, de verschillende manieren waarop je het LCD kunt gebruiken en hoe je een HD44780-compatibel display met de Xmega aanstuurt.

Onderwerpen

De behandelde onderwerpen zijn:

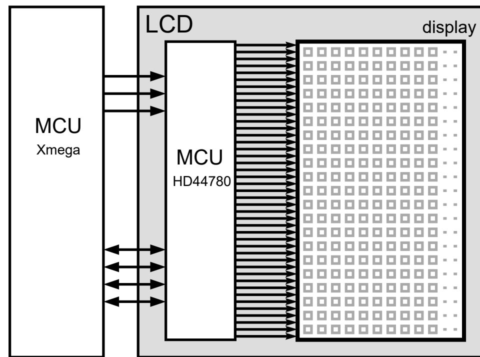
- De instelling van het contrast van het LCD.
- Het instellen van de achtergrondverlichting.
- De communicatie met het LCD volgens het HD44780-protocol. Daarbij komt aan de orde: de timing bij de HD44780, de *busy flag*, de aansturing van de geheugens van de HD44780, de positionering van tekst op het LCD.
- Een LCD-bibliotheek.
- De weergave van gehele getallen met behulp van `utoa` en `ultoa`.
- Geformatteerd afdrukken met behulp van `dtostrf` en `sprintf`.
- Het afdrukken van gebroken getallen.

De voorbeelden tonen verschillende mogelijkheden van het LCD en een aantal methoden om tekst en getallen op het LCD weer te geven:

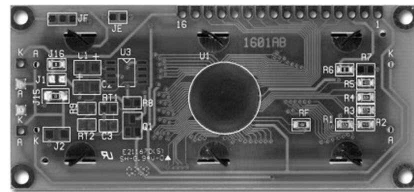
- Met 8-bit modus en een *worst case* tijdvertraging.
- Het bewegen van tekst in 8-bit modus.
- Het geformatteerd weergeven van getallen in 4-bit modus.
- Het gebruik van de LCD-bibliotheek.
- Het geformatteerd afdrukken met de functie `sprintf`.
- Het geformatteerd en ongeformatteerd afdrukken van gebroken getallen.

In het vorig hoofdstuk is de aansturing van displays met meerdere leds besproken. Met de $5x7$-, $5x8$- of $16x16$-ledarrays kunnen ook grotere displays worden samengesteld. Lichtkranten en beeldschermen met duizenden leds hebben intern speciale leddrivers en processoren die de aansturing regelen.

Voor toepassingen met microcontrollers bestaan er relatief kleine grafische en karaktergeoriënteerde displays. Deze displays bevatten een microcontroller die de pixels van het scherm aanstuurt, zie figuur 18.1. De microcontroller van het display moet met de microcontroller van de applicatie communiceren. Hiervoor zijn



Figuur 18.1 : **Communicatie met HD44780.**
De Xmega communiceert met de HD44780 die de pixels aanstuurt.



Figuur 18.2 : **Achterkant LCD met HD44780.**
De HD44780 zit onder de zwarte schijf.

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.

Dit hoofdstuk bespreekt de karaktergeoriënteerde displays met een HD44780-controller. Dat zijn 1-, 2- of 4-regelige displays met 16, 20 of 40 karakters per regel. Het maximale aantal karakters per regel is 40.

De eerste stap die je doet, als je met een display aan de slag gaat, is dat uitzoekt welke microcontroller het display bevat en welk protocol er gebruikt wordt. Er bestaan heel veel verschillende uitvoeringen. Sommige displays zijn compatibel en andere niet. Displays met een SED1278 van Epson, KS0066 van Samsung, ST7066 van Sitronix en SPLC780A1 van Sunplus zijn compatibel met de HD44780.

Paragraaf 18.2 bespreekt de verschillende aspecten van HD44780-compatibele displays. De aansluiting van een LCD met een Xmega of een andere microcontroller met een lage voedingsspanning vereist speciale aandacht. Omdat bij een verkeerd gebruik de Xmega beschadigd kan worden, staat in de volgende paragraaf hoe het LCD met de Xmega verbonden kan worden.

18.1 Het aansluiten van een HD44780 op de Xmega

De aanbevolen voedingsspanning van de Xmega ligt tussen de 1,6 V en 3,6 V. De voorbeelden in dit boek gebruiken altijd 3,3 V. De meeste HD44780-compatibele displays werken op 5 V, maar functioneren wel correct op ingangssignalen tussen 2,7 V en 5,5 V. Bij lagere signaalniveaus reageert de HD44780 wel langzamer.
Een signaal 5 V mag echter *nooit* op de ingang van de Xmega worden aangesloten. De ingangen van de Xmega zijn *niet* 5 V tolerant. De uitgangssignalen van een 5 V-LCD zijn 5 V. Deze signalen mogen dus *nooit* op de ingang van een Xmega worden aangesloten.

Er zijn twee opties: een 3,3 V display gebruiken of een 5 V display en er tegelijkertijd voor zorgen dat er geen 5 V-signalen van het display naar de Xmega gaan. Dit laatste kan op verschillende manieren worden bereikt. De HD44780 kent vier verschillende modi:

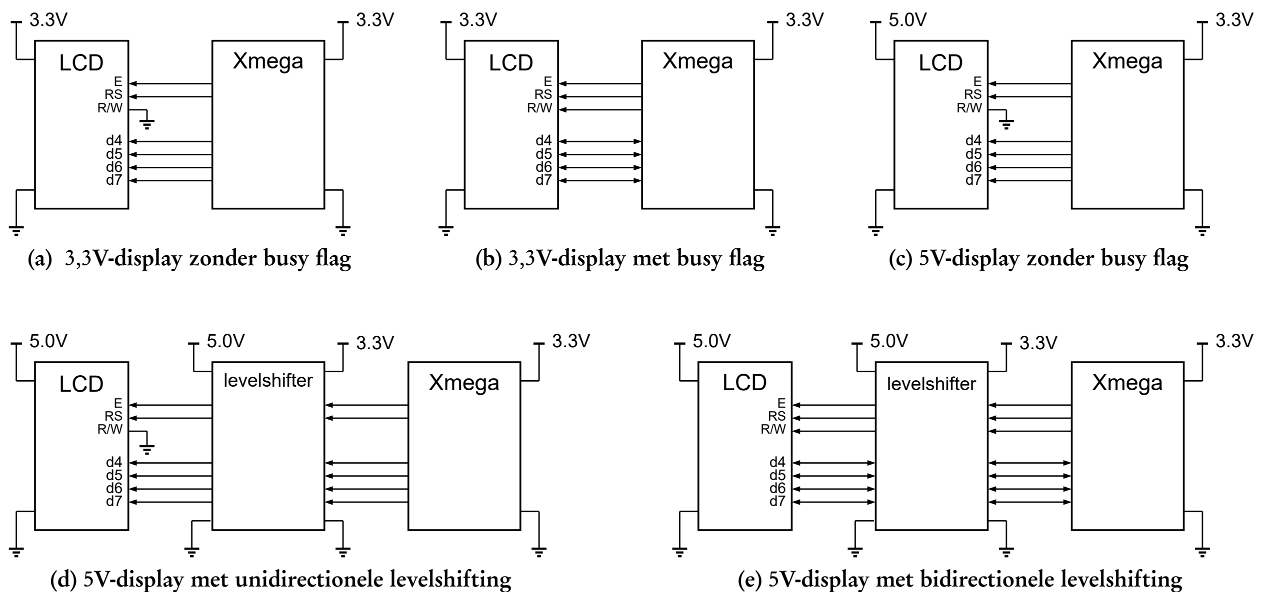
- de 4-bits modus zonder *busy flag*,
- de 8-bits modus zonder *busy flag*,
- de 4-bits modus met *busy flag*,
- de 8-bits modus met *busy flag*.

Deze instellingen worden in de volgende paragraaf besproken. Bij de modi met de *busy flag* wordt er ook informatie van het display naar de Xmega gestuurd. Zonder speciale voorzieningen mogen de modi met de *busy flag* dus niet worden gebruikt. Bij de methoden zonder *busy flag* zijn er alleen signalen van de Xmega naar het display. De meeste HD44780-compatibele displays hebben een voedingsspanning van 5 V nodig en reageren correct op signalen van 3,3 V. Wel zijn de vertragingstijden dan groter.

Om van een hoge naar een lage spanning te gaan, wordt vaak een 74AHC125 of een CD4050 gebruikt.

De TXB0104 en TXB0108 van Texas Instruments zijn respectievelijk 4-bits en 8-bits voorbeelden van bidirectionele levelshifters, die geschikt zijn om bij een LCD te gebruiken.

Een alternatief is om een zogenoemde *levelshifter* te gebruiken. Er bestaan unidirectionele en bidirectionele levelshifters. Unidirectionele levelshifters kunnen signalen omzetten van een hogere naar een lagere spanning of van een lagere naar een hogere spanning. Bidirectionele levelshifters converteren een bidirectioneel signaal van een laag naar hoog niveau en omgekeerd.



(a) 3,3V-display zonder busy flag

(b) 3,3V-display met busy flag

(c) 5V-display zonder busy flag

(d) 5V-display met unidirectionele levelshifting

(e) 5V-display met bidirectionele levelshifting

Figuur 18.3 : Vijf methoden om een HD44780-LCD met de 4-bit modus aan een Xmega aan te sluiten. Bij de methoden (a) en (b) heeft het display dezelfde voedingsspanning als de Xmega. Bij de methoden (d) en (e) is een levelshifter tussen de Xmega en het LCD geplaatst. Methode (c) gebruikt geen levelshifter; dat mag alleen als de `R/W` laag is.

In figuur 18.3 staan vijf methoden om een HD44780-compatibel display aan te sluiten op de Xmega. Getekend is de 4-bits modus. De 8-bits modus is identiek, alleen zijn er dan acht datalijnen in plaats van vier.

De schema's van figuur 18.3 geven niet alle aansluitingen van het LCD. De volgende paragraaf bespreekt de HD44780-compatibele displays in detail. Het probleem met de mismatch in spanningen is hier als eerste speciaal toegelicht, om te benadrukken dat de ontwerper hiermee rekening moet houden en dat er verschillende oplossingen mogelijk zijn.

18.2 Het karaktergeoriënteerde display op basis van HD44780

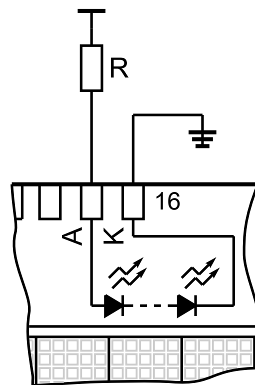
Er bestaan displays met twee verschillende karakterformaten, namelijk met karakters van 5 × 8 pixels en van 5 × 10 pixels. De meeste displays hebben karakters met 5 × 8 pixels. De HD44780 ondersteunt bij beide displays.

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: één voor de bovenste twee regels en één voor de onderste twee regels.

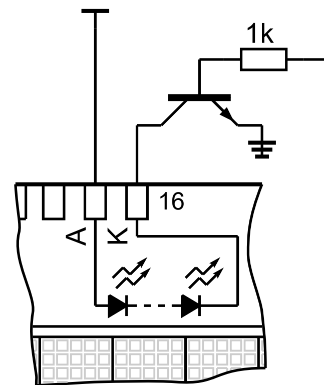
Achtergrondverlichting (*back light*)

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.

Een display met een HD44780 heeft veertien of zestien aansluitingen. Een display met achtergrondverlichting (*back light*) 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.



Figuur 18.4 : Achtergrondverlichting LCD altijd aan.

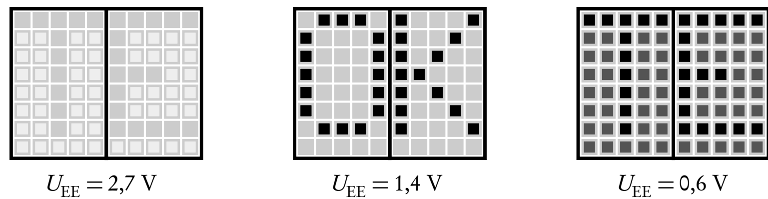


Figuur 18.5 : Achtergrondverlichting wordt met transistor geschakeld vanuit microcontroller.

De waarde van de weerstand in figuur 18.5 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.

De achtergrondverlichting bestaat uit een of meer in serie geschakelde leds. In figuur 18.4 is de anode (A) met de voeding en de kathode (K) met de referentie verbonden. Er is een weerstand nodig om de stroom door de leds te beperken. De waarde van de weerstand R hangt af van de eigenschappen van de leds en de voedingsspanning.

Figuur 18.5 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.

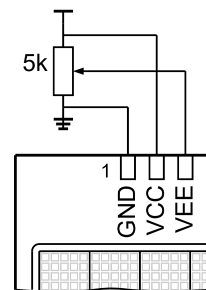


Figuur 18.6 : Het effect op het contrast bij een LCD. Links is de contrastspanning te hoog, in het midden precies goed en rechts te laag. De waarden van de contrastspanning zijn voor ieder display anders. In dit voorbeeld gaat het om een 5 V-display.

Contrast van het display

De contrastspanning is voor elke type display anders en hangt sterk af van het omgevingslicht en of de achtergrondverlichting aan staat.

Het LCD wordt gevoed via de pinnen VCC en GND. 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 18.6 laat dit voor verschillende spanningen zien.



Figuur 18.7 : De voeding en de regeling van het contrast van het LCD. Met de potmeter is het contrast van het LCD instelbaar.

De potmeter in figuur 18.7 regelt het contrast. De spanning VEE is dan instelbaar tussen de voedingsspanning VCC en GND.

Communicatie met HD44780

Voor de communicatie zijn elf aansluitingen beschikbaar: drie besturingslijnen en acht datalijnen. Tabel 18.1 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 is `R/W` met de referentie verbonden.

Tabel 18.1 : 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 modus
8	D1	Data	niet gebruikt in 4-bit modus
9	D2	Data	niet gebruikt in 4-bit modus
10	D3	Data	niet gebruikt in 4-bit modus
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)

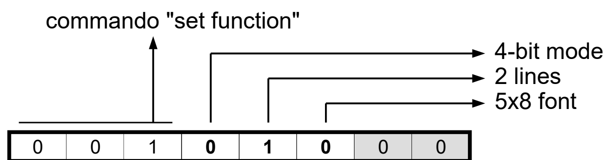
Er zijn acht datalijnen. De informatie kan verstuurd worden in een 8-bit modus of in een 4-bit modus. Deze modus wordt ingesteld bij het initialiseren van het LCD. In de 8-bit modus wordt de informatie in bytes overgestuurd. Signaal D7 bevat de meest significante bit en D0 de minst significante bit.

Bij de 4-bit modus 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 het LCD blijven open. De datalijnen D7, 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.

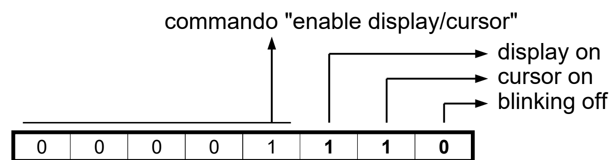
Een *nibble* is een groep van vier bits.

Een *byte* is een groep van acht bits. Een *byte* bestaat uit twee *nibbles*.

Tabel 18.2 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 18.8 : HD44780-instructie 0x28.



Figuur 18.9 : HD44780-instructie 0x0E.

Figuur 18.8 toont de instructie 0x28 die de 4-bit modus selecteert en die aangeeft dat het display twee regels heeft en dat er een 5×8-karakterset is. Figuur 18.9 laat zien dat met de instructie 0x0E het display en de cursor aangezet worden en dat de cursor niet knippert.

Tabel 18.2 : 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		

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.

Tabel 18.3 : 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 18.4 : 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

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 18.10 geeft het tijdsdiagram voor het schrijven van informatie. Het signaal R/W moet laag zijn en het signaal RS moet hoog zijn bij het versturen van data (tekst) en laag zijn 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 18.3 en 18.4 staan een aantal tijdskarakteristieken uit de datasheet. Voor verschillende displays kunnen deze tijden anders zijn.

Voorbeelden op het internet maken het enable-signaal E vaak één instructie hoog en bij de volgende instructie direct weer laag:

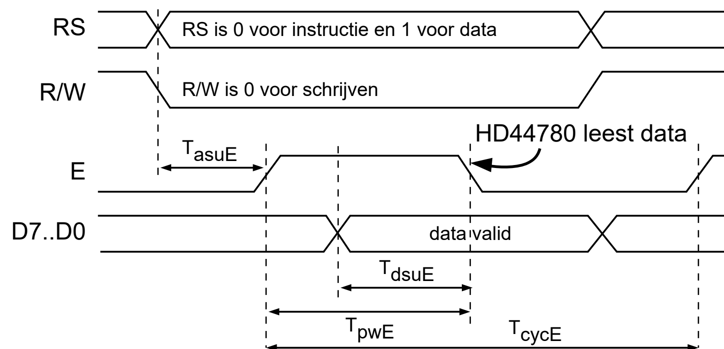
```
PORTD.OUT = 'x';           // databyte is 'x'
PORTE.OUTSET = PIN5_bm;    // bit 5 port E (signal E) high
PORTE.OUTCLR = PIN5_bm;    // bit 5 port E (signal E) low
```

Het effect is dat `E` dan twee klokslagen hoog is. Dit is prima voor klokfrequenties lager dan 4 MHz, maar 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 van 2,7 V. Een oplossing voor dit probleem is het toevoegen van een extra tijdvertraging:

De exacte tijdvertraging hangt bij `_delay_us` af van de klokfrequentie. Bij een klok van 2 MHz komt de gewenste vertraging overeen met één klokslag. Het minimale aantal klokslagen van `_delay_us` is drie. De werkelijke tijdvertraging is dan 1,5 μs. Bij een klok van 32 MHz komt de gewenste vertraging overeen met 16 klokslagen en is de vertraging wel 0,5 μs.

```
PORTD.OUT = 'x';           // databyte is 'x'
PORTE.OUTSET = PIN5_bm;    // bit 5 port E (signal E) high
_delay_us(0.5);
PORTE.OUTCLR = PIN5_bm;    // bit 5 port E (signal E) low
```

De extra tijdvertraging is 0.5 μs en is groter dan strikt genomen nodig is, maar dit geeft een veilige marge voor displays met een lage spanning.



Figuur 18.10 : 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 zijn voor het schrijven van instructies. Nadat `E` laag wordt, leest de HD44780 de `D7..D0`.

Tabel 18.5 : Tijdvertragingen van de HD44780.
C staat voor cursor en D voor display.

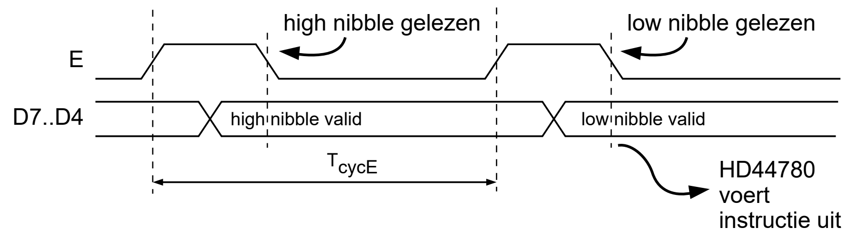
Command	Delay
clear display	1,52 ms
return home	1,52 ms
move C	37 μs
enable D/C	37 μs
shift D/C	37 μs
function set	37 μs
move C CGRAM	37 μs
move C display	37 μs
write character	37 μs
read busy flag	37 μs
read character	37 μs

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 18.5 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 μs gewacht worden.

```
PORTD = 'x';               // databyte is 'x'
PORTE.OUTSET = PIN5_bm;    // bit 5 port E (signal E) high
_delay_us(0.5);            // delay enable
PORTE.OUTCLR = PIN5_bm;    // bit 5 port E (signal E) low
_delay_us(50);             // delay display
```

De vertraging van 50 μs is ruim voldoende bij de meeste displays. Dit is echter niet genoeg bij het leegmaken van het display en bij de *return home*-opdracht.

In de 4-bit modus 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 18.11 laat dit zien. De twee pulsen moeten wel minimaal T_{cycE} (= 1000 ns) uit elkaar liggen.



Figuur 18.11 : Het schrijven van een instructie in 4-bit modus. De HD44780 leest de *nibbles* nadat `E` laag gemaakt is.

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 18.5 om de signalen op het juiste moment 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 één 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.
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.

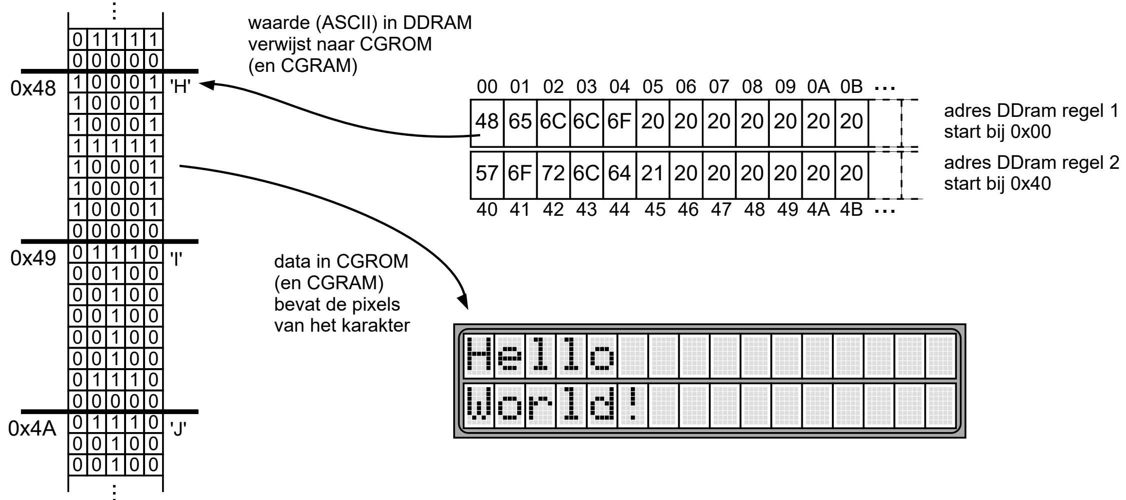
Het maximaal aantal karakters bij een HD44780 display is tachtig.

In figuur 18.12 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 in dit voorbeeld 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 I staat meer informatie over ASCII en in tabel I.1 staat een overzicht van alle 7-bits ASCII-waarden.

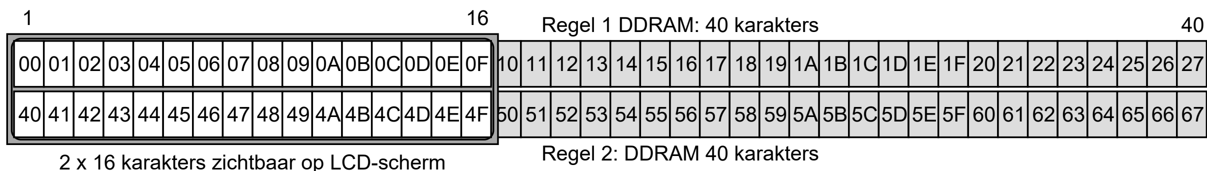
De datasheet geeft een compleet overzicht van beide karaktersets. Veel programmeurs gebruiken alleen de 7-bits ASCII-waarden, die in beide sets liggen tussen `0x20` tot en met `0x7D`.

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 karaktersets.

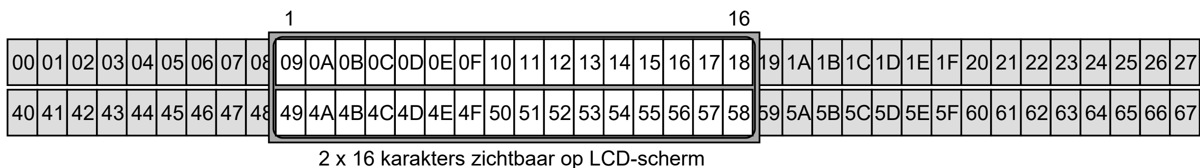


Figuur 18.12 : **Het CGROM en het DDRAM.** Het DDRAM bevat de ASCII-waarden van de af te beelden tekst. Het CGROM (en het CGRAM) bevat de pixelinformatie van de karakterset. Het adres van een karakter komt overeen met de betreffende ASCII-waarde.

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 ($\sqrt{}$). De karakterset van de HD44780U-A02 komt voor een groot deel overeen met de ASCII-ISO8859. Dit zijn de standaard ASCII-tekens van `0x20` tot en met `0x7D` aangevuld met veel Europese karakters (ö, Æ, Å). Een deel van de ISO8859-symbolen is vervangen door Griekse en Cyrillische letters.

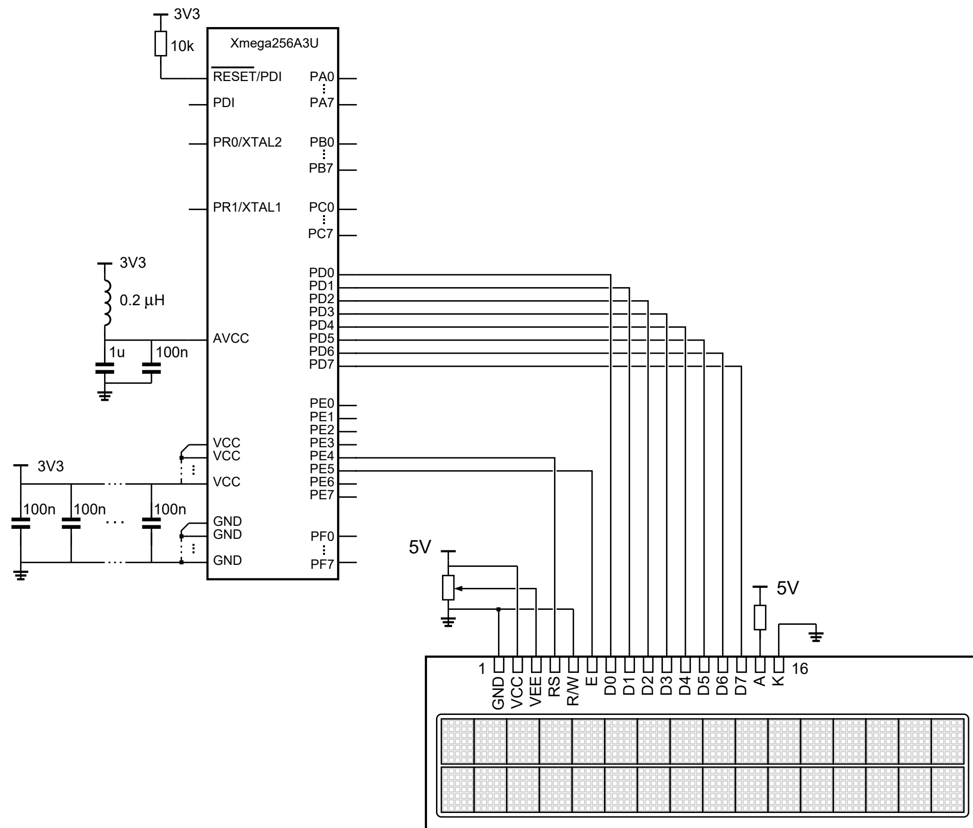


Figuur 18.13 : De adressering van het geheugen en de relatie met karakters die zichtbaar zijn bij een 2x16 display.



Figuur 18.14 : **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 18.13 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 18.14 is het LCD negen posities naar rechts geschoven ten opzichte van het DDRAM. De verschuiving geldt altijd voor beide regels.



Figuur 18.15 : **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.

18.3 Toepassing LCD met 8-bit modus en tijdvertraging

Figuur 18.15 geeft een compleet schema voor de 8-bit modus. De datalijnen zijn aangesloten op poort D van de microcontroller. Pin 4 van poort E is aangesloten op de `RS`-pin en pin 5 van poort E 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 de referentie.

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 gestuurd moeten worden. Figuur 18.16 geeft de initialisatie voor de 8-bit modus en figuur 18.17 die voor de 4-bit modus. De acties met de donkergrijze achtergrond zijn verplicht voor de initialisatie. De diverse commando's hebben een eigen verwerkingstijd, zie ook tabel 18.5. Het volgende commando mag pas nadat deze tijd is verstreken gegeven worden, anders wordt het genegeerd. Dit voorbeeld gebruikt *worst case* tijdvertragingen.

Het programma staat in code 18.1 en schrijft met de functie `lcd_write` informatie naar het display. De functie heeft twee ingangsvariabelen. Ingang `d` is het karakter dat afgebeeld wordt of het commando dat gegeven wordt. Ingang `rs` geeft aan of er

Code 18.1 : Aansturing LCD met acht datalijnen en een *worst case* tijdvertraging.

```
#define F_CPU   2000000UL

#include <avr/io.h>
#include <util/delay.h>

#define lcd_cmd(d)    lcd_write((d), 0)
#define lcd_putc(d)   lcd_write((d), 1)

void lcd_write(char d, uint8_t rs)
{
  if (rs) {
    PORTE.OUTSET = PIN4_bm;  // RS high (data)
  } else {
    PORTE.OUTCLR = PIN4_bm;  // RS low (command)
  }

  PORTD.OUT    = d;          // assign data
  PORTE.OUTSET = PIN5_bm;    // make E high
  _delay_us(0.5);
  PORTE.OUTCLR = PIN5_bm;    // make E low
  _delay_us(50);             // wait 50 us
}

void lcd_init(void)
{
  PORTD.DIRSET = 0xFF;             // port D: 8-bit data
  PORTE.DIRSET = PIN5_bm|PIN4_bm;  // pin 4 and 5 port E: RS en E

  _delay_ms(50);
  lcd_cmd(0x30);
  _delay_ms(5);
  lcd_cmd(0x30);
  _delay_us(50);
  lcd_cmd(0x30);
  lcd_cmd(0x38);     // 8-bits, 2 lines, 5x8 font
  lcd_cmd(0x0C);     // display on, cursor off, blink off
  lcd_cmd(0x06);     // move cursor right
  lcd_cmd(0x01);     // clear display
  _delay_us(1500);   // extra wait after clear_display
}

int main(void)
{
  lcd_init();           // initialization LCD

  lcd_putc('L');        // write text
  lcd_putc('C');
  lcd_putc('D');

  while (1) {
    asm volatile ("nop");
  }
}
```

Instructie	Omschrijving	Vertraging	
Power on		45 ms	Vertragingen altijd nodig
00 0011----		4,1 ms	
00 0011----		100 µs	
00 0011----		37 µs	
00 001110--	8-bit mode, 2 lines, 5x8 font	37 µs	Vertragingen niet per se nodig. Kan ook met busy flag worden opgelost
00 00001111	display on, cursor on, blink on	37 µs	
00 00000001	clear display	1,52 ms	
00 00000110	auto-increment, no shift	37 µs	
10 01001100	write 'L'	37 µs	
10 01000011	write 'C'	37 µs	
10 01000100	write 'D'	37 µs	
⋮	⋮	⋮	⋮

Figuur 18.16 : **Initialisatie bij de 8-bit modus.**

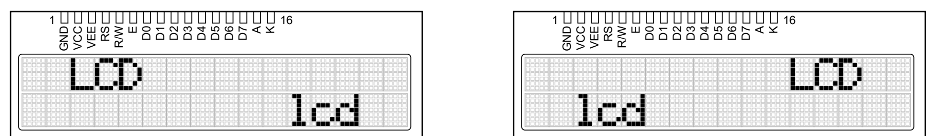
Instructie	Omschrijving	Vertraging	
Power on		45 ms	Vertragingen altijd nodig
00 0011----		4,1 ms	
00 0011----		100 µs	
00 0011----		37 µs	
00 0010----	4-bit mode	37 µs	Vertragingen niet per se nodig. Kan ook met busy flag worden opgelost
00 0010 00 1000	4-bit mode, 2 lines, 5x8 font	37 µs	
00 0000 00 1111	display on, cursor on, blink on	37 µs	
00 0000 00 0001	clear display	1,52 ms	
00 0000 00 0110	auto-increment, no shift	37 µs	
10 0100 10 1100	write 'L'	37 µs	
10 0100 10 0011	write 'C'	37 µs	
10 0100 10 0100	write 'D'	37 µs	
⋮	⋮	⋮	⋮

Figuur 18.17 : **Initialisatie bij de 4-bit modus.**

een instructie of een karakter naar het display wordt gestuurd. Vervolgens wordt het enable-signaal minimaal 0,5μs hoog gemaakt en wordt er 50 ns gewacht.
Met de functie `lcd_write` zijn twee macro's gedefinieerd: een macro `lcd_cmd`, die een instructie naar het display stuurt, en een macro `lcd_putc`, die een karakter schrijft.

Op regel 24 van dit voorbeeld staat een functie `lcd_init` die het LCD initialiseert. Deze functie definieert eerst poort D en de pinnen 4 en 5 van poort E als uitgang. Vervolgens initialiseert deze functie het LCD in de 8-bit modus volgens het diagram van figuur 18.16.

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.



Figuur 18.18 : **Twee toestanden van het display bij code 18.2.** Links voor `i=2` en rechts voor `i=11`.

18.4 Toepassing met bewegende tekst

Het hoofdprogramma uit code 18.1 zet met `lcd_putc` drie keer een karakter op het scherm. Code 18.2 gebruikt een functie `lcd_puts` 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 en met 13. 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.

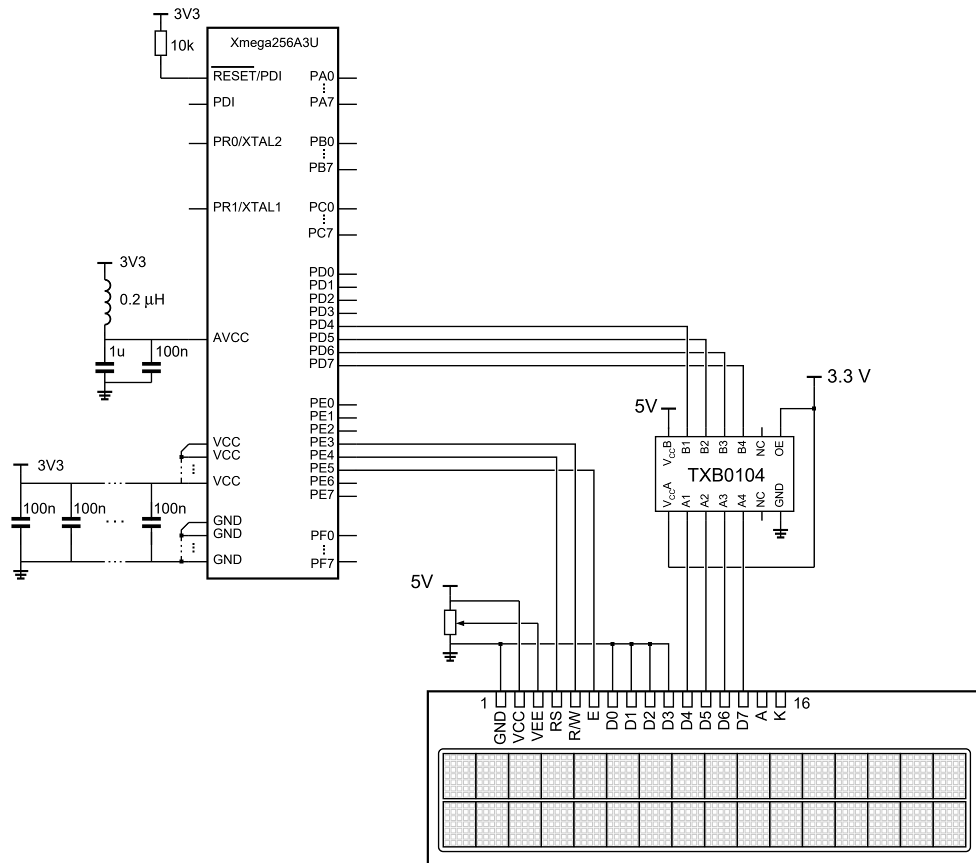
De tekst op regel 1 wordt rechts aangevuld (lcd_cmd(0x06)) en de tekst op regel 2 wordt links (lcd_cmd(0x04)) aangevuld. Het effect is dat de tekst op regel 1 van links naar rechts beweegt en die op regel 2 van rechts naar links beweegt. Figuur 18.18 toont het resultaat van de code 18.2 voor de waarden i=2 en i=11.

Code 18.2 : Aansturing LCD in 8-bit modus met bewegende tekst.

```
42 //  Regel 1 tot en met 40 zijn identiek met die van code 18.1.
43
44 void lcd_puts(char *s)
45 {
46   while (*s) {
47     lcd_putc(*s++);
48   }
49 }
50
51 int main(void)
52 {
53   uint8_t i=0;
54
55   lcd_init();
56
57   while (1) {
58     lcd_cmd(0x01);            // clear display
59     _delay_us(1500);
60     lcd_cmd(0x6);             // cursor direction right
61     lcd_cmd(1<<7|i);          // move to DDRAM location i
62     lcd_puts("LCD");
63     lcd_cmd(0x4);             // cursor direction left
64     lcd_cmd(1<<7|(0x4F-i));   // move to DDRAM location 0x4F-i
65     lcd_puts("dcl");
66     _delay_ms(1000);
67     if (++i==14) i=0;
68   }
69 }
```

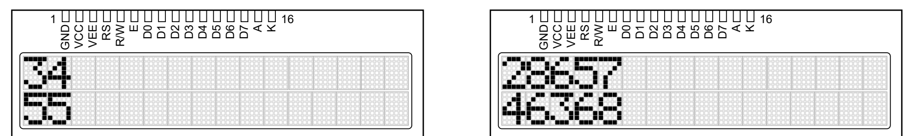
18.5 Toepassing LCD met 4-bit modus en busy flag

Deze paragraaf bespreekt de aansturing van het display in de 4-bit modus. Het programma leest de *busy flag* en controleert of het volgende commando verstuurd kan worden. Het schema staat in figuur 18.19 en bevat vier datalijnen en drie controllijnen. De besturingssignalen R/W, RS en E zijn verbonden met respectievelijk pin 3, pin 4 en pin 5 van poort E. De datalijnen D7, D6, D5 en D4 zijn verbonden met de pinnen 7, 6, 5 en 4 van poort D. De kathode en anode van de achtergrondverlichting zijn niet aangesloten.



Figuur 18.19 : 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 18.3 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 10.1. Figuur 18.20 toont twee toestanden van het display.



Figuur 18.20 : Twee schermafdrukken van de simulatie van code 18.3. 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 18.3 regel 8-23

Met **`#define`**'s worden zes macro's gedefinieerd, die de besturingssignalen `RS`, `R/W` en `E` hoog of laag maken. De twee macro's `lcd_cmd` en `lcd_putc` sturen respectievelijk een instructie of een karakter naar het display en maken gebruik van de functie `lcd_write`. De macro `set_rs` geeft `RS` de juiste waarde, de macro `lcd_e` genereert een enable-puls en `lcd_clear` maakt het display leeg.

Code 18.3 : Weergeven van getallen van Fibonacci op LCD met 4-bit modus.

```
1   #define F_CPU   2000000UL
2
3   #include <avr/io.h>
4   #include <util/delay.h>
5   #include <stdlib.h>
6   #include <limits.h>
7
8   #define RShigh()    PORTE.OUTSET = PIN4_bm
9   #define RSlow()     PORTE.OUTCLR = PIN4_bm
10  #define RWhigh()    PORTE.OUTSET = PIN3_bm
11  #define RWlow()     PORTE.OUTCLR = PIN3_bm
12  #define Ehigh()     PORTE.OUTSET = PIN5_bm
13  #define Elow()      PORTE.OUTCLR = PIN5_bm
14
15  #define lcd_cmd(d)  (lcd_write((d), 0))
16  #define lcd_putc(d) (lcd_write((d), 1))
17  #define lcd_rs(rs)  ((rs==1)? \
18                      (RShigh()):(RSlow()))
19  #define lcd_e()     (Ehigh()); \
20                      _delay_us(1); \
21                      (Elow())
22  #define lcd_clear() lcd_write(0x01, 0); \
23                      _delay_us(1500)
24
25  void lcd_write (uint8_t b, uint8_t rs)
26  {
27    uint8_t x;
28
29    PORTD.DIRCLR  = 0xF0;     // read mode
30    RWhigh();
31    RSlow();
32    do {
33      Ehigh();
34      _delay_us(0.5);
35      x = PORTD.IN & 0xF0;
36      Elow();
37      _delay_us(0.5);
38      Ehigh();
39      _delay_us(0.5);
40      x |= (PORTD.IN & 0xF0) >> 4;
41      Elow();
42    } while (x & 0x80);
43
44    PORTD.DIRSET  = 0xF0;    // write mode
45    RWlow();
46    lcd_rs(rs);
47    PORTD.OUT = (PORTD.OUT & 0x0F) |
48                                (b & 0xF0);
49    lcd_e();
50    PORTD.OUT = (PORTD.OUT & 0x0F) |
51                                  (b << 4);
52    lcd_e();
53  }
54
55  void lcd_init(void)
56  {
57    PORTD.DIRSET = 0xF0;          // PC7..PC4 : data
58    PORTE.DIRSET = PIN3_bm|PIN4_bm|PIN5_bm;//RW,RS,E
59
60    RWlow();
61    _delay_ms(50);
62    lcd_rs(0);
63    PORTD.OUT = (PORTD.OUT & 0x0F) | 0x30;
64    lcd_e();
65    _delay_ms(4);
66    PORTD.OUT = (PORTD.OUT & 0x0F) | 0x30;
67    lcd_e();
68    _delay_ms(100);
69    PORTD.OUT = (PORTD.OUT & 0x0F) | 0x20; // 4-bit
70    lcd_e();
71    lcd_write(0x28,0);  // 4-bit, 2 lines, 5x8 font
72    lcd_write(0x0C,0);  // disp on, curs off, bl off
73    lcd_write(0x06,0);  // move right
74  }
75
76  void lcd_puts(char *s)
77  {
78    while ( *s ) {
79      lcd_putc(*s++);
80    }
81  }
82
83  int main(void)
84  {
85    uint16_t h, f1 = 1, f2 = 1;
86    char buffer[6];
87
88    lcd_init();
89
90    while (1) {
91      lcd_clear();
92      lcd_cmd(1<<7|0x0);    // start at line 1
93      utoa(f1, buffer, 10);
94      lcd_puts(buffer);
95      lcd_cmd(1<<7|0x40);   // start at line 2
96      utoa(f2, buffer, 10);
97      lcd_puts(buffer);
98      _delay_ms(1000);          // wait
99      if (f2 > (UINT_MAX/2)) {  // calc. next fib.
100       f1 = 1;
101       f2 = 1;
102     } else {
103       h  = f2;
104       f2 = f1 + f2;
105       f1 = h;
106     }
107   }
108 }
```

Uitleg code 18.3 regel 25-54
`lcd_write()`

De ingangsparameter `d` van de functie `lcd_write` bevat de databyte die naar het display geschreven wordt en de parameter `rs` geeft aan of het een instructie (0) is of een karakter (1). 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, zijn de datalijnen aanvankelijk ingang (`PORTD.DIRCLR = 0xF0`) en is `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 (`PORTD.DIRSET = 0xF0`) gemaakt en `R/W` laag gemaakt. Vervolgens verstuurt de functie eerst het hoge en daarna het lage *nibble* van `d`.

Regel 55-74
`lcd_init()`

De functie `lcd_init` 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 er drie nibbles naar het display gestuurd conform figuur 18.17. Nadat het display in de 4-bit modus is gezet, verstuurt de functie met `lcd_cmd` de overige instellingen.

Regel 76-81
`lcd_puts()`

De functie `lcd_puts` komt overeen met `lcd_puts` uit code 18.2 en drukt de string, waar pointer `s` naar wijst, af. De end-of-string (`'\0'`) wordt niet afgedrukt.

Regel 83-108
`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 `lcd_puts`. 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.

18.6 Toepassing met een LCD-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.

Bij ATmega-microcontrollers is de HD44780-compatibele LCD-bibliotheek van Peter Fleury zeer populair. Helaas is deze bibliotheek zonder forse aanpassingen niet bruikbaar bij de Xmega. Hoewel er veel oplossingen te vinden zijn voor de HD44780-compatibele displays, zijn deze meestal niet volledig. Daarom hoort bij dit boek een complete LCD-bibliotheek, die geschikt is voor de 4-bits- en de 8-bits-modus en die bruikbaar is met en zonder *busy flag*. De bibliotheek bestaat uit twee bestanden `lcd.c` en `lcd.h`. Het c-bestand bevat een groot aantal functies. Tabel 18.6 geeft de functies die toegankelijk zijn voor de gebruiker. Het *header*-bestand `lcd.h` bevat een aantal definities die de gebruiker kan aanpassen voor zijn

Tabel 18.6 : Overzicht van de functies LCD-bibliotheek van dit boek.

Functie	Omschrijving
`void lcd_init(void)`	Initialisatie display en selectie type cursor
`void lcd_clear(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_cmd(uint8_t cmd)`	Stuur instructie naar het LCD
`void lcd_data(uint8_t data)`	Identiek aan `lcd_putc` zonder interpretatie `'\n'`

situatie. De belangrijkste definities staan in tabel 18.7 met de standaardwaarde en de waarde, die gebruikt is in het voorbeeld van code 18.4.

De bestanden `lcd.c` en `lcd.h` moeten in de werkfolder worden geplaatst. Aan `lcd.c` hoeft — of beter moet — niets worden gewijzigd. Omdat de bibliotheek vertragingsfuncties uit `delay.h` gebruikt, moet `F_CPU` bekend zijn. Het *header*-bestand moet wel worden aangepast.

Overigens zijn de bibliotheken van Fleury en van dit boek geen echte bibliotheken, 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` altijd zelf gecompileerd worden.

In code 18.4 staat een programma dat exact hetzelfde doet als code 18.3 maar nu met behulp van de LCD-bibliotheek. Tabel 18.7 geeft behalve de standaardwaarden voor de macrodefinities uit `lcd.h` ook de instelling die bij code 18.4 hoort.

De LCD-bibliotheek kan gebruikt worden met de 4-bits en de 8-bits modus en met en zonder *busy flag*. Bij gebruik van de *busy flag* moet men er op letten dat er op geen enkele ingang van de Xmega 5 V komt te staan. Als de macro `LCD_4BIT_MODE` `0` is, wordt de 8-bits modus gebruikt. Als deze macro ongelijk aan `0` is, wordt de

Tabel 18.7 : Overzicht van de belangrijkste macrodefinities uit de LCD-bibliotheek. De laatste kolom geeft de waarden die in code 18.4 gebruikt zijn.

Functie	Omschrijving	Standaardwaarde	code 18.4
`LCD_4BIT_MODE`	4-bit of 8-bit	`1`	`1`
`LCD_BUSY_FLAG`	busy flag	`0`	`1`
`LCD_LINES`	Aantal regels	`2`	`2`
`LCD_DISP_LENGTH`	Regellengte	`16`	`16`
`LCD_LINE_LENGTH`	Regellengte DDRAM	`0x40`	`0x40`
`LCD_DATA_PORT`	Poort voor data	`PORTD`	`PORTD`
`LCD_D0_PORT`	Poort voor data D0	`LCD_DATA_PORT`	`LCD_DATA_PORT`
:	:	:	:
`LCD_D7_PORT`	Poort voor data D7	`LCD_DATA_PORT`	`LCD_DATA_PORT`
`LCD_D0_bp`	Pin voor data bit `0` (D0)	`PIN0_bp`	`PIN0_bp`
:	:	:	:
`LCD_D7_bp`	Pin voor data bit 7 (D7)	`PIN7_bp`	`PIN7_bp`
`LCD_COMM_PORT`	Poort voor communicatie	`PORTD`	`PORTE`
`LCD_RS_PORT`	Poort voor signaal RS	`LCD_COMM_PORT`	`LCD_COMM_PORT`
`LCD_RW_PORT`	Poort voor signaal RW	`LCD_COMM_PORT`	`LCD_COMM_PORT`
`LCD_E_PORT`	Poort voor signaal E	`LCD_COMM_PORT`	`LCD_COMM_PORT`
`LCD_RS_PIN`	Pin voor signaal RS	`PIN0_bp`	`PIN4_bp`
`LCD_RW_PIN`	Pin voor signaal RW	`PIN1_bp`	`PIN3_bp`
`LCD_E_PIN`	Pin voor signaal E	`PIN2_bp`	`PIN5_bp`

4-bits modus gebruikt. Als `LCD_BUSY_FLAG 0` is, wordt *busy flag* niet gebruikt. In code 18.4 wordt 4-bits modus met de *busy flag* gebruikt.
Het display heeft twee regels en zestien karakters per regel. De vier data-ingangen D7, D6, D5 en D4 zijn verbonden met de pinnen 7 tot en met 4 van poort D. De besturingssignalen `R/W`, `RS` en `E` zijn verbonden met respectievelijk de pinnen 3, 4 en 5 van poort E.

Code 18.4 : **Een voorbeeld van een toepassing met de LCD-bibliotheek.** De functie `show_fibonacci` staat in een apart bestand. Het bijbehorende schema staat in figuur 18.19. Het programma gebruikt de 4-bit modus en is functioneel gelijk aan code 18.3. In de bijbehorende `lcd.h` staan de definities uit de laatste kolom van tabel 18.7.

main.c

```
1  #define F_CPU   2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5  #include <limits.h>
6  #include "lcd.h"
7
8  void show_fibonacci(uint16_t f1, uint16_t f2);
9
10 int main(void)
11 {
12   uint16_t f1 = 1;
13   uint16_t f2 = 1;
14   uint16_t h;
15
16   lcd_init();
17
18   while (1) {
19     show_fibonacci(f1, f2);
20     _delay_ms(1000);
21
22     if (f2 > (UINT_MAX/2)) {
23       f1 = 1;
24       f2 = 1;
25     } else {
26       h  = f2;
27       f2 = f1 + f2;
28       f1 = h;
29     }
30   }
31 }
```

show_fibonacci.c

```
1  #include <stdint.h>
2  #include <stdlib.h>
3  #include "lcd.h"
4
5  char buffer[20];
6
7  void show_fibonacci(uint16_t f1, uint16_t f2)
8  {
9    lcd_clear();
10   utoa(f1, buffer, 10);
11   lcd_puts(buffer);
12
13   lcd_gotoxy(0,1);
14   utoa(f2, buffer, 10);
15   lcd_puts(buffer);
16 }
```

Het hoofdprogramma van code 18.4 is bijna identiek aan het hoofdprogramma uit code 18.3. Het afbeelden van het resultaat op het display wordt nu door de functie `show_fibonacci` gedaan dat in een apart bestand staat. Om naar de tweede regel van het display te gaan is de functie `lcd_gotoxy` gebruikt. Deze functie is net als de functies `lcd_init`, `lcd_clear` en `lcd_puts` gedefinieerd in de LCD-bibliotheek.

Tabel 18.8 : 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`)

18.7 Geformatteerd afdrukken op een LCD

Gewone C-applicaties gebruiken `printf` met de *format specifiers* `%e`, `%f` en `%g` om drijvende-kommagetallen en gebroken getallen af te drukken. Omdat 8-bits microcontrollers beperkte geheugen- en rekenfaciliteiten hebben, is het geformatteerd afdrukken standaard meestal niet geïmplementeerd.

Lezen uit bestanden en afdrukken naar bestanden is vanuit een microcontroller in het algemeen ook niet mogelijk. Een microcontroller kent standaard alleen een aantal interfaces — zoals een UART en een SPI — waarmee gecommuniceerd kan worden. De microcontroller heeft in tegenstelling tot een pc geen toetsenbord en beeldscherm. Er is geen `stdin` en geen `stdout` en de functies `scanf` en `printf` zijn niet direct bruikbaar. Wel kunnen `printf` en `scanf` zo geconfigureerd worden dat de informatie bijvoorbeeld naar een UART gestuurd kan worden of van een `UART` ontvangen kan worden.

In paragraaf 19.11 wordt met behulp van `FDEV_SETUP_STREAM()` de functies `printf`, `fprintf` en `fscanf` zo geconfigureerd dat deze functies via een UART communiceren. Daarbij wordt ook verteld hoe er met `fprintf` informatie naar een LCD geschreven kan worden.

De functie `sprintf`

In code 18.3 en 18.4 is de functie `utoa` gebruikt om de unsigned integers `f1` en `f2` om te zetten naar een string. De functie `lcd_puts` stuurt deze string daarna naar het LCD. In plaats van `utoa` kan de conversie ook met behulp van de functie `sprintf` worden gedaan. De functie `sprintf` schrijft informatie geformatteerd naar een stringbuffer. Regel 10 en 14 moeten dan aangepast worden, zoals dat gedaan is bij de functie `show_fibonacci` van code 18.5.
De functie `sprintf` gebruikt iets meer programma- en datageheugen dan `utoa`, maar het echte nadeel is dat deze functie veel trager is dan `utoa`. De functie `sprintf` heeft bijna 600 klokslagen nodig en `utoa` ruim 250 klokslagen.

Code 18.5 : Alternatieve functie `show_fibonacci` met behulp van `sprintf`.

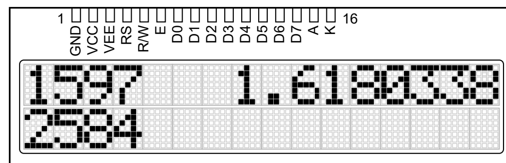
```
1  #include <stdio.h>
2  #include "lcd.h"
3
4  char buffer[20];
5
6  void show_fibonacci(uint16_t f1, uint16_t f2)
7  {
8    lcd_clear();
9    sprintf(buffer,"%u", f1);
10   lcd_puts(buffer);
11
12   lcd_gotoxy(0,1);
13   sprintf(buffer,"%u", f2);
14   lcd_puts(buffer);
15 }
```

18.8 Het weergeven van gebroken getallen op een LCD

Er zijn verschillende manieren om gebroken getallen naar een display te schrijven. Iedere methode heeft zijn eigen bijzonderheden. Bij een aantal alternatieven zijn speciale compiler- of linkeropties nodig en de gebruikte methode beïnvloedt de grootte en de snelheid van de applicatie.

Gebroken getallen met de functie `dtostrf`

Bij *avr-libc* zijn aan de standaardbibliotheek een aantal functies toegevoegd. In tabel 18.8 staat een overzicht. De meeste functies uit deze tabel converteren een variabele met een bepaald dataformaat naar een alfanumerieke string. Bij de functies `dtostrf` en `dtostre` is de math-bibliotheek `libm` nodig.



Figuur 18.21 : Het LCD met rechtsboven de Gulden Snede.

De functies `dtostrf` en `dtostre` hebben de math-bibliotheek nodig. Het linken van de code moet met de optie `-lm` gedaan worden.

De functie `show_fibonacci` uit code 18.6 drukt naast de getallen van Fibonacci 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 alfanumerieke 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 18.21.

Code 18.6 : Toevoeging voor afdrukken Gulden Snede met behulp van `dtostrf`.

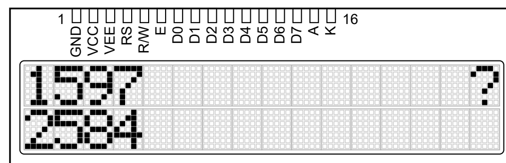
```
// compiler option: -lm
#include <stdint.h>
#include <stdlib.h>
#include "lcd.h"

char buffer[20];

void show_fibonacci(uint16_t f1, uint16_t f2)
{
  lcd_clear();
  utoa(f1, buffer, 10);
  lcd_puts(buffer);
  lcd_gotoxy(0,1);
  utoa(f2, buffer, 10);
  lcd_puts(buffer);
  lcd_gotoxy(7,0);
  dtostrf((double) f2/f1, 9, 7, buffer);
  lcd_puts(buffer);
}
```

Code 18.7 : Toevoeging voor afdrukken Gulden Snede met behulp van `sprintf`.

```
// options: -Wl,-u,vprintf -lprintf_flt -lm
#include <stdio.h>
#include "lcd.h"

char buffer[20];

void show_fibonacci(uint16_t f1, uint16_t f2)
{
  lcd_clear();
  sprintf(buffer,"%u", f1);
  lcd_puts(buffer);
  lcd_gotoxy(0,1);
  sprintf(buffer,"%u", f2);
  lcd_puts(buffer);
  lcd_gotoxy(7,0);
  sprintf(buffer,"%9.7f", (double) f2/f1);
  lcd_puts(buffer);
}
```

Gebroken getallen met de functie `sprintf`

Code 18.7 heeft in principe dezelfde functionaliteit als code 18.6, maar gebruikt in plaats van de functies `utoa` en `dtostrf` de functie `sprintf`. Zonder compilatie met speciale compiler- en linkeropties zal deze oplossing echter niet het juiste resultaat tonen. In plaats van het gebroken getal, verschijnt er een vraagteken, zoals in figuur 18.22.



Figuur 18.22 : De afbeelding met `sprintf` zonder opties.

Standaard bevatten de `printf`-functies bij *avr-gcc* alle functionaliteiten, behalve degene die te maken hebben met gebroken getallen. De *format specifiers* `%e`, `%f` en `%g` zijn niet geïmplementeerd en leveren altijd een vraagteken (`?`) op.

De *avr-gcc*-compiler kent drie implementaties voor de `printf`-functies:

- De standaard versie die geen gebroken getallen kent.
- Een minimale versie met alleen basale integer- en stringconversies. De meeste opties zijn niet beschikbaar. Van de *flags* is bijvoorbeeld alleen `#` voorhanden.
- Een uitgebreide versie die wel gebroken getallen kan afdrukken en die de meeste opties kent.

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.

Met compiler- en linkeropties kan één van de andere implementaties geselecteerd worden. Voor de minimale versie zijn deze opties nodig:

```
-Wl,-u,vfprintf -lprintf_min
```

Voor de uitgebreide versie zijn deze opties nodig:

```
-Wl,-u,vfprintf -lprintf_flt -lm
```

De optie `-lm` voegt de math-bibliotheek toe, de optie `-Wl,-u,vfprintf` verwijdert de standaard implementatie van `printf` en met de optie `-lprintf_flt` voegt de bibliotheek met de uitgebreide versie van `printf` toe.

De applicatie met de functie `dtostrf` die gecompileerd is met optie `-lm`, is kleiner en sneller dan de applicatie die de functie `sprintf` gebruikt.

Code 18.8 : Gebroken getallen zonder speciale functies.

```
1  // compiler option: -lm
2  #include <stdint.h>
3  #include <stdlib.h>
4  #include "lcd.h"
5
6  char buffer[20];
7
8  void show_fibonacci(uint16_t f1, uint16_t f2)
9  {
10   uint16_t h;
11
12   lcd_clear();
13   utoa(f1, buffer, 10);
14   lcd_puts(buffer);
15   lcd_gotoxy(0,1);
16   utoa(f2, buffer, 10);
17   lcd_puts(buffer);
18   lcd_gotoxy(7,0);
19   h = f2/f1;
20   utoa(h, buffer,10);
21   lcd_puts(buffer);
22   lcd_putc('.');
23   ultoa(((long double)f2/f1 - h) * 10000000,
24                                  buffer, 10);
25   lcd_puts(buffer);
26 }
```

Code 18.9 : Gebroken getallen zonder `double`.

```
1  // compiler option: -lm
2  #include <stdint.h>
3  #include <stdlib.h>
4  #include "lcd.h"
5
6  char buffer[20];
7
8  void show_fibonacci(uint16_t f1, uint16_t f2)
9  {
10   uint16_t h;
11   uint64_t h64;
12
13   lcd_clear();
14   utoa(f1, buffer, 10);
15   lcd_puts(buffer);
16   lcd_gotoxy(0,1);
17   utoa(f2, buffer, 10);
18   lcd_puts(buffer);
19   lcd_gotoxy(7,0);
20   h = f2/f1;
21   utoa(h, buffer,10);
22   lcd_puts(buffer);
23   lcd_putc('.');
24   h64 = ((uint64_t) f2)*10000000/f1 -
25                  ((uint64_t) h)*10000000;
26   ultoa ((unsigned long) h64, buffer, 10);
27   lcd_puts(buffer);
28 }
```

Gebroken getallen zonder speciale functies

Code 18.8 bevat een alternatief voor code 18.6 en code 18.7. De conversie gebeurt in dit voorbeeld met behulp van `utoa` en `ultoa`. De truc is om het gebroken getal te splitsen door het een deel voor de komma en het deel achter de komma en apart af te drukken. Figuur 18.23 laat zien dat de geheeltallige deling van `f2` en `f1` het deel

`h` voor de komma geeft en dat het getal achter de komma afhangt van de gebroken deling van `f2` en `f1`, van `h` en vann een factor, die het aantal cijfers achter de komma bepaalt.

$$\frac{34}{21} = \boxed{1}\ \boxed{,}\ \boxed{6190476}$$

Figuur 18.23 : **Omzetting van Φ in een deel voor en achter de komma.** Van een gebroken getal is het deel voor de komma gevonden met een geheeltallige deling. Het deel achter de komma is de gebroken deling verminderd met het deel voor de komma. De factor `10000000` bepaalt het aantal cijfers achter de komma.

Omdat er in dit voorbeeld zeven cijfers achter de komma worden getoond, is er een typecasting met **`long double`** nodig bij de gebroken deling `f1/f2`. Met zes cijfers achter de komma voldoet **`double`** ook.

Code 18.9 geeft een variant die ook geen **`double`** gebruikt en alles met gehele getallen doet. Voordat `f2` door `f1` gedeeld wordt, is `f2` vermenigvuldigd met de factor `10000000`. Er is wel een 64-bits getal nodig voor de berekening van het deel achter de komma.

De grootte en snelheid van de laatste twee voorbeelden zijn niet veel beter dan het voorbeeld uit code 18.7 met de functie `sprintf`. Code 18.6 met de functie `dtostrf` is wel duidelijk sneller dan code 18.7. Bovendien hoeft de uitgebreide `printf`-bibliotheek dan niet geselecteerd te worden.

19

UART

Doelstelling

Dit hoofdstuk behandelt de USART's van de Xmega. 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 USART en het instellen van de *baud rate*.
- De instelling van het RS232-protocol.
- De communicatie met het RS232-protocol tussen een Xmega en een pc.
- Het ontvangen en verzenden van data.
- Circulaire buffers voor de buffering van gegevens.
- De USART-driver van Atmel en een *wrapper* voor deze driver.
- Geformatteerd versturen en ontvangen van data via de UART met `printf` en `scanf`.

De communicatie met de UART wordt gedemonstreerd met de volgende 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 een *wrapper* voor de USART-driver van Atmel.
- 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`.
- Het gebruik van de standaard streams bij USART`0` van poort F.

De Xmega heeft verschillende mogelijkheden voor seriële communicatie, namelijk met behulp van een USART, via een SPI, met TWI of via USB. 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 externe DAC, een serieel LCD of een andere microcontroller. De USB-interface is vooral bestemd voor communicatie met een PC en 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 ook met de RS232-poort van een pc communiceren.

USB (*Universal Serial Bus*) is net als Firewire en Ethernet een voorbeeld van snelle seriële verbindingen die bij een pc worden gebruikt.

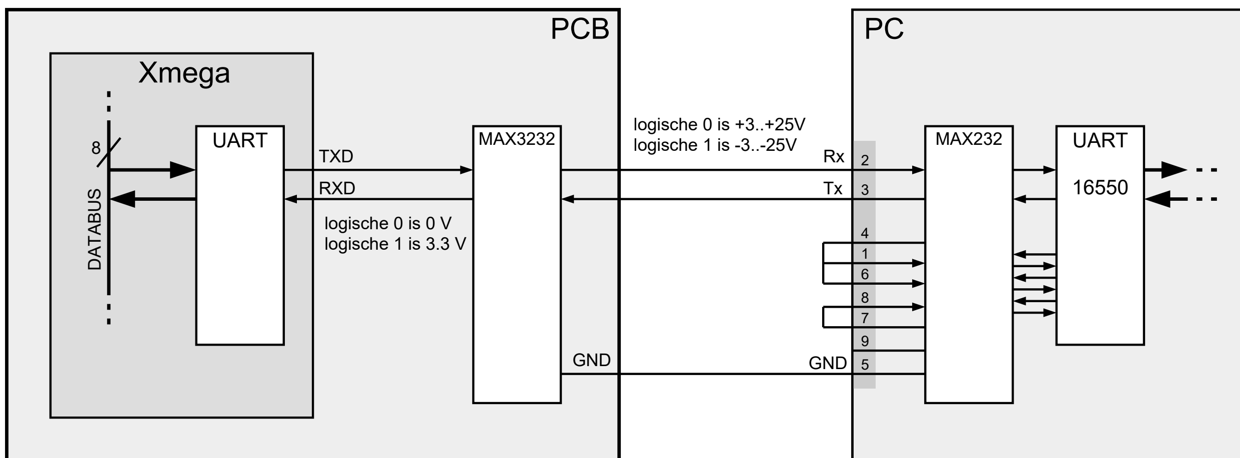
Het feit dat de Xmega meerdere manieren kent om serieel te communiceren, geeft aan dat deze manier van communiceren belangrijk is. Bij parallelle communicatie

worden een aantal bits gelijktijdig verzonden. Bij seriële communicatie worden de bits na elkaar verstuurd. Hoewel dat laatste onhandig lijkt, is serieel versturen efficiënter dan parallel. Ten eerste is maar één 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's van de Xmega ook synchroon gebruikt kunnen worden, worden in de meeste applicaties de USART's van de Xmega asynchroon toegepast. Het kloksignaal voor de synchronisatie wordt dan niet gebruikt. De betreffende aansluiting is dan beschikbaar voor andere doeleinden. Feitelijk is de USART dan een UART (*Universal Asynchronous Transmitter Receiver*) met één uitgang (TXD) en één ingang (RXD).



Figuur 19.1 : De verbinding tussen een pc en een Xmega. De TXD en de RXD van een UART van de Xmega 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 Xmega en de pc zit een MAX3232 die de signalen naar de juiste signaalniveaus converteert.

De MAX232 is de meest populaire *RS232 level converter*. Fabrikant Maxim levert een groot aantal varianten, onder andere de MAX233 die geen externe condensatoren nodig heeft en een MAX3232 die geschikt is voor 3,3 V.

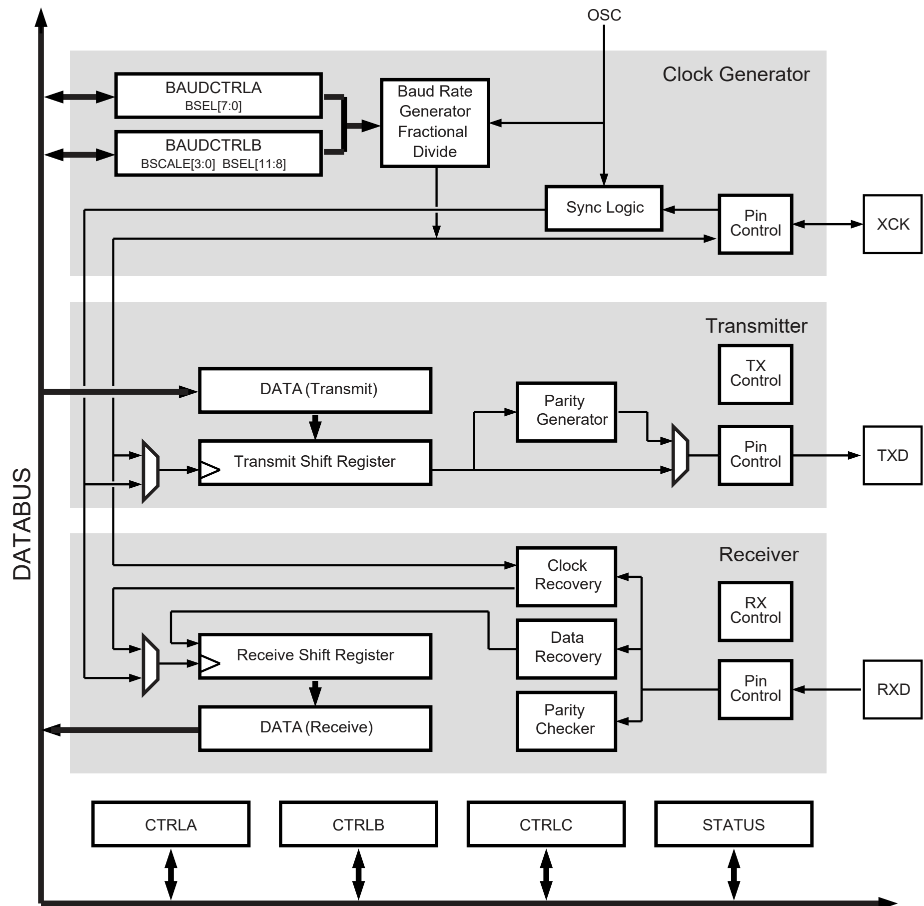
Naast de Rx en Tx kent de RS232-poort ook een aantal signalen voor *handshaking*. Deze aansluitingen ontbreken bij de UART van de Xmega. In figuur 19.1 zijn deze signalen teruggekoppeld naar de pc.

In figuur 19.1 is de UART van de Xmega 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 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 19.1 maakt zichtbaar dat de pc intern ook een MAX232 en een UART heeft. In bijlage B staat meer informatie over RS232 en tevens een aantal voorbeelden — in de taal C — van pc-applicaties die informatie van en naar de RS232-poort sturen.

Er zijn twee `BAUDCTRL`-registers: `BAUDCTRLA` bevat de acht minst significante bits van `BSEL`; `BAUDCTRLB` bevat het 4-bits getal `BSCALE` en de vier meest significante bits van het 12-bits getal `BSEL`.

Het `DATA (transmit)`-register en het `DATA (receive)`-register hebben beide hetzelfde adres. Als er naar `DATA` wordt geschreven, wordt automatisch `DATA (transmit)` gebruikt en als er uit `DATA` wordt gelezen, gebeurt dit automatisch uit `DATA (receive)`.



Figuur 19.2 : **De opbouw van de USART bij de Xmega.** Het bovenste deel genereert het kloksignaal voor de schuifregisters. De frequentie van dit signaal wordt bepaald uit `BAUDCTRLA/B` en de systeemklok of — als de USART synchroon in slave mode wordt gebruikt — door het externe signaal `XCK`.
In het midden staat de zender. De te verzenden waarde wordt in `DATA (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 `DATA (receive)` plaatst. Helemaal onderaan staan de drie controlregisters en het statusregister.

19.1 Opbouw USART en het instellen van baudsnelheid

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 19.2 laat zien dat de USART van de Xmega uit drie delen bestaat: een blok dat een kloksignaal met de juiste *baud rate* of baudsnelheid 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 waarden van de `BAUDCTRLx`-registers en door de frequentie f_{cpu} van de systeemklok of door een extern kloksignaal dat is verbonden met `XCK`.

Bij de generatie van het kloksignaal gebruikt de Xmega een zogenoemde *fractional baud rate generator*. Bij een gewone *baud rate generator* wordt de systeemklok gebruikt om het datasignalen te sampelen. In het algemeen worden er meestal per bit acht of zestien samples genomen. Het effect is dat de maximale baudsnelheid niet al te hoog is en dat de hoge baudsnelheden vaak slecht matchen met de gangbare kloksnelheden. Voor andere baudsnelheden wordt met behulp van een teller de systeemklok gedeeld. Het aantal mogelijke baudsnelheden is daarmee zeer beperkt. Voor een systeem met zestien samples en een systeemklok van 2 MHz is de snelheid gelijk aan $125000/n$ bps, waarbij $n \geq 0$ en n geheeltallig is.

Tabel B.1 in bijlage B geeft de standaard snelheden voor RS232-communicatie.

De *fractional baud rate generator* bevat een mechanisme dat de systeemklok ook niet-geheeltallig kan delen. Daardoor zijn er veel meer snelheden mogelijk. De truc is dat de teller, waarmee de baudsnelheid wordt ingesteld, op gezette tijden een klokslag overslaat. De Xmega gebruikt twee getallen voor het instellen van de juiste baudsnelheid: `BSCALE` en `BSEL`. De schaalfactor `BSCALE` ligt tussen −7 en +7 en `BSEL` is een 12-bits getal dat ligt tussen 0 en 4095.

De UART's van de Xmega gebruikt normaal 16 samples en 8 samples bij de *double transmission speed*. Deze modus wordt ingesteld met de `CLK2X`-bit van register `CTRLB`.

De frequentie f_{baud} van het signaal, waarmee de schuifregisters worden geklokt, hangt af van de klokfrequentie f_{cpu}, het aantal samples N en de waarden van `BSCALE` en `BSEL`:

$$f_{baud} = \begin{cases} \dfrac{f_{cpu}}{N\,2^{BSCALE}\,(BSEL+1)} & \text{voor} \quad BSCALE \geq 0 \\ \dfrac{f_{cpu}}{N\,(2^{BSCALE}\,BSEL+1)} & \text{voor} \quad BSCALE < 0 \end{cases} \tag{19.1}$$

Meestal is er een gewenste *baud rate* gegeven en moeten daarbij geschikte waarden van `BSEL` en `BSCALE` gezocht worden. Dit wordt meestal gedaan door een geschikte waarde voor `BSCALE` te kiezen en hieruit `BSEL` te berekenen:

$$BSEL = \begin{cases} \dfrac{f_{cpu}}{N\,2^{BSCALE}\,f_{baud}} - 1 & \text{voor} \quad BSCALE \geq 0 \\ \dfrac{1}{2^{BSCALE}}\,(\dfrac{f_{cpu}}{N\,f_{baud}} - 1) & \text{voor} \quad BSCALE < 0 \end{cases} \tag{19.2}$$

Soms neemt men een oscillator met een 'magische' frequentie. Dat zijn frequenties die goed passen bij de *baud rate*-reeks:

f_{cpu} **in MHz**
1,8432
3,6864
7,3728
11,0592
14,7456

Bij de Xmega met de *fractional baud rate generator* is dit niet nodig.

Deze berekening levert in het algemeen een gebroken getal op voor `BSEL`. De fout Δ_{baud} in procenten is:

$$\Delta_{baud} = (\frac{f_{baud,benadering}}{f_{baud}} - 1) \times 100\% \tag{19.3}$$

Deze fout moet afhankelijk van het aantal databits en van aantal samples volgens de datasheet niet groter zijn dan 1 à 2%.

De meeste voorbeelden in dit hoofdstuk gebruiken een systeemklok van 2 MHz en een communicatiesnelheid van 115200 baud. Als voor `BSCALE` −7 wordt genomen, moet `BSEL` gelijk zijn aan 10,889. Afgerond is dat 11. De gebruikte baudsnelheid is dan 115108 en de afrondingsfout is kleiner dan 0,1%.

Het bepalen van de juiste BSCALE en BSEL kan lastig zijn. Een strategie is om bij de laagste waarde van BSCALE — dat is −7 — te beginnen en vervolgens de bijbehorende BSEL te berekenen. Als de berekende BSEL groter is dan 4095, herhaal dit dan met de volgende waarde van BSCALE. De eerste BSCALE, waarvoor een geldige BSEL wordt gevonden, is geschikt.

De software van de application note 1307, *Using the XMEGA USART*, bevat een excelbestand voor de berekening van BSCALE en BSEL uit de klokfrequentie en de gewenste baudsnelheid. Ook op de internetsite van dit boek staat een tool om BSCALE en BSEL voor een gewenste baudsnelheid te berekenen.

De waarden van BSCALE en BSEL staan in de registers BAUDCTRLA en BAUDCTRLB, zie ook figuur 19.3. Register BAUDCTRLB bevat het 4-bits getal BSCALE en de vier meest significante bits van het 12-bits BSEL. De acht minst significante bits van BSEL staan in register BAUDCTRLA.

Figuur 19.3 : De registers BAUDCTRLA en BAUDCTRLB voor het instellen van BSCALE en BSEL.

In code 19.1 staat de functie set_usartctrl, die BSEL en BSCALE toekent aan de registers BAUDCTRLA en BAUDCTRLB van een UART. Register BAUDCTRLA bevat de minst significante bits van BSEL en register BAUDCTRLB bevat de schaalfactor BSCALE en de vier meest significante bits van BSEL. De bits van BSCALE zijn vier posities naar links geschoven en de vier meest significante bits worden gevonden door BSEL acht posities naar rechts te schuiven.

Code 19.1 : De functie usart_set_bsel_bscale voor het instellen van BSEL en BSCALE.

```
void set_usartctrl(USART_t *usart, uint8_t bscale, uint16_t  bsel)
{
  usart->BAUDCTRLA = bsel;
  usart->BAUDCTRLB = (bscale << 4) | (bsel >> 8);
}
```

Als de macrodefinitie BSCALE en BSEL de waarden van de schaalfactor en de periodeselectie bevatten, is de aanroep om de baudsnelheid van USART1 van poort D in te stellen:

```
    set_usartctrl(&USARTD1, BSCALE, BSEL);
```

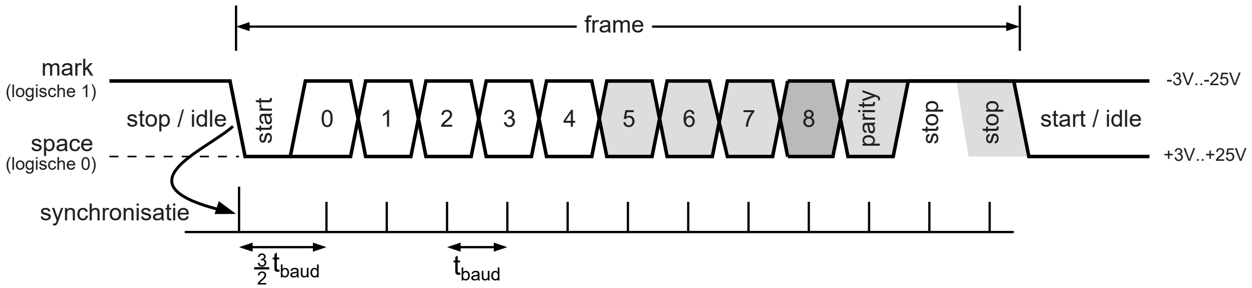
In code 19.2 staat een variant van functie set_usartctrl. De waarde 4 is vervangen door de macrodefinitie USART_BSCALE0_bp; dat is de positie van bit 0 van de BSCALE-bits. Daarnaast is een aantal groepsmaskers toegevoegd om er zeker van te zijn dat de juiste bitgroepen gebruikt worden.

Code 19.2 : De functie usart_set_bsel_bscale met de macrodefinities uit io.h.

```
void set_usartctrl (USART_t *usart, uint8_t bscale, uint16_t  bsel)
{
  usart->BAUDCTRLA = (bsel & USART_BSEL_gm);
  usart->BAUDCTRLB = ((bscale << USART_BSCALE0_bp) & USART_BSCALE_gm) |
                     ((bsel >> 8) & ~USART_BSCALE_gm);
}
```

19.2 Instelling protocol

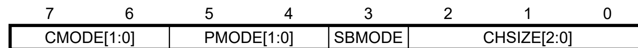
Het protocol voor de asynchrone communicatie bestaat uit een startbit, één of meer stopbits en een aantal databits. Figuur 19.4 toont een frame van het protocol. De bits zijn met frequentie f_{baud} verstuurd. De tijd tussen twee bits is de baudperiode $t_{\text{baud}} = 1/f_{\text{baud}}$. Bij het lezen start de synchronisatie op de neergaande flank van de 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 19.4 : De synchronisatie van het frame bij asynchrone communicatie vindt plaats op de neergaande flank van de startbit. De grijs gekleurde bits zijn optioneel. De ontvanger detecteert de flank van de startbit. Op anderhalve baudperiode bevindt zich het midden van de eerste bit. Tussen de bits zit steeds één baudperiode.

Het frame bevat vijf tot negen databits. Een 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 Xmega kent één of twee stopbits. De meeste toepassingen gebruiken een protocol met acht databits, geen pariteitsbit en een stopbit: het zogenoemde 8N1-protocol.

Als de USART gebruikt wordt in de `IRCOM`-modus, hebben de bits in `CTRLC` een andere betekenis.
Een aantal bits van `CTRLC` hebben in de master-SPI modus geen betekenis.



Figuur 19.5 : Het `CTRLC`-register stelt het protocol in. Voor de asynchrone modus zijn de `CMODE`-bits laag. De `PMODE`-bits stellen het pariteitsbit in en het bit `SBMODE` stelt de stopbit in. De drie `CHSIZE`-bits bepalen het aantal databits.

Figuur 19.5 toont het `CTRLC`-register van de USART. De bits zijn verdeeld in vier groepen voor de modus, de pariteit, het aantal stopbits en het aantal databits.
De USART kent vier modi: de asynchrone modus, de synchrone modus, een modus voor de infrarood communicatie en de master-SPI modus. Dit hoofdstuk bespreekt alleen de asynchrone modus. Paragraaf 21.4 bespreekt de master-SPI modus van de USART.

Tabel 19.1 : De modus van de USART.

modus	`CMODE[1:0]`
asynchroon	`00`
synchroon	`10`
IRCOM	`01`
Master SPI	`11`

Tabel 19.2 : De pariteit van de USART.

pariteit	`PMODE[1:0]`
geen	`00`
even	`10`
oneven	`11`

Tabel 19.3 : De stopbits van de USART.

stopbit	`SBMODE`
1	`0`
2	`1`

Tabel 19.4 : De databits van de USART.

databits	`CHSIZE[2:0]`
5	`000`
6	`001`
7	`010`
8	`011`
9	`111`

In de tabellen 19.1, 19.2, 19.3 en 19.4 staan de mogelijke waarden van deze bits voor de modus, de pariteit, het aantal stopbits en het aantal databits.

19.3 Ontvangen en verzenden van data

Figuur 19.2 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 heten beide registers `DATA`. Te verzenden gegevens worden in register `DATA` 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, plaatst 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 de dataregisters. Het `DATA(receive)`-register kan alleen gelezen worden als alle informatie ontvangen is. Er kan alleen naar het `DATA (transmit)`-register geschreven worden als de eerder verzonden gegevens verwerkt zijn.

7	6	5	4	3	2	1	0
-	-	-	RXEN	TXEN	CLK2X	MPCM	TXB8

Figuur 19.6 : Het register `CTRLB` van de USART. De bits `RXEN` en `TXEN` zetten de zender en de ontvanger aan. De `CLK2X`-bit zet *double clock speed* aan.

7	6	5	4	3	2	1	0
RXCIF	TXCIF	DREIF	FERR	BUFOVF	PERR	-	RXB8

Figuur 19.7 : Het statusregister van de USART. De bits `RXCIF`, `TXCIF` en `DREIF` geven aan dat er gegevens ontvangen zijn, verstuurd zijn en het dataregister van de zender leeg is.

Het register `STATUS` — zie figuur 19.7 — heeft drie bits die de status van de UART aangeven: `RXCIF`, `TXCIF` en `DREIF`. De `RXCIF`-bit (*Receive Complete*) is hoog als er gegevens zijn ontvangen. Onderstaande code blijft de **`while`**-lus doorlopen totdat `RXCIF` hoog is. Daarna wordt de inhoud van register `DATA` aan `data` toegekend.

```
while ( !(USARTD1.STATUS & USART_RXCIF_bm) ) {}; // wait until data is read
data = USARTD1.DATA;                             // read data
```

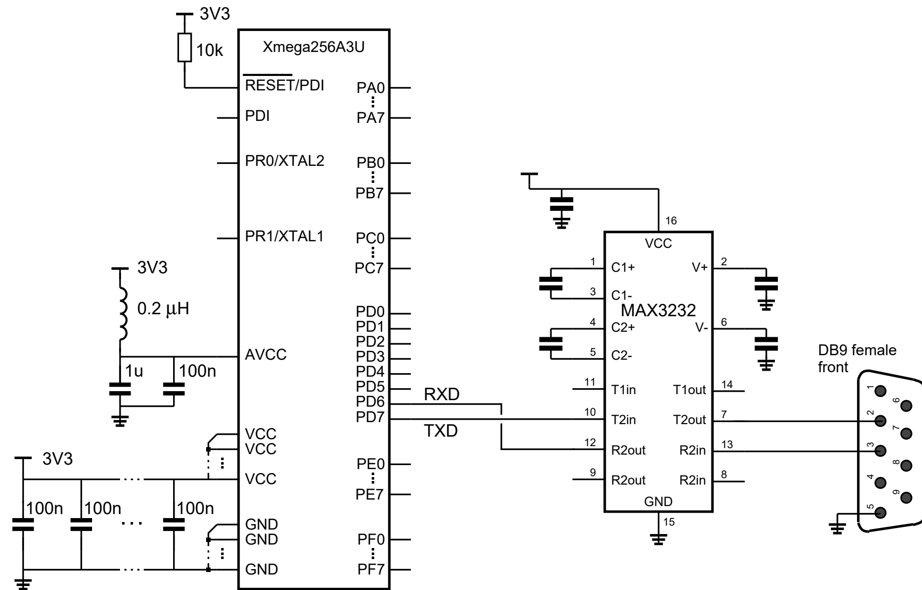
De `TXCIF`-bit (*Transmit Complete*) wordt hoog als de gegevens verstuurd zijn. In onderstaande code krijgt register `DATA` de waarde `data` en daarna wordt er gewacht totdat `TXCIF` hoog is en de gegevens verstuurd zijn:

```
USARTD1.DATA = data;                             // write data
while ( !(USARTD1.STATUS & USART_TXCIF_bm) ) {}; // 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 de `DREIF`-bit te gebruiken.

```
while ( !(USARTD1.STATUS & USART_DREIF_bm) ) {}; // wait until DATA is empty
USARTD1.DATA = data;                             // write data
```

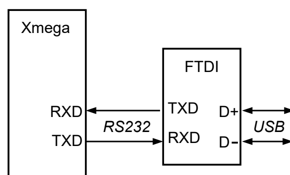
Nu wordt eerst gewacht totdat `DATA` leeg is. Nadat `data` aan `DATA` is toegekend, gaat het programma gewoon verder. Onderwijl verstuurt de zender de databyte die in `DATA` is geplaatst.



Figuur 19.8 : **Het schema voor een seriële verbinding met de USART1 van poort D.** Pin 2 en pin 3 van DB9-connector zijn verbonden via de MAX3232 met respectievelijk de `PD7` en `PD6` van de Xmega. Dat zijn de `TXD` en de `RXD` van `USART1` van poort D.

19.4 Het versturen van karakters via USART1 van poort D

In figuur 19.8 staat het schema voor het verzenden en het ontvangen van informatie via de UART. De MAX3232 is een MAX232 die geschikt is voor 3,3 V. Deze component zorgt voor de inversie en de *level shifting*. In figuur 19.8 voldoen de signalen rechts van de de MAX3232 aan de RS232-norm. De signalen `TXD` en `RXD` links van de MAX3232 liggen tussen GND en VCC. De MAX232 heeft vier condensatoren van 1 μF nodig. Er bestaan veel varianten van de MAX232. Zo is de MAX3233 van Maxim een IC met dezelfde functionaliteit, alleen heeft deze component geen condensatoren nodig.



Figuur 19.9 : De FTDI-chip converteert RS232 naar USB.

Omdat de meeste computers tegenwoordig geen COM-poort meer hebben, bestaan er speciale RS232-USB-converters, die een RS232-signaal omzetten naar USB. Met speciale drivers ziet de computer deze USB-aansluiting als een virtuele COM-poort. In figuur 19.9 is dit getekend voor de populaire FTDI-chip.

In figuur 19.8 zijn pin 2 en pin 3 van de DB9-connector via de MAX232 verbonden met respectievelijk `PD7` en `PD6` van de Xmega. Dat zijn de `TXD` en de `RXD` van UART1 van poort D. In code 19.3 staat het programma dat voortdurend een serie karakters verstuurt. In dit voorbeeld wordt de `RXD` niet gebruikt. Het versturen van gegevens is eenvoudiger dan het ontvangen. Bovendien kan de communicatie worden getest door de schakeling aan te sluiten op een pc en het resultaat met een hyperterminal te bekijken.

Het protocol van 8 databits, geen (*None*) pariteitsbit en 1 stopbit wordt ook aangeduid als 8N1.

De initialisatiefunctie `init_uart_bscale_bsel` stelt op regel 20 de USART in op de asynchrone modus, op acht databits, een stopbit en geen pariteitsbit. Op regel 22 stelt de functie `set_usartctrl` de juiste baudsnelheid in. In dit voorbeeld gebruikt de Xmega de interne klok van 2 MHz. De baudsnelheid is 115200 bps. Met een `BSCALE` van −7 en een `BSEL` van 11 is de fout kleiner dan 0,1 %.

De zender wordt op regel 19 geactiveerd door de `TXEN`-bit in register `CTRLB` hoog te maken. Het `CTRLB`-register staat in figuur 19.6 en bevat naast de `TXEN`- en de `RXEN`-bit, waarmee respectievelijk de zender en de ontvanger worden aangezet, onder andere ook de `CLK2X`-bit dat de UART instelt op de *double clock speed*. In dit geval wordt de *normal clock speed* gebruikt.

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 `DATA` leeg is en kent daarna aan `DATA` de ASCII-waarde van de lusvariabele toe. Deze waarde is gelijk aan de lusvariabele verhoogd met de ASCII-waarde van `'0'`.

In dit voorbeeld wordt de interne klok van de Xmega gebruikt. De frequentie kan tot 5% afwijken van de 2 MHz. Het is verstandig de interne klok te kalibreren. Bijvoorbeeld met de functie `AutoCalibration2M` uit paragraaf 23.5.

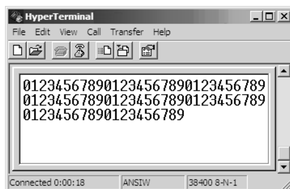
Code 19.3 : Het versturen van gegevens met UART1 vaan poort D.

```
 1 #define F_CPU     2000000UL
 2
 3 #include <avr/io.h>
 4 #include <util/delay.h>
 5
 6 void set_usartctrl(USART_t *usart, uint8_t bscale, uint16_t bsel)
 7 {
 8   usart->BAUDCTRLA = (bsel & USART_BSEL_gm);
 9   usart->BAUDCTRLB = ((bscale << USART_BSCALE0_bp) & USART_BSCALE_gm) |
10                      ((bsel >> 8) & ~USART_BSCALE_gm);
11 }
12
13 void init_uart_bscale_bsel(USART_t *usart, int8_t bscale, int16_t bsel)
14 {
15   PORTD.DIRSET = PIN7_bm;          // TXD
16   PORTD.DIRCLR = PIN6_bm;          // RXD (not used)
17
18   usart->CTRLA = 0;                // no interrupts
19   usart->CTRLB = USART_TXEN_bm;    // enable transmitter
20   usart->CTRLC = USART_CMODE_ASYNCHRONOUS_gc|
21                  USART_PMODE_DISABLED_gc|USART_CHSIZE_8BIT_gc;
22   set_usartctrl(usart, bscale, bsel);
23 }
24
25 int main(void)
26 {
27   init_uart_bscale_bsel(&USARTD1, -7, 11);     // BAUD RATE 115200
28
29   while (1) {
30     for (int i=0; i<=9; i++) {
31       while ( ! (USARTD1.STATUS & USART_DREIF_bm) ) ;
32       USARTD1.DATA = i + '0';
33     }
34     _delay_ms(500);
35
36   }
37 }
```

Een (hyper-)terminal is een communicatieprogramma dat verbinding maakt met andere computers via modems, RS232 of Ethernet.

Standaard zit het programma HyperTerminal niet meer bij Windows. Een goed alternatief is Putty: http://www.chiark.greenend.org.uk/ sgtatham/putty/

Een ander alternatief is Tera Term: http://ttssh2.sourceforge.jp/index.html.en



Figuur 19.10 : De uitvoer van de applicatie uit code 19.3 in een hyperterminal.

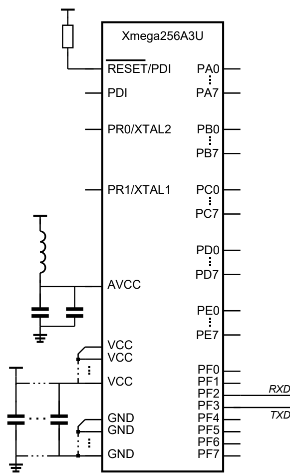
In figuur 19.10 staat de schermafdruk van een hyperterminal, die aangesloten is op de schakeling van figuur 19.8 en waarbij de microcontroller geprogrammeerd is met het programma van code 19.3. Voor een goede communicatie moet de terminal ingesteld zijn op 115200 baud en op het 8N1-protocol.

19.5 Het ontvangen, converteren en versturen van karakters

Het programma van code 19.4 ontvangt een karakter via de `RXD`-ingang van de UART. Het converteert dit karakter van hoofdletter naar kleine letter en omgekeerd. Daarna stuurt het dit geconverteerde karakter via de `TXD` terug. Dit voorbeeld, zie figuur 19.11, gebruikt de `RXD` en `TXD` van USART`0` van poort F.

De functie `uart_init_bscale_bsel` is identiek met die uit code 19.3, alleen regel 12 is aangepast. De zender en de ontvanger staan in dit geval allebei aan. De functie `change_case` converteert een hoofdletter naar onderkast en een kleine letter naar bovenkast. De andere karakters blijven ongewijzigd.

In de oneindige lus van regel 32 wacht het hoofdprogramma op gegevens. Het ontvangen karakter wordt opgeslagen in `data`. Daarna wacht het programma op regel 34 totdat het `DATA`-register van de zender leeg is en stuurt het karakter geconverteerd terug.



Figuur 19.11 : De Xmega met de aansluiting voor `USART0` van poort F.
Op het Xmega-bord, uit bijlage J, zijn de `RXD` en `TXD` verbonden met de tweede Xmega, die deze signalen omzet naar USB.

Code 19.4 : Het ontvangen en versturen van gegevens met de UART.

```
 1  #define F_CPU     2000000UL
 2  #include <avr/io.h>
 3  #include <ctype.h>
 4
 5  void set_usartctrl (USART_t *usart, uint8_t bscale, uint16_t bsel);
 6
 7  void init_uart_bscale_bsel(USART_t *usart, int8_t bscale, int16_t bsel)
 8  {
 9    PORTF.DIRSET = PIN3_bm;                         // TXD
10    PORTF.DIRCLR = PIN2_bm;                         // RXD
11    usart->CTRLA = 0;                               // no interrupts
12    usart->CTRLB = USART_TXEN_bm | USART_RXEN_bm;
13    usart->CTRLC = USART_CMODE_ASYNCHRONOUS_gc|
14                         USART_PMODE_DISABLED_gc|USART_CHSIZE_8BIT_gc;
15    set_usartctrl(usart, bscale, bsel);
16  }
17
18  char change_case(char c)
19  {
20    if ( isupper(c) )  return tolower(c);
21    if ( islower(c) )  return toupper(c);
22    return c;
23  }
24
25  int main(void)
26  {
27    char data;
28
29    init_uart_bscale_bsel(&USARTF0, -7, 289);  // BAUD RATE 38400
30
31    while(1) {
32      while ( ! (USARTF0.STATUS & USART_RXCIF_bm) ) {};
33      data = USARTF0.DATA;
34      while ( ! (USARTF0.STATUS & USART_DREIF_bm) ) {};
35      USARTF0.DATA = change_case(data);
36    }
37  }
```

In code 19.4 staat alleen het prototype van de functie `set_usartctrl`. Deze functie is beschreven in code 19.2.

De baudsnelheid is nu ingesteld op 38400.

Als bij het uitvoeren van code 19.4 de hyperterminal het ingevoerde karakter echoot, is direct te zien dat het teruggegeven karakter is gewijzigd.
De `a` wordt beantwoordt met een `A`, de `b` met een `B`, et cetera.

19.6 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_{byte} die nodig is om een byte te versturen of te ontvangen is dan:

$$t_{\text{byte}} = 10 * \frac{1}{f_{\text{baud}}}$$

Tabel 19.5 : Tijd nodig voor versturen van een byte.

f_{baud}	t_{byte}
2400	4,2 ms
4800	2,1 ms
9600	1,1 ms
14400	694 μs
19200	521 μs
28800	347 μs
38400	260 μs
57600	174 μs
76800	130 μs
115200	87 μs

In tabel 19.5 is t_{byte} berekend voor verschillende baudsnelheden. Vooral bij lage snelheden kost het versturen of ontvangen van een karakter relatief veel tijd. In plaats van te wachten totdat een byte verstuurd of ontvangen is, is het in praktische situaties vaak beter om met interrupts te werken.

De USART kent drie interrupts die actief worden als respectievelijk alle data ontvangen zijn, alle data verstuurd zijn en de ontvangstbuffer weer gereed is om data te verwerken. Als één van deze interrupts plaats vindt, is de betreffende *flag* — `RXCIF`, `TXCIF` of `DREIF` — van het statusregister hoog, zie figuur 19.7. Om de interrupts te activeren en het juiste gevoeligheidsniveau te geven, bevat register `CTRLA` drie groepen met interruptgevoeligheidsbits. Deze groepen horen bij de drie genoemde interrupts. In figuur 19.12 is register `CTRLA` getekend.

Figuur 19.12 : Het register `CTRLA` met de interruptgevoeligheidsbits.

In code 19.5 staat een programma dat een interrupt gebruikt om een karakter te lezen, aan te passen en weer terug te sturen via de UART. Het hoofdprogramma maakt uitgang `PC0` voortdurend hoog en laag en heeft geen bemoeienis met de communicatie via de UART en het omzetten van de karakters. Het programma gebruikt de *receive complete* interrupt. Telkens als een karakter ontvangen is, start deze interrupt de interruptfunctie, die het karakter wijzigt en direct weer terugstuurt.
De gebruikte schakeling is weer het schema uit figuur 19.11. Als pin `PC0` verbonden is met een led, zal deze led voortdurend knipperen, terwijl de UART intussen met de pc communiceert.

Het begin van code 19.5 komt voor een groot deel overeen met de eerste 23 regels van code 19.4. Om de interruptfuncties te gebruiken is op regel 4 de include met `interrupt.h` toegevoegd. Verder is in de functie `init_uart_bscale_bsel` regel 14 gewijzigd. De *receive complete*-interrupt is aangezet door in register `CTRLA` het gevoeligheidsniveau van de betreffende bits in te stellen op het lage niveau. Iedere keer als er gegevens — in dit geval de acht bits van het karakter — ontvangen zijn, wordt `RXCIF` in het statusregister hoog en triggert dit de interruptfunctie.
Op regel 7 en 8 staan de prototypen van de functies `set_usartctrl` en `change_case`. Deze functies zijn beschreven in code 19.2 en 19.4.

De interruptfunctie staat op regel 21, kopieert eerst het ontvangen karakter uit het `DATA`-register van de ontvanger naar de variabele `data` en plaatst vervolgens het gewijzigde karakter in het `DATA`-register van de zender.

Code 19.5 : Het ontvangen en versturen van gegevens met een interrupt.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6
7  void set_usartctrl (USART_t *usart, uint8_t bscale, uint16_t  bsel);
8  char change_case(char c);
9
10 void init_uart_bscale_bsel(USART_t *usart, int8_t bscale, int16_t bsel)
11 {
12   PORTF.DIRSET = PIN3_bm;                       // TXD
13   PORTF.DIRCLR = PIN2_bm;                       // RXD
14   usart->CTRLA = USART_RXCINTLVL_LO_gc;         // RXD interrupt
15   usart->CTRLB = USART_TXEN_bm | USART_RXEN_bm;
16   usart->CTRLC = USART_CMODE_ASYNCHRONOUS_gc|
17                        USART_PMODE_DISABLED_gc|USART_CHSIZE_8BIT_gc;
18   set_usartctrl(usart, bscale, bsel);
19 }
20
21 ISR(USARTF0_RXC_vect)
22 {
23   char data;
24
25   data = USARTF0.DATA;
26   USARTF0.DATA = change_case(data);
27 }
28
29 int main(void)
30 {
31   PORTC.DIRSET = PIN0_bm;                      // Pin 0 port C output
32
33   init_uart_bscale_bsel(&USARTF0, -7, 289);    // BAUD RATE 38400
34
35   PMIC.CTRL   |= PMIC_LOLVLEN_bm;
36   sei();
37
38   while(1) {
39     PORTC.OUTTGL = PIN0_bm;                    // Toggle Pin 0 port C
40     _delay_ms(500);
41   }
42 }
```

Het hoofdprogramma maakt eerst pin `0` van poort C een uitgang, initialiseert de UART, maakt het interruptmechanisme gevoelig voor laag niveau interrupts en zet het globale interruptmechanisme aan. Daarna volgt de oneindige lus die pin `0` van poort C iedere `500` ms afwisselend hoog en laag maakt. Een led die op deze uitgang is aangesloten zal met een frequentie van 1 Hz knipperen.

De interruptfunctie leest voortdurend de gegevens uit het `DATA (receive)`-register en schrijft deze — geconverteerd – naar het `DATA (transmit)`-register. Meestal wordt

er net als in code 19.5 een hulpvariabele gebruikt. Dit is niet per se nodig, het kan ook zonder hulpvariabele:

```
ISR(USARTF0_RXC_vect)
{
  USARTF0.DATA = change_case(USARTF0.DATA);
}
```

Bovenstaande code werkt prima. De gegenereerde hex-code is exact hetzelfde. Waarschijnlijk gebruikt men dit niet graag, omdat het er vreemd uitziet. Vooral als de interruptfunctie de gegevens ongewijzigd terugkaatst, *lijkt* de toewijzing onzinnig.

```
ISR(USARTF0_RXC_vect)
{
  USARTF0.DATA = USARTF0.DATA;
}
```

Dit is echter niet het geval. Deze interruptfunctie ontvangt de gegevens en stuurt deze ongewijzigd terug. De `DATA` links van het isgelijkteken is de `DATA (transmit)` van de zender en de `DATA` rechts van het isgelijkteken is de `DATA (receive)` van de ontvanger.

19.7 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. Er zijn twee interruptmogelijkheden voor het verzenden, namelijk *transmit complete* en *data transmit empty*. De interruptroutine voor het ontvangen zet de gelezen informatie in een globale variabele `data` en de interruptroutines voor het verzenden zetten `data` in het `DATA (transmit)`-register.

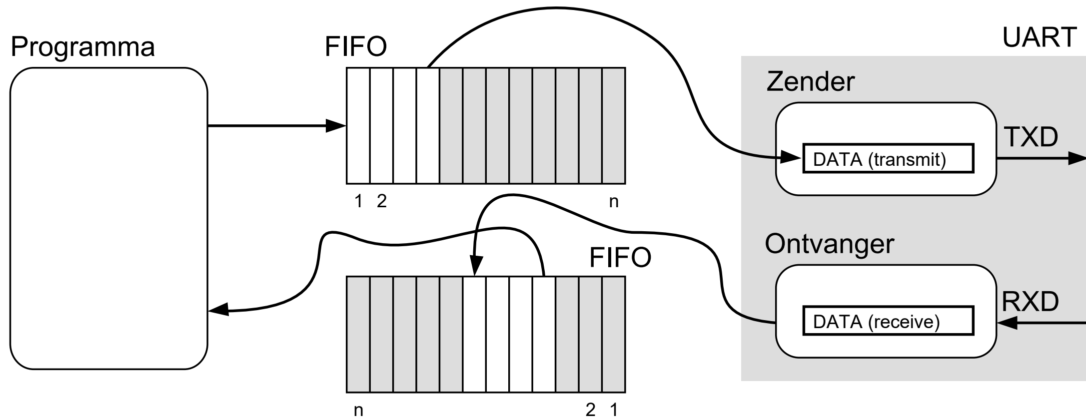
```
ISR(USARTF0_RXC_vect)
{
  data = USARTF0.DATA;
}
```

```
ISR(USARTF0_TXC_vect)
{
  USARTF0.DATA = data;
}
```

```
ISR(USARTF0_DRE_vect)
{
  USARTF0.DATA = data;
}
```

De interruptfunctie voor *data transmit empty* wordt getriggerd als het register `DATA (transmit)` leeg is. Dit heeft alleen zin als er geldige gegevens zijn om te versturen. Het zou erg toevallig zijn, als op het moment dat `DATA (transmit)` leeg is, de variabele `data` ook weer nieuwe te versturen gegevens bevat.
Ook bij het versturen van gegevens met *transmit complete* is het heel toevallig als `data` weer nieuwe gegevens bevat. Bij de *receive complete* treedt een soortgelijk probleem op. Als de microcontroller de vorige waarde van `data` nog niet verwerkt heeft, wordt de huidige waarde van `data` overschreven met de laatst ontvangen gegevens.

De oplossing is om een buffer voor het lezen en voor het schrijven te gebruiken. In figuur 19.13 plaatst de ontvanger de gegevens in een buffer. Het programma leest — op een moment dat het programma het schikt — de gegevens uit de buffer. De te verzenden gegevens plaatst het programma in een andere buffer. Op een moment dat de zender het schikt, verstuurt de zender deze gegevens.



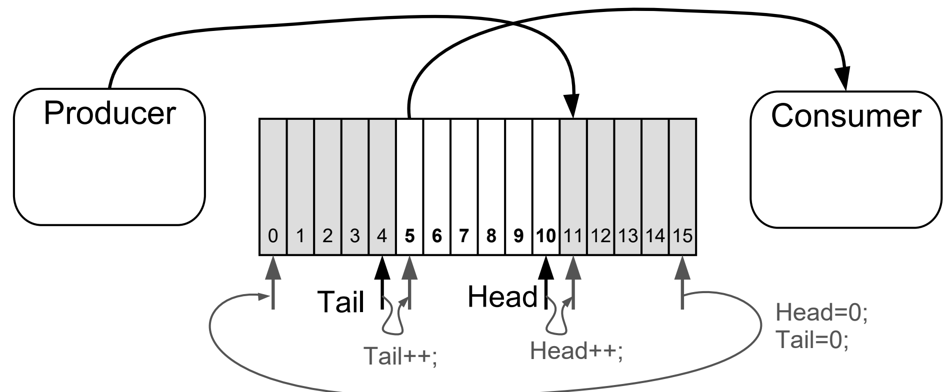
Figuur 19.13 : Communicatie met de UART met behulp van twee fifo-buffers. Het programma zet de te versturen bytes aan het begin van de fifo. De zender van de UART neemt de byte die aan het eind van de fifo staat. De ontvanger van de UART plaatst de te versturen databyte vooraan in de rij. Het programma pakt de bytes achteraan weg.

In figuur 19.13 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.

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 19.14 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;
```

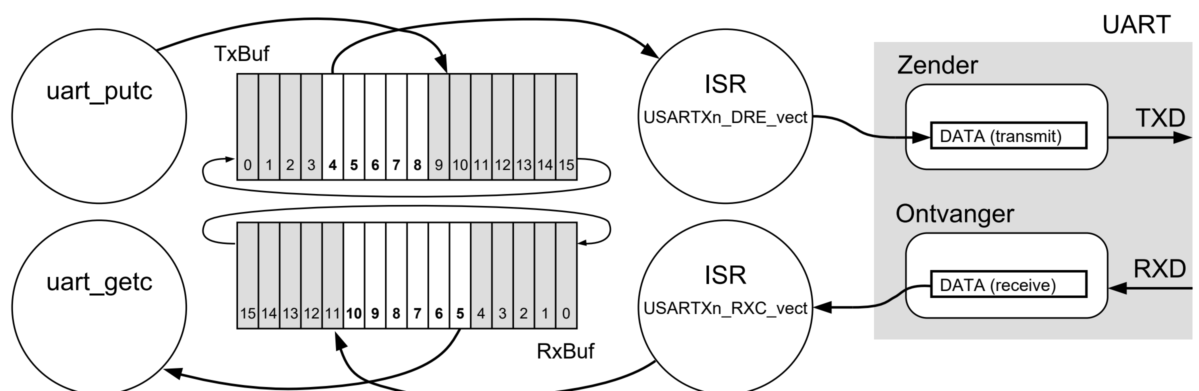


Figuur 19.14 : Een circulaire buffer tussen een producent en een consument van gegevens.
De producent hoogt eerst `Head` op en voegt dan de nieuwe databyte toe. De consument hoogt eerst `Tail` op en leest dan de databyte uit de buffer. `Head` en `Tail` worden weer nul gemaakt als de maximale waarde bereikt is.

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 $n-1$.

19.8 Circulaire buffers bij de communicatie met een UART

Figuur 19.15 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 19.15 : De toepassing van circulaire buffers bij de communicatie met een UART.
De ontvanger triggert `ISR(USARTxn_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(USARTxn_DRE_vect)`. Deze routine kopieert de te versturen byte naar de zender.

De interruptfunctie `USARTxn_RXC_vect` zet de nieuwe gegevens in de buffer `RxBuf` en de functie `uart_getc` leest de gegevens volgens het fifo-principe uit de buffer.
De functie `uart_putc` schrijft de gegevens naar de buffer `TxBuf` en de interruptfunctie `USARTxn_DRE_vect` stuurt de gegevens volgens het fifo-principe naar buiten.

Code 19.6 : Het ontvangen en versturen van gegevens met een circulaire buffer.

```
1  #define F_CPU   2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6  #define RX_BUFFER_SIZE 16
7  #define TX_BUFFER_SIZE 16
8  #define RX_BUFFER_MASK ( RX_BUFFER_SIZE - 1 )
9  #define TX_BUFFER_MASK ( TX_BUFFER_SIZE - 1 )
10
11 volatile uint8_t RxBuf[RX_BUFFER_SIZE];
12 volatile uint8_t RxHead = 0;
13 volatile uint8_t RxTail = 0;
14 volatile uint8_t TxBuf[TX_BUFFER_SIZE];
15 volatile uint8_t TxHead = 0;
16 volatile uint8_t TxTail = 0;
17
18 void set_usartctrl(USART_t *usart,
19                 uint8_t bscale, uint16_t  bsel);
20 void init_uart_bscale_bsel(USART_t *usart,
21                    int8_t bscale, int16_t bsel);
22
23 ISR(USARTF0_RXC_vect)
24 {
25   RxHead = (RxHead + 1) & RX_BUFFER_MASK;
26   RxBuf[RxHead] = USARTF0.DATA;
27 }
28
29 unsigned char uart_getc(void)
30 {
31   while (RxHead == RxTail) ;
32   RxTail = (RxTail+1) & RX_BUFFER_MASK;
33
34   return RxBuf[RxTail];
35 }
36
37 ISR(USARTF0_DRE_vect)
38 {
39   if ( TxHead != TxTail ) {
40     TxTail = (TxTail+1) & TX_BUFFER_MASK;
41     USARTF0.DATA = TxBuf[TxTail];
42   } else {
43     USARTF0.CTRLA |= USART_DREINTLVL_OFF_gc;
44   }
45 }
```

```
47 void uart_putc(uint8_t data)
48 {
49   uint8_t tmp;
50
51   tmp = (TxHead+1) & TX_BUFFER_MASK;
52   while (tmp == TxTail) ;
53   TxBuf[tmp] = data;
54   TxHead = tmp;
55
56   USARTF0.CTRLA |= USART_DREINTLVL_LO_gc;
57 }
58
59 void uart_puts(char *s)
60 {
61   char c;
62
63   while ( (c = *s++) ) {
64     uart_putc(c);
65   }
66 }
67
68 int main(void)
69 {
70   int c;
71
72   init_uart_bscale_bsel(&USARTF0, -7, 289);
73
74   PMIC.CTRL |= PMIC_LOLVLEN_bm;
75   sei();
76
77   while(1) {
78     c = uart_getc();
79     uart_puts("Character: '");
80     uart_putc(c);
81     uart_puts("'\n");
82   }
83 }
```

Het programma staat in code 19.6 en gebruikt USART0 van poort F. Vanaf regel 18 staan er twee prototypen. De functie `set_usartctrl` staat in code 19.2 en de initialisatiefunctie `uart_init_bscale_bsel` is volledig identiek met die uit code 19.5. Ook in dit geval activeert de laatste functie alleen de interruptvlag `RXCIF`. De interruptvlag `DREIF` wordt door de functie `uart_putc` aangezet.

Vanaf regel 6 worden de buffers gedefinieerd. 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 interruptfunctie `ISR(USARTF0_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` wacht totdat er nieuwe gegevens in de buffer staan, schuift dan de index `RxTail` een positie op en geeft de byte, die op locatie `RxTail` staat, terug.

De functie `uart_putc` schuift de index `RxHead` een positie op, kopieert de te schrijven byte naar die nieuwe positie en zet de interruptvlag `DREIF` aan. De interruptfunctie `ISR(USARTF0_DRE_vect)` wordt actief als het `DATA (transmit)` register leeg is. Deze functie schuift als er gegevens in `TxBuf` staan de index `TxTail` één positie op en kopieert de byte die op deze positie staat naar `DATA (transmit)`. Als alle gegevens verstuurd zijn, zet deze functie de interruptvlag `DREIF` uit.

Het voorbeeld met de circulaire buffer is oorspronkelijk gebaseerd op *application note* AVR306 van Atmel en de UART-bibliotheek van Peter Fleury voor de ATmega. De namen van variabelen, definities en functies in code 19.6 komen overeen met de namen die Peter Fleury gebruikt.

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 19.6 test de meeste van deze situaties.

Dit voorbeeld gebruikt specifiek USART0 van poort F. De application note 1307, *Using the XMEGA USART*, beschrijft een meer algemene USART-driver voor de Xmega, die geschikt is voor iedere USART van de Xmega.

De interruptfunctie `ISR(USARTF0_RXC_vect)` test niet of de buffer vol is. De buffer `RxBuf` moet voldoende groot zijn, anders gaan eerder ontvangen gegevens verloren. De functie `uart_getc` wacht op regel 31 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 52 totdat er ruimte is in buffer `TxBuf` om gegevens weg te schrijven. De interruptfunctie `ISR(USARTF0_DRE_vect)` test of buffer `TxBuf` leeg is. Als deze leeg is, wordt de interruptvlag `DREIF` uitgezet. Dit voorkomt dat de zender data gaat versturen als de buffer leeg is. Als `uart_putc` nieuwe data in de buffer zet, wordt `DREIF` weer aangezet.

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.

19.9 De USART-driver van Atmel en een bijbehorende wrapper

Er bestaan verschillende UART-bibliotheken voor de microcontrollers van Atmel. Het voorbeeld in de vorige paragraaf is gebaseerd op de *application note* AVR306 van Atmel. Het nadeel van het voorbeeld van code 19.6 is dat het oorspronkelijk bestemd is voor een ATmega en dat deze implementatie niet generiek is. Het is alleen geschikt voor USART0 van poort F.

Code 19.7 : Ontvangen en versturen met de USART-driver van Atmel en de wrapper.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6  #define ENABLE_UART_F0   1        // necessary for wrapper
7  #define F0_BAUD          115200
8  #define F0_CLK2X         0
9  #include "uart.h"                 // header file wrapper
10
11 int main(void)
12 {
13   uint16_t c;
14
15   init_uart(&uartF0, &USARTF0, F_CPU, F0_BAUD, F0_CLK2X);
16
17   PMIC.CTRL |= PMIC_LOLVLEN_bm;
18   sei();
19
20   while(1) {
21     if ( (c = uart_getc(&uartF0)) == UART_NO_DATA) {
22       continue;
23     }
24
25     uart_puts(&uartF0, "Character: '");
26     uart_putc(&uartF0, c);
27     uart_puts(&uartF0, "'\n");
28   }
29 }
```

Deze paragraaf gebruikt de driver, die hoort bij de *application note* AVR1307. Deze driver is geschikt voor de Xmega en gebruikt ook twee circulaire buffers. Bovendien is de opzet generiek. De driver is geschikt voor alle USART's en kan ook gebruikt worden, bij een applicatie met meerdere UART's.

Het nadeel is dat deze USART-driver nogal elementair is. Functies als `putc` en `getc` ontbreken. Ondanks de uitvoerige *application note* en de uitgebreide beschrijving bij de software moet een nieuwe gebruiker veel moeite doen om deze driver te begrijpen en te kunnen toepassen.

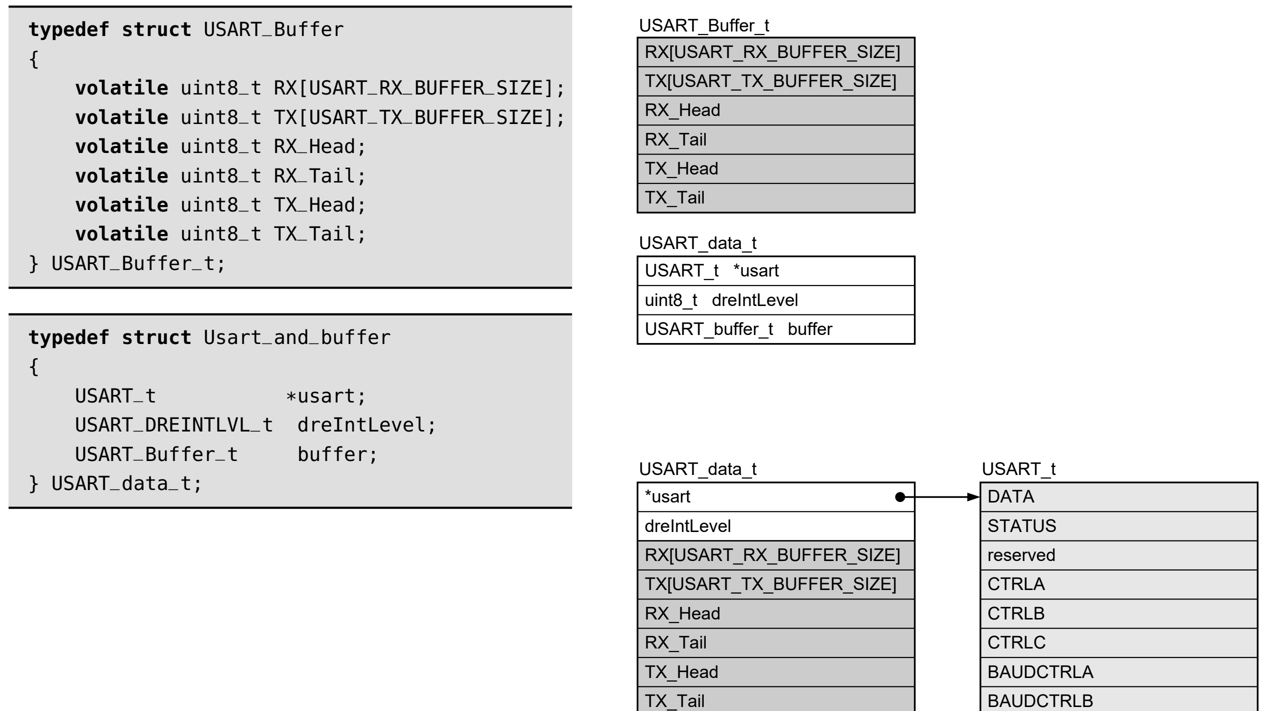
Deze paragraaf beschrijft een *wrapper*, die om de USART-driver van Atmel heen is gebouwd. Het programma uit code 19.7 heeft een gelijke functionaliteit als code 19.6, maar maakt gebruikt van de USART-driver met de *wrapper*. Bij de definities is op regel 6 aangegeven dat USART`0` van poort F gebruikt wordt.
Het hoofdprogramma gebruikt de initialisatiefunctie `init_uart` om deze UART de juiste baudsnelheid te geven. In de oneindige lus leest de functie `uart_getc` een karakter en verzenden de functies `uart_putc` en `uart_puts` de tekst terug.

Om de USART-driver met de wrapper te kunnen gebruiken zijn twee c-bestanden en drie h-bestanden nodig: `usart_driver.c`, `uart.c`, `avr_compiler.h`, `usart_driver.h` en `uart.h`. De h-bestanden `avr_compiler.h` en `usart_driver.h` horen bij de USART-driver. Het h-bestand `uart.h` hoort bij *wrapper* en sluit, als de twee andere h-

bestanden niet zijn ingesloten, deze automatisch in. De driversoftware staat in `usart_driver.c` en de *wrapper*-software in `uart.c`.

Opbouw USART-driver

De USART-driver gebruikt twee datastructuren. De datastructuur `USART_Buffer_t` definieert, op dezelfde manier als in code 19.6 de circulaire buffers `RX` en `TX` en de bijbehorende indices. De datastructuur `USART_data_t` bevat drie velden: een pointer naar een USART, een interruptniveau en de datastructuur `USART_Buffer_t`.



```
typedef struct USART_Buffer
{
    volatile uint8_t RX[USART_RX_BUFFER_SIZE];
    volatile uint8_t TX[USART_TX_BUFFER_SIZE];
    volatile uint8_t RX_Head;
    volatile uint8_t RX_Tail;
    volatile uint8_t TX_Head;
    volatile uint8_t TX_Tail;
} USART_Buffer_t;
```

```
typedef struct Usart_and_buffer
{
    USART_t           *usart;
    USART_DREINTLVL_t  dreIntLevel;
    USART_Buffer_t     buffer;
} USART_data_t;
```


Figuur 19.16 : De datastructuren van de USART-driver. Links staan de typedefinities van `USART_Buffer_t` en `USART_data_t`. Rechtsboven zijn deze twee datastructuren getekend. Datastructuur `USART_data_t` bevat in feite `USART_Buffer_t` en bevat een pointer naar de datastructuur van de UART, zoals de tekening rechtsonder laat zien.

De driver definieert een functie `USART_InterruptDriver_Initialize` die voor een bepaalde USART de datastructuur initialiseert. Het interruptniveau van de *data register empty* wordt ingesteld en de indices van de buffers worden `0` gemaakt. Deze functie wordt gebruikt bij het definiëren van de datastructuur voor de UART's. Onderstaand voorbeeld definieert twee datastructuren `uartD0` en `uartD1` voor respectievelijk de USART `0` en 1 van poort D:

```
USART_data_t uartD0, uartD1;

USART_InterruptDriver_Initialize(uartD0, &USARTD0, USART_DREINTLVL_LO_gc);
USART_InterruptDriver_Initialize(uartD1, &USARTD1, USART_DREINTLVL_MED_gc);
```

De betreffende USART's moeten nog wel worden ingesteld en de pinnen voor de `TXD`-aansluitingen moeten nog als uitgang worden gedefinieerd.

Code 19.8 : De initialisatiefunctie van de USART-driver.

```
81  void USART_InterruptDriver_Initialize(USART_data_t * usart_data,
82                                        USART_t * usart,
83                                        USART_DREINTLVL_t dreIntLevel)
84  {
85    usart_data->usart = usart;
86    usart_data->dreIntLevel = dreIntLevel;
87
88    usart_data->buffer.RX_Tail = 0;
89    usart_data->buffer.RX_Head = 0;
90    usart_data->buffer.TX_Tail = 0;
91    usart_data->buffer.TX_Head = 0;
92  }
```

In de inleiding van deze paragraaf is gesteld dat de USART-driver geen `putc` en `getc` kent. Dat is niet helemaal waar. De driver kent namelijk wel een `USART_PutChar` en een `USART_GetChar`, alleen zijn beide functies niet compleet. Functie `USART_PutChar` test niet of de buffer vol is en `USART_GetChar` test niet of de buffer leeg is.

Bovendien ontbreken in de bibliotheek van de USART-driver de twee interruptfuncties die voor de circulaire buffers nodig zijn. De driver beschikt daarentegen wel over twee functies `USART_RXComplete` en `USART_DataRegEmpty` die gebruikt kunnen worden om de *receive complete*- en de *data register empty*-interruptfunctie te maken.

Opbouw wrapper

De wrapper bestaat uit een c-bestand en een h-bestand. Het eerste bestand bevat onder andere de initialisatiefuncties, de functie `uart_puts` en de functies `uart_getc` en `uart_putc` die in code 19.9 staan. Deze functies zijn inhoudelijk identiek met de overeenkomstige functies uit code 19.6. Er zijn echter twee verschillen. Ten eerste hebben de functies een extra parameter `uart` voor het doorgeven van de juiste datastructuur. Ten tweede is er bij de `uart_getc` geen **`while`** gebruikt, maar een **`if`** statement. Deze `uart_getc` blijft niet wachten op data, maar geeft `UART_NO_DATA` terug als er geen data aanwezig is. De functies `uart_getc` en `uart_putc` gebruiken verschillende functies van de USART-driver.

Het h-bestand `uart.h` van de wrapper bevat, naast de definitie van `UART_NO_DATA` en de prototypen van alle functies, de datastructuren voor alle UART's en de benodigde interruptfuncties.

Op zich is het niet gebruikelijk om functies in een h-bestand te zetten. Functies staan altijd in een c-bestand. De *linker* gebruikt alleen de functies uit het gecompileerde c-bestand, die nodig zijn om het programma samen te stellen. Omdat interruptfuncties los staan van het hoofdprogramma, zouden de beschreven interruptfuncties altijd allemaal geïmplementeerd worden. Bij zeven UART's zijn er veertien interruptfuncties nodig. Als deze alle veertien in het c-bestand worden opgenomen, worden deze allemaal geïmplementeerd. Anderzijds is het verstandig om een c-bestand van een driver of wrapper niet iedere keer aan te passen.

Code 19.9 : De functies `uart_getc` en `uart_putc` van de wrapper.

```
51  uint16_t uart_getc(USART_data_t *uart)
52  {
53    uint8_t data;
54
55    if ( ! USART_RXBufferData_Available(uart) ) {
56      return UART_NO_DATA;
57    }
58
59    data = USART_RXBuffer_GetByte(uart);
60
61    return (data & 0x00FF);
62  }
63
64  void uart_putc(USART_data_t *uart, uint8_t data)
65  {
66    if ( USART_TXBuffer_FreeSpace(uart) ) {
67      USART_TXBuffer_PutByte(uart, data);
68    }
69  }
```

Door de interruptfuncties in het h-bestand op te nemen kunnen deze voorwaardelijk worden ingesloten. In code 19.7 staan de regels:

```
6  #define ENABLE_UART_F0   1        // necessary for wrapper
       ...
9  #include "uart.h"                 // header file wrapper
```

De definitie van de macro `ENABLE_UART_F0` staat voor de insluiting van `uart.h`. Bij het evalueren van het h-bestand is `ENABLE_UART_F0` dan bekend. Code 19.10 geeft het fragment uit het h-bestand dat alleen uitgevoerd wordt als deze macro gedefinieerd is. Regel 228 declareert de datastructuur `uartF0` en op regel 230 en 235 staan de *receive complete*- en de *data register empty*-interruptfuncties.

Code 19.10 : Het fragment uit uart.h met de datastructuur en de ISR's voor UART F0.

```
227  #if ENABLE_UART_F0
228  USART_data_t uartF0;
229
230  ISR(USARTF0_RXC_vect)
231  {
232    USART_RXComplete(&uartF0);
233  }
234
235  ISR(USARTF0_DRE_vect)
236  {
237    USART_DataRegEmpty(&uartF0);
238  }
239  #endif
```

Bij de drivers van Atmel is de buffergrootte instelbaar, maar deze is dan wel hetzelfde voor alle UART's.

Voor de andere UART's bevat het h-bestand identieke definities. Er kunnen ook meerdere UART's tegelijkertijd gedefinieerd worden:

```
#define ENABLE_UART_D0   1    // UART0 from port D
#define ENABLE_UART_D1   1    // UART1 from port D
#include "uart.h"
```

Met alleen de declaratie van de datastructuur en de definitie van interruptfuncties is de UART nog niet operationeel. De UART moet de juiste instelling krijgen en de bijbehorende RX- en TX-pin moeten respectievelijk in- en uitgang zijn.

Code 19.11 : De initialisatiefunctie voor de UART's.

```
233  void init_uart(USART_data_t *uart, USART_t *usart, uint32_t f_cpu, uint32_t baud, uint8_t clk2x)
234  {
235    uint16_t bsel;
236    int8_t bscale;
237
238    bscale = calc_bscale(f_cpu, baud, clk2x);
239    bsel   = calc_bsel(f_cpu, baud, bscale, clk2x);
240
241    USART_InterruptDriver_Initialize(uart, usart, USART_DREINTLVL_LO_gc);
242    USART_Format_Set(uart->usart, USART_CHSIZE_8BIT_gc, USART_PMODE_DISABLED_gc, !USART_SBMODE_bm);
243    USART_Rx_Enable(uart->usart);
244    USART_Tx_Enable(uart->usart);
245    USART_RxdInterruptLevel_Set(uart->usart, USART_RXCINTLVL_LO_gc);
246    USART_Baudrate_Set(uart->usart, bsel, bscale);
247
248    set_usart_txrx_direction(uart->usart);
249  }
```

De initialisatiefunctie `init_uart` staat in code 19.11 en wijkt inhoudelijk weinig af van die uit code 19.5. In feite is alleen de initialisatiefunctie voor de USART-driver toegevoegd en zijn de functies uit de USART-driver gebruikt.

Code 19.12 : De functie voor het berekenen van `BSEL`.

```
182  uint16_t calc_bsel(uint32_t f_cpu, uint32_t baud, int8_t scale, uint8_t clk2x)
183  {
184    uint8_t factor = 16;
185
186    factor = factor >> (clk2x & 0x01);
187    if ( scale < 0 ) {
188      return round( (((double)(f_cpu)/(factor*(double)(baud))) - 1) * (1<<-(scale)) );
189    } else {
190      return round( ((double)(f_cpu)/(factor*(double)(baud))/(1<<(scale))) - 1);
191    }
192  }
```

De `init_uart` heeft niet de instelwaarden `BSCALE` en `BSEL` als parameters, maar de klokfrequentie `f_cpu`, de baudsnelheid `baud` en de `clk2x`. De functie `calc_bscale` bepaalt uit deze drie parameters de waarde `bscale` en vervolgens berekent de functie `calc_bsel` uit deze zelfde parameters en `bscale` de waarde van `bsel`.
De functie `calc_sel` uit code 19.12 is de implementatie van formule 19.2. Deze berekening maakt gebruik van de math-bibliotheek.

De schaalfactor `BSCALE` is geschikt als deze met de bijbehorende `BSEL` een baudsnelheid oplevert die niet te veel afwijkt van de gewenste waarde. Bij een lage schaalfactor waarde van `BSCALE` is de fout relatief klein. Aan de andere kant is `BSEL`

Code 19.13 : De functie voor het berekenen van BSCALE en BSEL.

```
205 int8_t calc_bscale(uint32_t f_cpu, uint32_t baud, uint8_t clk2x)
206 {
207   int8_t   bscale;
208   uint16_t bsel;
209
210   for (bscale = -7; bscale<8; bscale++) {
211     if ( (bsel = calc_bsel(f_cpu, baud, bscale, clk2x)) < 4096 ) return bscale;
212   }
213
214   return bscale;
215 }
```

juist groot als BSCALE laag is. Als BSCALE de laagste waarde (—7) heeft, zal bij lage snelheden BSEL groter zijn dan 4095 — de maximale waarde van BSEL.
De functie calc_bscale staat in code 19.13 en berekent voor iedere schaalfactor de bijbehorende BSEL. De functie begint daarbij bij de laagste waarde. Als de berekende BSEL kleiner is dan 4096, is een geschikte combinatie van BSCALE en BSEL gevonden. Als er geen geschikte waarde is, geeft de functie de waarde 8 terug.

Code 19.14 : De functie set_usart_txrx_direction.

```
100 void set_usart_txrx_direction(USART_t *usart)
101 {
102   #ifdef USARTC0
103    if ( (uint16_t) usart == (uint16_t) &USARTC0 ) {
104      PORTC.DIRSET = PIN3_bm;
105      PORTC.DIRCLR = PIN2_bm;
106      return;
107    }
108   #endif
            ...
151   #ifdef USARTF1
152    if ( (uint16_t) usart == (uint16_t) &USARTF1) {
153      PORTF.DIRSET = PIN7_bm;
154      PORTF.DIRCLR = PIN6_bm;
155      return;
156    }
157   #endif
158 }
```

De UART heeft een ingang RXD en een uitgang TXD. De richting van deze aansluitpinnen moet correct worden ingesteld. In het algemeen wordt niet iedere UART gebruikt en niet elke Xmega heeft hetzelfde aantal UART's. Zo heeft de Xmega128a4u bijvoorbeeld vijf en de Xmega128a1u acht UART's.
De ingangsparameter usart van de functie set_usart_txrx_direction uit code 19.14 is een pointer die wijst naar een USART. De functie zoekt tussen de beschikbare USART's met welke deze usart overeenkomt. Regel 102 test of de gebruikte Xmega een USARTC0 heeft. Als usart naar het adres van deze datastructuur wijst, is de USART gevonden en krijgen de betreffende aansluitpinnen de juiste richting en wordt de functie beëindigd.

19.10 Het versturen van getallen via de UART

Code 19.15 is identiek aan code 19.7, 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.

Code 19.15 : Het versturen van getallen met de wrapper voor de USART-driver.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6  #define ENABLE_UART_F0   1
7  #define F0_BAUD          115200
8  #define F0_CLK2X         0
9  #include "uart.h"
10
11 int main(void)
12 {
13   uint16_t c;
14   char buffer[3];
15
16   init_uart(&uartF0, &USARTF0, F_CPU, F0_BAUD, F0_CLK2X);
17
18   PMIC.CTRL    |= PMIC_LOLVLEN_bm;
19   sei();
20
21   while(1) {
22     if ( (c = uart_getc(&uartF0)) == UART_NO_DATA) {
23       continue;
24     }
25
26     uart_puts(&uartF0, "Character: '");
27     uart_putc(&uartF0, c);
28     uart_puts(&uartF0, "' Hex: 0x");
29     itoa(c, buffer, 16);
30     uart_puts(&uartF0, buffer);
31     uart_putc(&uartF0, '\n');
32   }
33 }
```

19.11 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 19.16 : Het gebruik van `FDEV_SETUP_STREAM` om een eigen `printf` te maken.

```
 1 #define F_CPU     2000000UL
 2 
 3 #include <avr/io.h>
 4 #include <avr/interrupt.h>
 5 #include <stdio.h>
 6 
 7 #define ENABLE_UART_F0   1
 8 #define F0_BAUD          115200
 9 #define F0_CLK2X         0
10 #include "uart.h"
11 
12 int uart_fputc(char c, FILE *stream)
13 {
14   uart_putc(&uartF0,c);
15   return 0;
16 }
17 
18 FILE uart_stdout = FDEV_SETUP_STREAM(uart_fputc, NULL,_FDEV_SETUP_WRITE);
19 
20 int main(void)
21 {
22   uint16_t c;
23 
24   init_uart(&uartF0, &USARTF0, F_CPU,F0_BAUD, F0_CLK2X);
25   stdout = &uart_stdout;
26 
27   PMIC.CTRL    |= PMIC_LOLVLEN_bm;
28   sei();
29 
30   while(1) {
31     if ( (c = uart_getc(&uartF0)) == UART_NO_DATA) {
32       continue;
33     }
34     printf("Character: '%c' Hex: %#x\n", c, c);
35   }
36 }
```

Code 19.16 gebruikt een eigen `printf`. Op regel 12 staat een functie `uart_fputc` die met behulp van de functie `uart_putc` uit de *wrapper*-bibliotheek een karakter naar de UART schrijft. De functie `uart_fputc` 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 18 een `FILE`-structuur `uart_stdout`. De toekenning van regel 25 zorgt er voor dat de standaard filepointer `stdout` naar deze `FILE`-structuur wijst. De `printf` van regel 34 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 19.17 : Het gebruik van `FDEV_SETUP_STREAM` om een eigen `printf` en `scanf` te maken.

```
#define F_CPU     2000000UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>

#define ENABLE_UART_F0   1
#define F0_BAUD          115200
#define F0_CLK2X         0
#include "uart.h"

int uart_fputc(char c, FILE *stream);
int uart_fgetc(FILE * stream);

FILE uart_stdinout = FDEV_SETUP_STREAM(uart_fputc, uart_fgetc, _FDEV_SETUP_RW);

int uart_fputc(char c, FILE *stream)
{
  if (c == '\n') uart_putc(&uartF0, '\r');
  uart_putc(&uartF0, c);

  return 0;
}

int uart_fgetc(FILE * stream)
{
  int c;

  while ( (c = uart_getc(&uartF0)) == UART_NO_DATA) ;

  return c;
}

int main(void)
{
  uint8_t c;

  init_uart(&uartF0, &USARTF0, F_CPU, F0_BAUD, F0_CLK2X);
  stdout = stdin = &uart_stdinout;

  PMIC.CTRL    |= PMIC_LOLVLEN_bm;
  sei();

  while(1) {
    scanf("%c", &c);
    printf("Character: '%c' Hex: %#x\n", c, c);
  }
}
```

Code 19.17 gebruikt zowel de `printf`- als de `scanf`-functie. De `FILE`-structuur `uart_stdinout` definieert een *stream* met een lees- en een schrijffunctie. De functie `uart_fputc` is identiek aan die uit code 19.16, alleen is er een **`if`** toegevoegd die bij een *end-of-line* (`'\n'`) een extra *carriage return* (`'\r'`) verstuurd. Nieuwe regels beginnen dan bij een Window's terminal ook op het begin van de regel.

De standaard uitvoer of `stdout` wordt onder andere door de functies `putchar()`, `puts()` en `printf()` gebruikt. De standaard invoer `stdin` wordt onder andere door `getchar()`, `puts()` en `printf()` gebruikt. De standaard in- en uitvoer worden ook aangeduid als standaard *streams*.

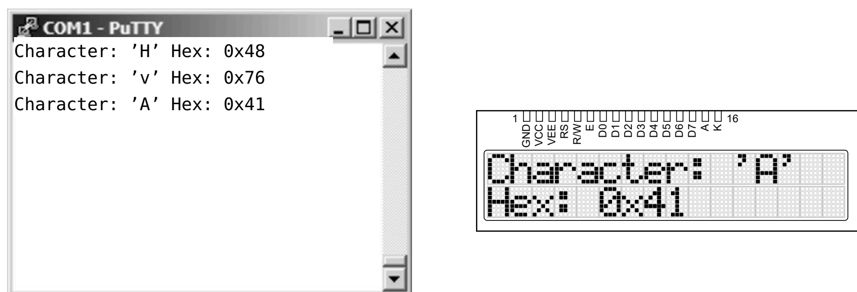
De leesfunctie `uart_fgetc` gebruikt `uart_getc` uit de *wrapper* en blijft wachten tot er een karakter gelezen is. De functie `uart_fgetc` heeft net als de standaard `fgetc` een ingangsvariabele `FILE *` en een retourtype **`int`**. De structuur `uart_stdinout` is op regel 15 gekoppeld aan zowel `stdout` als aan `stdin`. De macro `FDEV_SETUP_STREAM` krijgt nu beide functies mee en de setupmodus is `_FDEV_SETUP_RW`.
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.

Naast `stdout` bestaat er ook een uitvoer `stderr`, die speciaal bestemd is voor waarschuwingen en foutmeldingen.

Meerdere streams tegelijk

Het is ook mogelijk om meerdere *streams* te gebruiken. Het programma van code 19.18 leest karakters van de UART en schrijft deze net als in code 19.17 geformatteerd terug, maar stuurt dezelfde informatie ook naar een LCD. Op regel 17 en 18 zijn nu twee `FILE`-structuren gedefinieerd. In het hoofdprogramma worden deze twee structuren op regel 33, 34 en 37 gekoppeld aan de drie filepointers `uart_in`, `uart_out` en `lcd_out`.

Op regel 43 wordt met `fscanf` een karakter van de UART gelezen. Dit karakter wordt op regel 44 met `fprintf` geformatteerd naar de UART en op regel 46 naar de LCD geschreven. In het eerste geval is de filepointer bij `fprintf` `uart_out` en in het tweede geval is dat `lcd_out`. Een resultaat van het programma staat in figuur 19.17.



Figuur 19.17 : Het resultaat van code 19.18. Links staat de uitvoer van de UART en rechts 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 45 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 zijn. Alleen als er een niet afdrukbaar karakter wordt ingevoerd (bijvoorbeeld `^J`, `^M`) komt er ook tekst op andere posities terecht.
Het schrijven naar de UART kan ook gedaan worden met een `sprintf` en een `uart_puts` op de manier zoals in paragraaf 18.8 voor een LCD is beschreven.

Code 19.18 : Een eigen `printf` en `scanf` voor de UART en een eigen `printf` voor het LCD. In deze code staan alleen de prototypen van `uart_fgetc` en `uart_fputc`. Deze functies zijn identiek met de functies uit code 19.17.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <stdio.h>
6
7  #define ENABLE_UART_F0   1
8  #define F0_BAUD          115200
9  #define F0_CLK2X         0
10 #include "uart.h"
11 #include "lcd.h"
12
13 int uart_fputc(char c, FILE *stream);
14 int uart_fgetc(FILE * stream);
15 int lcd_fputc(char c, FILE *stream);
16
17 FILE uart_stdinout = FDEV_SETUP_STREAM(uart_fputc, uart_fgetc, _FDEV_SETUP_RW);
18 FILE lcd_stdout    = FDEV_SETUP_STREAM(lcd_fputc, NULL, _FDEV_SETUP_WRITE);
19
20 int lcd_fputc(char c, FILE *stream)
21 {
22   lcd_putc(c);
23
24   return 0;
25 }
26
27 int main(void)
28 {
29   char c;
30   FILE *uart_in, *uart_out, *lcd_out;
31
32   init_uart(&uartF0, &USARTF0, F_CPU, F0_BAUD, F0_CLK2X);
33   uart_in  = &uart_stdinout;
34   uart_out = &uart_stdinout;
35
36   lcd_init();
37   lcd_out = &lcd_stdout;
38
39   PMIC.CTRL |= PMIC_LOLVLEN_bm;
40   sei();
41
42   while(1) {
43     fscanf(uart_in, "%c", &c);
44     fprintf(uart_out, "Character: '%c' Hex: %#x\n", c, c);
45     lcd_clear();
46     fprintf(lcd_out, "Character: '%c'\nHex: %#x\n", c, c);
47   }
48 }
```

19.12 Een vaste stream voor USART0 van poort F

USART0 van poort F van de Xmega256a3u is bij het Xmega-bord uit bijlage J permanent verbonden met de Xmega32a4u, die in de communicatiemodus als RS232-USB-converter werkt. Bij dit bord ligt het voor de hand om via USART0 van poort F te communiceren en om voor deze UART een bibliotheekbestand te maken, die hiervoor de standaard streams instelt.

De regels 7 tot en met 32 uit code 19.17 bevatten de instelling van de streams voor USART0 van poort F en kunnen ook in een bestand `stream.c` worden gezet. Dit deel met de definities voor de streams is dan ook bij andere projecten bruikbaar. Het hoofdprogramma uit code 19.17 stelt van regel 38 tot en met 41 de USART0 van poort F en de standaard streams in. Door dit deel in een aparte functie `init_stream` te plaatsen en deze functie aan `stream.c` toe te voegen, kan hiermee de seriële verbinding worden geïnitialiseerd. In code 19.19 is code 19.17 herschreven voor `stream.c` en een bijhorend headerbestand `stream.h`, met daarin onder andere het prototype van `init_stream()`.

Op de internetsite van dit boek staat een zip met de bestanden `uart.c`, `uart.h`, `stream.c` en `stream.h`.
Het bestand `stream.c` bevat ook een macrodefinitie `clear_screen()`, die het terminalvenster leeg maakt.

Code 19.19 : Code 19.17 herschreven met de bibliotheek `stream.c`.

```
 1  #define F_CPU     2000000UL
 2
 3  #include <avr/io.h>
 4  #include <avr/interrupt.h>
 5  #include <stdio.h>
 6
 7  #include "stream.h"
 8
 9  int main(void)
10  {
11    uint8_t c;
12
13    init_stream(F_CPU);
14    sei();
15
16    while(1) {
17      scanf("%c", &c);
18      printf("Character: '%c' Hex: %#x\n", c, c);
19    }
20  }
```

Op regel 13 initialiseert `init_stream()` de standaard streams. Deze functie gebruikt de klokfrequentie voor de configuratie van de baudsnelheid. De insluiting van `stream.h` op regel 7 bevat het prototype van `init_stream()` en de macrodefinitie `BAUD` met een vaste baudsnelheid. De functie stelt de gevoeligheid van de interruptfuncties in op het lage niveau, maar zet het interruptmechanisme niet aan. Dit gebeurt in het hoofdprogramma op regel 14 na de initialisatie van de streams.

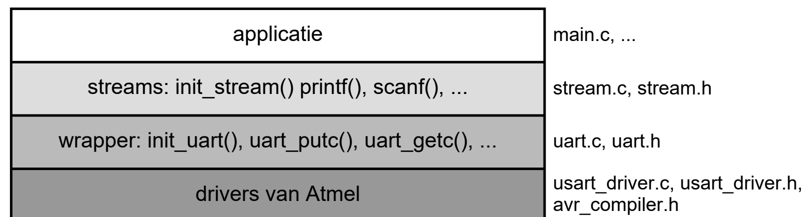
Code 19.15 en code 19.16 zijn ook te vereenvoudigen met `stream.c` en `stream.h`. Deze codes gebruiken de functies `uart_getc` en `uart_putc` waarvan de prototypes in `uart.h` staan. Daarom is in code 19.20 op regel 8 ook `uart.h` ingesloten. Voor deze insluiting staat nu niet de definitie `ENABLE_UART_F0`, omdat deze al gedefinieerd is in `stream.c` en de interruptfuncties daar ingesloten zijn.

De variabele `uartF0` is eveneens al in `stream.c` gedeclareerd. Daarom is op regel 9 met **`extern`** aangegeven dat deze variabele extern gedeclareerd is.

Code 19.20 : Het programma van code 19.16 herschreven voor `stream.c`.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <stdio.h>
6
7  #include "stream.h"
8  #include "uart.h"
9  extern USART_data_t uartF0;
10
11 int main(void)
12 {
13   uint16_t c;
14
15   init_stream(F_CPU);
16   sei();
17
18   while(1) {
19     if ( (c = uart_getc(&uartF0)) == UART_NO_DATA ) {
20       continue;
21     }
22     printf("Character: '%c' Hex: %#x\n", c, c);
23   }
24 }
```

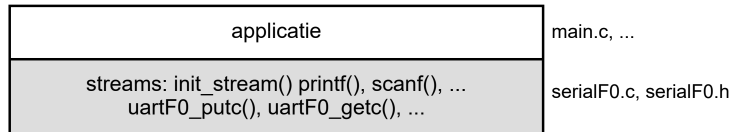
Figuur 19.18 toont de verschillende abstractielagen met het gebruik van `stream.c`. De applicatie gebruikt de normale `printf`- en `scanf`-functies die door de abstractielaag *streams* beschikbaar zijn gemaakt. Deze laag gebruikt zelf de laag *wrapper* met onder andere de definities van `uart_putc`- en `uart_getc`, die weer gebaseerd is op de USART-drivers van Atmel.



Figuur 19.18 : De verschillende abstractielagen bij de opzet met `stream.c`, `uart.c` en de drivers van Atmel.

Het nadeel van deze aanpak is dat er bij ieder project drie c-bestanden en vier h-bestanden nodig zijn en dat bij Atmel Studio de drie c-bestanden aan het project toegevoegd moeten worden. Een alternatief is om `uart_putc` en `uart_getc` rechtstreeks te implementeren en de stream-definities daar direct aan toe te voegen.

Op de site van dit boek staat naast de zip met `uart.c`, `stream.c` en de bijbehorende h-bestanden ook een zip met een c-bestand `serialF0.c` en een h-bestand `serialF0.h`. Er zijn, zoals figuur 19.19 laat zien, nu maar twee abstractielagen.



Figuur 19.19 : De abstractielagen bij de opzet met `serialF0.c`.

De initialisatiefunctie `init_stream()` heeft dezelfde naam als bij `stream.c`. Dit is bewust gedaan om de uitwisselbaarheid zo groot mogelijk te maken. Omdat er maar één stel standaard streams is, wordt óf de bibliotheek met `stream.c` óf die met `serialF0.c` gebruikt.
Code 19.19 van bladzijde 319 is met één kleine aanpassing ook met `serialF0.c` te gebruiken. Op regel 7 moet dan in plaats van `stream.h` het headerbestand `serialF0.h` worden ingesloten.

```
 7  #include "serialF0.h"
```

Intern gebruikt `serialF0.c` niet de drivers van Atmel. De buffers en de functies zijn anders opgebouwd, maar de basisprincipes komen ruwweg overeen met die uit paragraaf 19.8. Er is nu ook een buffer voor het zenden en voor het ontvangen. Twee gewone functies plaatsen de gegevens weer in de zendbuffer en halen deze uit de ontvangbuffer. Twee interruptroutines zorgen er weer voor dat de gegevens uit de zendbuffer worden verstuurd en in de ontvangbuffer worden geplaatst.
Bij deze implementatie hoeven de groottes van de buffers voor het zenden en het ontvangen alleen geen macht van twee te zijn. Alle afmetingen tot 256 voldoen als bufferdiepte.

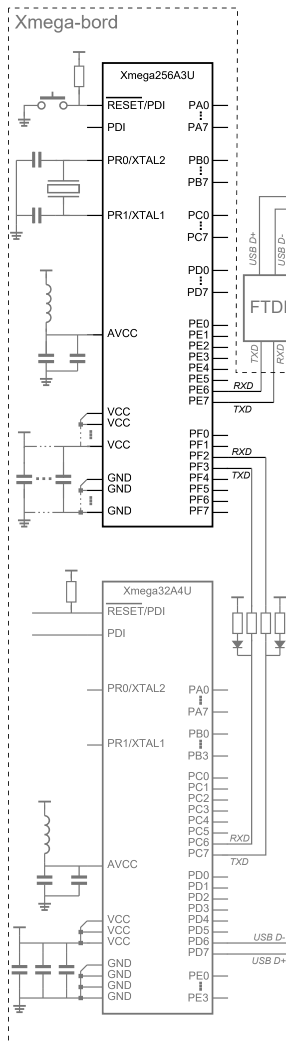
Code 19.21 : Code 19.20 herschreven met de bibliotheek `serialF0.c`.

```
 1  #define F_CPU     2000000UL
 2
 3  #include <avr/io.h>
 4  #include <avr/interrupt.h>
 5  #include <stdio.h>
 6
 7  #include "serialF0.h"
 8
 9  int main(void)
10  {
11    uint16_t c;
12
13    init_stream(F_CPU);
14    sei();
15
16    while(1) {
17      if ( (c = uartF0_getc()) == UART_NO_DATA ) {
18        continue;
19      }
20      printf("Character: '%c' Hex: %#x\n", c, c);
21    }
22  }
```

In code 19.21 is code 19.20 herschreven voor `serialF0.c` en `serialF0.h`. In plaats van `uart_getc()` gebruikt `serialF0.c` een functie `uartF0_getc()`. De functienamen zijn in `serialF0.c` bewust anders gekozen, zodat deze functies te gebruiken zijn naast de overeenkomstige functies uit `uart.c`.

Code 19.22 : Een programma dat twee UART's gebruikt

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <stdio.h>
6
7  #include "serialF0.h"
8  #define  ENABLE_UART_E1   1
9  #include "uart.h"
10
11 static int uart_fputc(char c, FILE *stream)
12 {
13   uart_putc(&uartE1,c);
14   return 0;
15 }
16
17 FILE e1_out = FDEV_SETUP_STREAM(uart_fputc, NULL,_FDEV_SETUP_WRITE);
18
19 int main(void)
20 {
21   uint16_t c;
22   FILE *f;
23
24   init_uart(&uartE1, &USARTE1, F_CPU, 38400, 0);
25   f = &e1_out;
26   PMIC.CTRL |= PMIC_LOLVLEN_bm;
27   init_stream(F_CPU);
28   sei();
29
30   while(1) {
31     c = getchar();
32     printf("Character: '%c' Hex: %#x\n", c, c);
33     fprintf(f, "Character: '%c' Hex: %#x\n", c, c);
34   }
35 }
```



Figuur 19.20 : Het Xmega-bord met twee UART-verbindingen. De Xmega256a3u heeft twee USB-verbindingen voor een virtuele COM-poort: één via USART0 van poort F en de Xmega32a4u en één via USART1 van poort E en de externe FTDI-chip.

In code 19.22 staat een programma dat USART0 van poort F en USART1 van poort E gebruikt. In figuur 19.20 toont een bijbehorend schema. De functies `getchar()` en `printf` gebruiken de uart van poort F, die met behulp van `serial.c` ingesteld is op de standaard streams. De functie `fprintf` gebruikt via de filepointer `f` de uart van poort E en gebruikt daarvoor `uart.c` en de drivers van Atmel.
De regels 8 tot en met 17 definiëren de stream `e1_out` en de insluiting op regel 7 definieert de standaard streams `stdin` en `stdout`. Regel 22 tot en met 26 initialiseren de filepointer `f` en de UART van poort E en regel 27 initialiseert de standaard stream via de UART van poort F.

Een alternatief voor de opzet van code 19.22, met de wrapper `uart.c` en de drivers van Atmel, is om `serialF0.c` te herschrijven voor USART1 van poort E en deze samen met `serialF0.c` te gebruiken.

20

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 Xmega is opgebouwd en hoe je deze kunt toepassen.

Onderwerpen

De behandelde onderwerpen zijn:

- Analoog-digitaalconversie en dan met name de ADC, die gebaseerd is op successieve approximatie en de pipelined-ADC van de Xmega.
- De opbouw van de ADC bij de Xmega: de ingangsselectie, het uitlezen van de uitgangsregisters, het instellen van de referentiespanning, de prescaling van het kloksignaal, de *single-ended* en de differentiële inputmodus.
- De handmatige en de automatische conversiemethoden.
- Het eventnetwerk met het eventmechanisme en de freerunningmodus.
- Fouten bij AD-conversie, offsetcorrecties en kalibratie.

Voorbeelden tonen verschillende manieren om de ADC te gebruiken:

- Handmatig converteren in de unsigned single-ended-modus.
- Handmatig converteren in de signed single-ended-modus met offsetcorrectie.
- Handmatig converteren met de differentiële ingangsmodus.
- Handmatig converteren met de differentiële ingangsmodus.
- Differentiële conversie met interrupt.
- Differentiële conversie op vaste tijdstippen met behulp van timer.
- Differentiële conversie van vier ingangen in freerunningmodus.

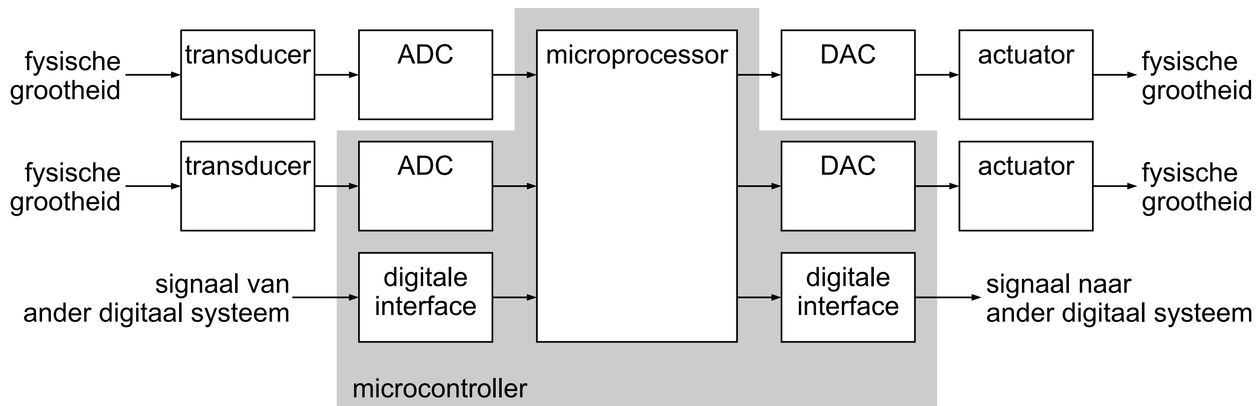
De microprocessor — het hart van een microcontroller — leest en schrijft alleen digitale signalen en voert daarop digitale rekenkundige en logische bewerkingen uit. In veel gevallen is de omgeving van de microprocessor niet digitaal. Er wordt bijvoorbeeld een druk, een temperatuur of een versnelling 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 van 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, geven 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.



Figuur 20.1 : 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 meestal een ADC, soms DAC en altijd verschillende digitale interfaces om met andere digitale systemen te communiceren.

De microprocessor moet analoge waarden kunnen lezen en apparaten analoog kunnen aansturen. Er zijn componenten nodig om dat te 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 20.1 toont de stappen die nodig zijn om een fysische 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. De Xmega's uit de B-, C- en D-serie hebben geen DAC. De Xmega's uit de A- en E-serie hebben één of twee DAC's. De Xmega256a3u heeft één tweekanaals DAC bij poort B genaamd DACB en twee ADC's een ADCA bij poort A en een ADCB bij poort B.

In paragraaf 23.1 komt de DAC van de Xmega256a3u uitgebreid aan de orde.

20.1 Analoog-digitaalconversie

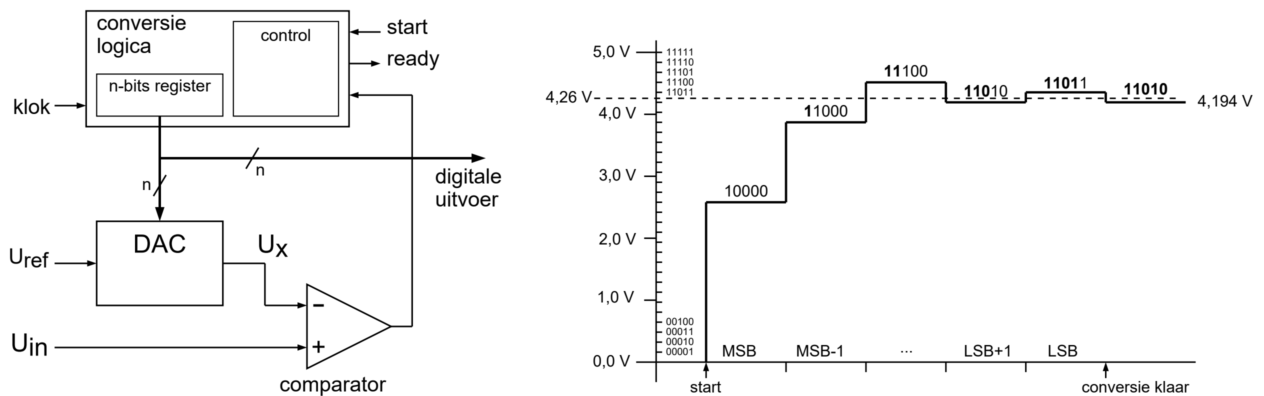
Er bestaan veel verschillende methoden om analoge signalen om te zetten in digitale signalen. Zeker als er een hoge nauwkeurigheid en een grote conversiesnelheid nodig zijn, zijn het complexe schakelingen. De belangrijkste en meest gebruikte omzetters zijn:

- *Ramp converter*
- *Dual slope converter*
- *Flash converter*
- *Successive approximation converter*
- *Sigma delta converter*
- *Pipelined converter*
- *Cyclic converter*

De ATmega gebruikt — net als veel andere microcontrollers — een 10-bits ADC, die gebaseerd is op successieve approximatie. Van deze ADC bestaan verschillende uitvoeringen, maar algemeen geldt dat deze converter in verhouding snel en redelijk goedkoop te produceren is, maar dat de nauwkeurigheid beperkt is. De Xmega gebruikt een 12-bits *Cyclic*-ADC bij de C-, D- en E-serie en een 12-bits *pipelined*-ADC bij de A- en B-serie. De *pipelined*-ADC is aanzienlijk sneller dan de *Cyclic*-ADC.

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. In figuur 20.2 staat een blokschema en een voorbeeld van de stapsgewijze benadering bij een 5-bits ADC.



Figuur 20.2 : **Een ADC op basis van successieve approximatie.** De linker figuur toont de opbouw van de ADC. De analoge comparator vergelijkt de door de DAC omgezette benadering U_x met het ingangssignaal U_{in}. De rechter figuur toont de stapsgewijze benadering van een ingangssignaal van 4,26 V voor een 5-bits ADC en een referentiespanning van 5 V. De digitale eindwaarde is `11010` en de DAC geeft 4,194 V af.

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 ingangsspanning.

Tijdens de omzetting staat in het dataregister steeds de benadering van het ingangssignaal. De DAC zet deze waarde om in een analoog signaal. De comparator vergelijkt dit signaal met het analoge ingangssignaal, dat geconverteerd moet worden. Als de conversie klaar is bevat het dataregister de beste benadering van het analoge ingangssignaal. Het analoge equivalent van deze benadering is de uitgangsspanning van U_x van de DAC en deze hangt af van de referentiespanning en van het aantal bits n:

$$U_x = \frac{\text{data}}{2^n - 1} U_{ref} \tag{20.1}$$

De nauwkeurigheid van dit type ADC is gelijk aan die van de DAC:

$$\Delta U_x = \frac{1}{2^n - 1} U_{ref} \qquad (20.2)$$

De nauwkeurigheid van successieve approximatie is ongeveer een bit. De meetfout is gelijk aan formule 20.2. Naast deze fout kent de ADC een *zero scale error* of *offset error*, *full scale error* of *gain error* en *niet-lineariteiten*.

De conversietijd t_{conv} van dit type ADC is evenredig met het aantal bits n en de periode T_{adc_clock} van het kloksignaal:

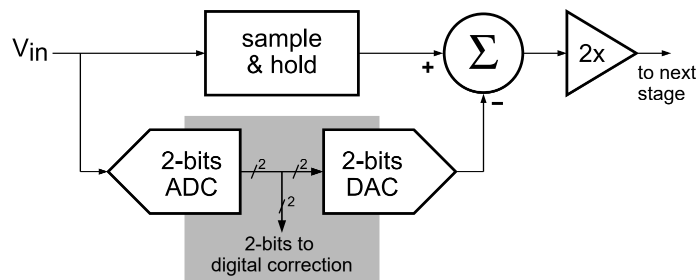
$$t_{conv} = nT_{adc_clock} \qquad (20.3)$$

De maximale klokfrequentie van een ADC gebaseerd op successieve approximatie ligt vaak rond de 100 kHz. Bij een 10-bits ADC is voor de conversie typisch 100 μs nodig.

Het principe van de pipelined ADC bij de Xmega

De ADC van de Xmega256a3u is een 12-bits pipelined-ADC. Deze ADC bestaat uit twaalf trappen. Een blokschema van deze trap — of *stage* — staat in figuur 20.3. De trap bestaat uit een 2-bits ADC, een 2-bits DAC, een *sample&hold*, een aftrekschakeling en een versterkerschakeling of *gain*.

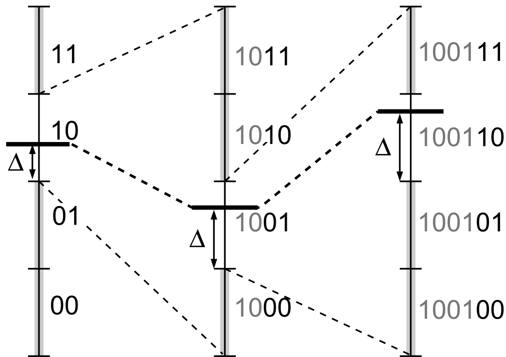
Het analoge ingangssignaal wordt bewaard door de *sample&hold* en wordt door de ADC omgezet in een 2-bits digitale waarde, die weer geconverteerd wordt naar een analoog signaal. De aftrekschakeling geeft het verschil tussen de werkelijke waarde en gevonden waarde — via de versterker en de analoge uitgang — door aan de volgende trap. Het grijze gebied in figuur 20.3 is digitaal en de rest van de trap is analoog.



Figuur 20.3 : De trap of *stage* van de pipelined ADC van de Xmega. Het analoge ingangssignaal wordt door de *sample&hold* vastgehouden en wordt omgezet door de ADC in een 2-bits digitale benadering. De digitale benadering wordt weer analoog gemaakt en van het oorspronkelijke analoge signaal afgetrokken. De restwaarde wordt met 2 vermenigvuldigd en doorgegeven aan de volgende trap.

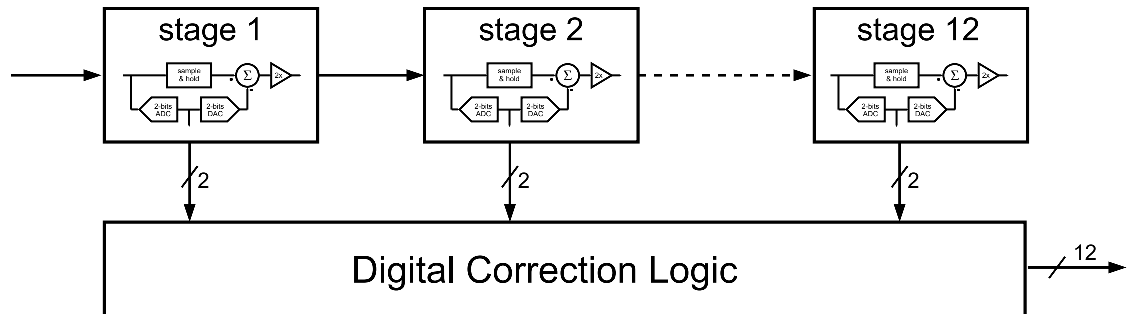
De 2-bits trap van figuur 20.3 verdeelt het meetgebied in vier delen. De digitale waarde geeft aan in welke deel de gezochte, analoge waarde zich bevindt. Het verschil Δ tussen de gezochte waarde en de gevonden waarde wordt versterkt en doorgegeven aan de volgende trap. Deze trap verdeelt het gevonden gebied weer in vier delen en bepaalt de volgende twee bits. In figuur 20.4 is dit proces voor drie trappen getekend.

De trap van de Xmega uit figuur 20.3 versterkt het verschil tussen de gezochte en de gevonden waarde met een factor 2 in plaats van de factor 4 waarop het voorbeeld uit figuur 20.4 is gebaseerd. Met een versterkingsfactor 4 zijn zes trappen nodig voor een 12-bits ADC. Om mogelijke fouten te corrigeren heeft de Xmega 12 trappen die elkaar overlappen. De 24-bits, die uit de twaalf trappen komen, bevatten dus redundante informatie.



Figuur 20.4 : Het principe van een pipelined ADC.
Een pipelined-ADC is opgebouwd uit meerdere trappen. De individuele 2-bits trap of *stage* is in figuur 20.3 getekend.
De eerste trap geeft de twee meest significante bits van de digitale waarde. De gezochte waarde ligt in het niet grijze gebied. Iedere trap geeft de *afwijking* met een versterkingsfactor door aan de volgende trap, die daarna de volgende twee bits bepaalt.

In figuur 20.5 staat de pipelined-ADC van de Xmega256a3u. Deze 12-bits ADC bevat twaalf 2-bits trappen en een digitale schakeling, die uit de 24-bits een 12-bits digitale waarde berekent.

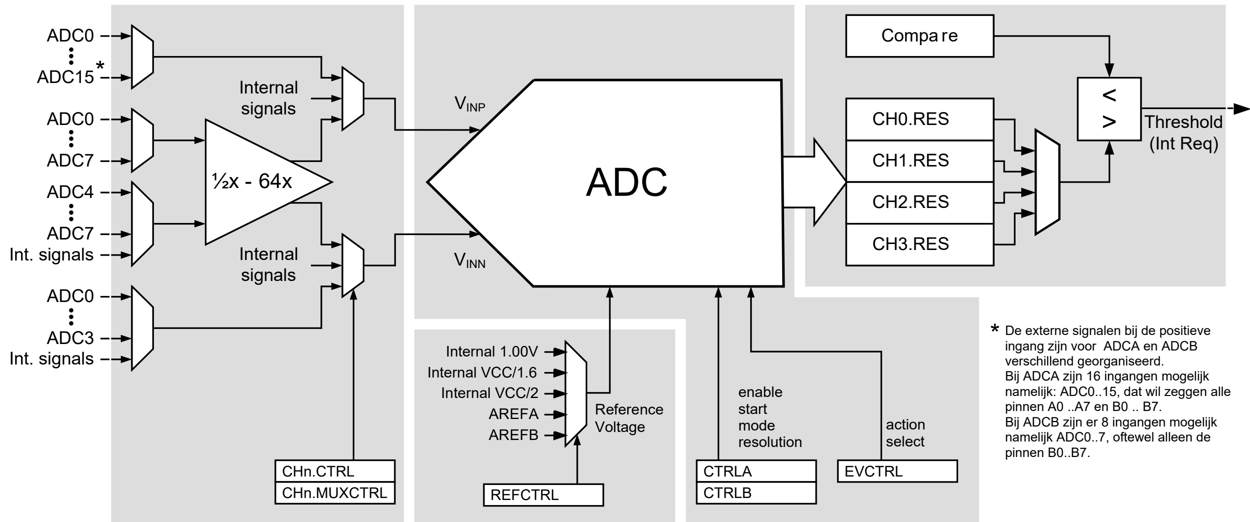


Figuur 20.5 : **De opbouw van de pipelined-ADC.** De pipelined-ADC van de Xmega256a3u bestaat uit twaalf 2-bits trappen en een digitaal blok, dat uit twaalf maal 2-bits de 12-bits uitkomst bepaalt.

20.2 De opbouw van de ADC bij de Xmega

Figuur 20.6 toont de opbouw van de ADC uit de datasheet van de Xmega256a3u. Er zijn vier blokken te onderscheiden: een ingangsselectieblok, een blok voor het referentiesignaal, een uitgangsblok en de feitelijke ADC.

De ADC's van de Xmega256a3u hebben vier zogenoemde *channels* of kanalen. Ieder kanaal heeft een eigen ingangsblok en een register voor het resultaat. De kanalen moeten wel allemaal dezelfde modus gebruiken. Doordat de kanalen de pipeline van de ADC gebruiken, kunnen er vier onafhankelijke metingen tegelijkertijd worden gedaan.



Figuur 20.6 : De opbouw van de ADC van de Xmega uit de AU-manual. Er zijn vier functionele blokken te onderscheiden. Links staat het blok voor de selectie van de ingangen, onderaan staat een blok voor de referentiespanning, aan de rechterkant staat het uitgangsblok en in het midden bevindt zich de feitelijke ADC.

De ingangsselectie van de ADC

De pipelined-ADC is een differentiële ADC met twee ingangen: een positieve ingang V_{INP} en een negatieve ingang V_{INN}. Alle aansluitingen van poort A kunnen met één of met beide ingangen van de ADCA worden verbonden en alle aansluitingen van poort B kunnen met één of met beide ingangen van de ADCB worden verbonden. De ingangen van poort B zijn ook bij de positieve ingang van ADCA te gebruiken, zie ook tabel H.2. De ingangsselectieblokken van de ADC's bevatten bovendien een regelbare verschilversterker waarmee de ingangssignalen ook versterkt of verzwakt kunnen worden.

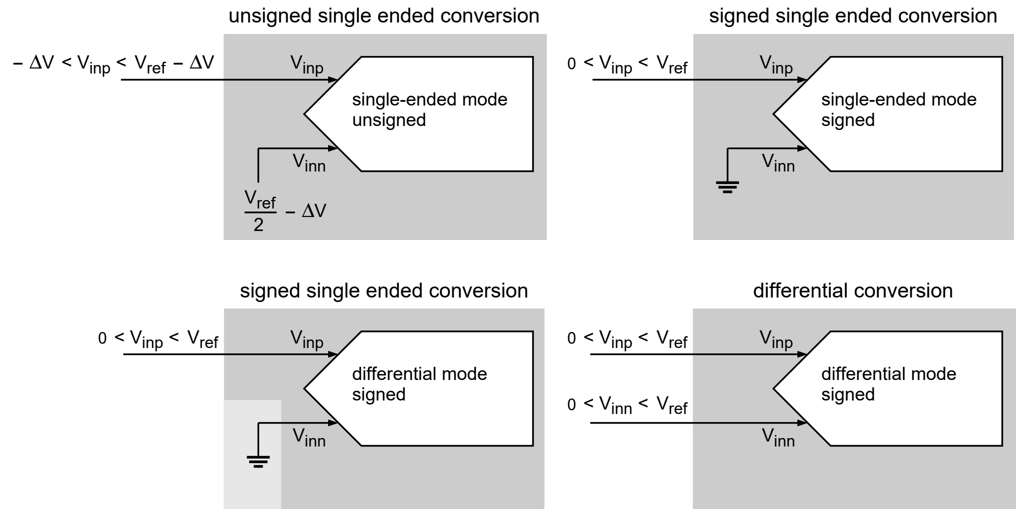
Figuur 20.7 : De registers voor het instellen van de kanalen `CHn`. Links staat register `CTRL` en rechts register `MUXCTRL`.

Ieder kanaal `CHn` of *channel* heeft twee registers `CTRL` en `MUXCTRL`, zie figuur 20.7, voor het instellen van onder andere de versterking, de ingangsmodus en de ingangsmultiplexers. De ADC kent vier verschillende ingangsmodi: de interne modus, de single-ended-modus en de differentiële modus met en zonder *gain*. De ingangsmodus wordt ingesteld met de twee `INPUTMODE`-bits, zie tabel 20.1, van register `CTRL` van kanaal `CHn`.

Tabel 20.1 : De ingangsmodi voor de channels van de ADC.

`INPUTMODE[1:0]`	**Groepsconfiguratie**	**Betekenis**
`00`	`ADC_CH_INPUTMODE_INTERNAL_gc`	Internal, no gain
`01`	`ADC_CH_INPUTMODE_SINGLEENDED_gc`	Single ended, no gain
`10`	`ADC_CH_INPUTMODE_DIFF_gc`	Differential, no gain
`11`	`ADC_CH_INPUTMODE_DIFFW_gc`	Differential, with gain

De betekenis van de selectiebits in de `CHn.MUXCTRL`-registers voor het instellen van de multiplexers hangt af van de gebruikte ingangsmodus. Naast de interne modus en de versterking zijn er drie basisconfiguraties te onderscheiden: *unsigned single-ended*, *signed single-ended* en *differential*.



Figuur 20.8 : Vier configuraties voor de ADC.

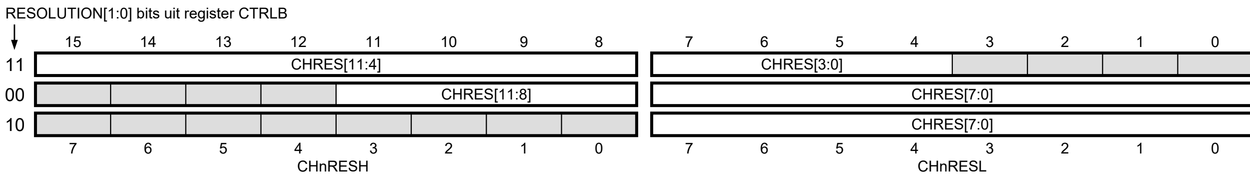
In figuur 20.8 zijn vier configuraties getekend. Bij *unsigned single-ended* is de ingangsmodus ingesteld op *single-ended* en de conversiemodus op *unsigned*. Bij *differential* is de ingangsmodus differentieel en is de conversiemodus altijd *signed*. Bij *signed single-ended* is de ADC is ingesteld op *single-ended* en de conversiemodus op *signed*. De negatieve ingang is dan automatisch verbonden met de interne `GND`. Omdat V_{INP} dan altijd groter is dan `GND`, worden er dan bij een 12-bits conversie effectief 11-bits gebruikt en ligt de uitkomst altijd tussen `0` en `2047`. In feite komt deze configuratie overeen met de differentiële modus, waarbij de negatieve ingang verbonden is met een interne of externe `GND`.

De positieve ingang V_{INP} is — mits er geen versterking is — bij ADCA verbonden met een intern signaal of met één van de pinnen van poort A of poort B. Met versterking kan dat alleen een pin van poort A zijn. De positieve ingang V_{INP} van ADCB kan alleen met een intern signaal of met één van de pinnen van poort B verbonden zijn. Met versterking kan dat alleen een pin van poort B zijn. Bij de differentiële modus zonder versterking en bij *signed single-ended* is de negatieve ingang V_{INN} verbonden met één van de interne signalen of met één van de pinnen 4 tot en met 7 van de betreffende poort. Bij de differentiële modus met versterking is de negatieve ingang verbonden met een intern signaal of met één van de pinnen `0` tot en met 3 van de betreffende poort.

Het uitgangsblok van de ADC

De Xmega is een 8-bits microcontroller. Voor de 12-bits waarde die de ADC berekent, zijn voor ieder kanaal twee registers nodig: een register `CHnRESH` voor de meest significante bits en een register `CHnRESL` voor de minst significante bits. De twee registers vormen samen een 16-bits register `CHnRES`. In figuur 20.9 zijn de registers van één van de vier kanalen getekend.

De berekende waarde kan op drie manieren in de twee registers komen te staan: de 12-bits waarde kan links of rechts uitgelijnd zijn en bij een 8-bits waarde is de uitkomst altijd rechts uitgelijnd. De twee `RESOLUTION`-bits uit register `CTRLB` van de ADC leggen de resolutie het aantal bits en de wijze van uitlijnen vast. Figuur 20.9 laat de drie manieren voor het weergeven van het conversieresultaat zien.



Figuur 20.9 : **De uitlijning van het resultaat van de conversie.** Een 12-bits resultaat kan rechts (`00`) en links (`11`) uitgelijnd in de registers `CHnRESL` en `CHnRESL` staan. Het 8-bits resultaat (`10`) is altijd rechts uitgelijnd.

De ADC is geschikt voor *signed* en *unsigned* conversies. Dit wordt ingesteld met de `CONVMODE`-bit uit register `CTRLB`. Niet gebruikte bits in de resultaatregisters worden bij een *signed* conversie automatisch 1 als de uitkomst negatief is. In alle andere gevallen worden de niet gebruikte bits `0`. Bij de rechts uitgelijnde 12-bits resolutie kan daarom de uitkomst direct aan een **`unsigned`** of **`signed`** worden toegekend:

```
uint16_t  x_unsigned;

x_unsigned = ADCA.CH0.RES;
```

```
int16_t  x_signed;

x_signed = ADCA.CH0.RES;
```

In sommige situaties is het praktisch om de hoge en lage byte apart uit te lezen, bijvoorbeeld als het resultaat direct via een UART of SPI verstuurd wordt. Bij de 8-bit modus staat de uitkomst in de lage byte en kan het resultaat direct uit de lage byte worden gehaald. In onderstaande voorbeelden wordt de uitkomst van de hoge en de lage byte in een array `a` gezet en wordt de lage byte van een 8-bits conversie direct aan `i8` toegekend:

```
uint8_t  a[2];

a[1] = ADCA.CH0.RESH;
a[0] = ADCA.CH0.RESL;
```

```
int8_t  i8;

i8 = ADCA.CH0.RESL;
```

Het resultaat van de conversie hangt af van de gebruikte ingangs- en conversiemodus. Voor de *unsigned single-ended*-conversie geldt:

$$\texttt{RES} \;=\; \frac{V_{\text{INP}} - (-\Delta V)}{V_{\text{REF}}} \cdot (\texttt{TOP} + 1) \tag{20.4}$$

waarbij `TOP` gelijk is aan 4095 of 255 bij een 12-bits of 8-bits conversie en ΔV ongeveer gelijk is 0,05 V_{ref}. De positieve ingang V_{INP} van de feitelijke ADC kan bij Xmega256a3u verbonden zijn met een intern signaal of met één van de pinnen van poort A of B. De negatieve ingang V_{INN} is in deze modus automatisch verbonden met $V_{\text{REF}}/2 - \Delta V$. Deze offsetspanning maakt het mogelijk om ook in deze modus nuldoorgangen te detecteren.

In figuur 20.6 zijn V_{INP} en V_{INN} de ingangen van de feitelijke ADC. In formule 20.5 is zowel V_{INP} als V_{INN} een fysieke pin of een intern signaal. Zonder versterking maakt dat niet uit, maar met versterking zijn dat strikt genomen niet de ingangen van de feitelijke ADC.

Voor alle andere modi geldt:

$$\texttt{RES} = \frac{V_{INP} - V_{INN}}{V_{REF}} \cdot \texttt{GAIN} \cdot (\texttt{TOP} + 1) \tag{20.5}$$

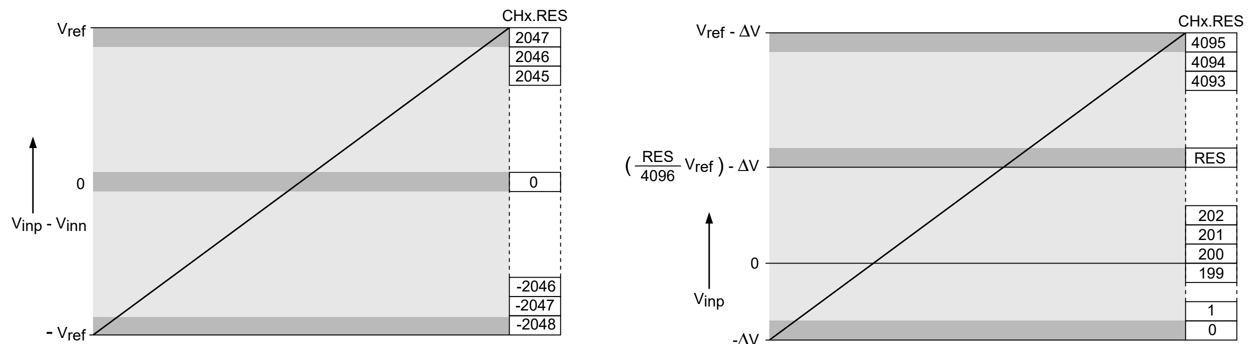
waarbij `TOP` gelijk is aan 2047 of 127 voor de 12-bits of 8-bits *signed*-conversies en gelijk is aan 4095 of 255 voor de 12-bits en 8-bits *unsigned differential*-modus. De `GAIN` kan een 1/2, 1, 2, 4, 8, 16, 32 of 64 zijn bij de *differential*-modus en is 1 bij alle andere modi.

De referentiespanning van de ADC

Een bandgap-referentie of *bandgap reference* is een schakeling, die een spanning afgeeft die onafhankelijk is van variaties in de voedingsspanning en van de temperatuur.

De ADC heeft een referentiespanning nodig. Deze spanning wordt intern bij de AD- en DA-conversies van de *stage* uit figuur 20.3 gebruikt. In principe definieert de referentiespanning het bereik van de ADC. De Xmega256a3u kent vijf referentiespanningen:

- een nauwkeurige spanning van 1,00 V afkomstig van een interne bandgap-referentie;
- een interne spanning die gelijk is aan `VCC`/1,6;
- een interne spanning die gelijk is aan `VCC`/2;
- een externe spanning die is aangesloten op A0 — de `AREF`-pin van poort A;
- een externe spanning die is aangesloten op B0 — de `AREF`-pin van poort B.



Figuur 20.10 : Het bereik van de ADC voor de 12-bits conversie. Links staat het bereik voor de *signed*-conversies. De ingangsspanning $V_{INP} - V_{INN}$ ligt tussen $-V_{ref}$ en V_{ref}. Bij *signed single-ended* is V_{INN} gelijk aan nul en is het resultaat altijd positief. Rechts staat het bereik voor de *unsigned single-ended*. De ingangsspanning V_{INP} ligt hier tussen $-\Delta V$ en $V_{ref} - \Delta V$

De externe referentiespanning moet liggen tussen 1 V en `AVCC`—0,6 V, waarbij `AVCC` de analoge voedingsspanning is, die maximaal 0,3 V af mag wijken van de voedingsspanning.

De referentiespanning bepaalt het bereik van de ADC en is anders bij de verschillende soorten conversies, zie ook figuur 20.10. Bij *signed*-conversies ligt de ingangsspanning $V_{INP} - V_{INN}$ tussen $-V_{ref}$ en V_{ref}. V_{INP} en V_{INN} moeten altijd allebei positief zijn. De ingangsspanning van de ADC is alleen negatief als V_{INN} groter is dan V_{INP}. De uitkomst loopt bij een 12-bits conversie van —2048 tot en met 2047.
Bij de *signed single-ended* conversie is V_{INN} gelijk aan nul en V_{INP} altijd groter dan V_{INN}. De ingangsspanning ligt dan altijd tussen 0 en V_{ref} en de uitkomst tussen 0 en 2047.

Bij *unsigned single-ended*-conversie ligt de ingangsspanning ligt V_{INP} tussen $-\Delta V$ en $V_{ref}-\Delta V$ en loopt de uitkomst bij een 12-bits conversie van 0 tot en met 4095.

Prescaling van de ADC-klok

De maximale klokfrequentie van een ADC is over het algemeen lager dan de frequentie van de systeemklok van de microcontroller. De conversie van een analoog signaal naar een digitaal gaat niet oneindig snel. Dat geldt voor alle typen ADC's. Bij successieve approximatie duurt het altijd enige tijd voordat de comparator een stabiel signaal geeft. Bij de pipelined-ADC heeft de *sample&hold* enige tijd nodig om stabiel te worden. Er is altijd enige tijd nodig voor de 2-bits AD- en DA-conversies en voor het aftrekken en versterken van de signalen. Op zich is het in de praktijk wel verstandig de klokfrequentie niet te laag te maken. De *sample&hold*-schakeling houdt de waarde gedurende een beperkte tijd vast.

De ADC's van de Xmega256a3u functioneren optimaal als de klokfrequentie van de ADC f_{adc} tussen 100 kHz en 2 MHz ligt. Bovendien mag f_{adc} niet hoger zijn dan een kwart van de systeemklok. Dus bij de standaardklok van 2 MHz moet f_{adc} tussen 100 kHz en 500kHz liggen.

Een veel gemaakte fout is het overnemen van code, die bestemd is voor een bepaalde kristalfrequentie en deze dan niet aanpast. De originele code is bijvoorbeeld geschreven voor 32 MHz en heeft een prescaling van 256. Als dan de standaard klok van 2 MHz gebruikt wordt, is de klokfrequentie van de ADC 7,8 kHz. Dat is veel te laag.

De prescaler van de ADC kan de systeemklok delen door 4, 8, 16, 32, 64, 128, 256 en 512. Tabel 20.2 geeft de prescaling van de ADC en geeft voor geschikte waarden bij 2 MHz en 32 MHz de klokfrequentie van de ADC.

Tabel 20.2 : De bits `PRESCALER[2..0]` van het `PRESCALER`-register. Gegeven zijn de bijbehorende prescaling en de klokfrequentie van de ADC voor een systeemklok van 2 en 32 MHz.

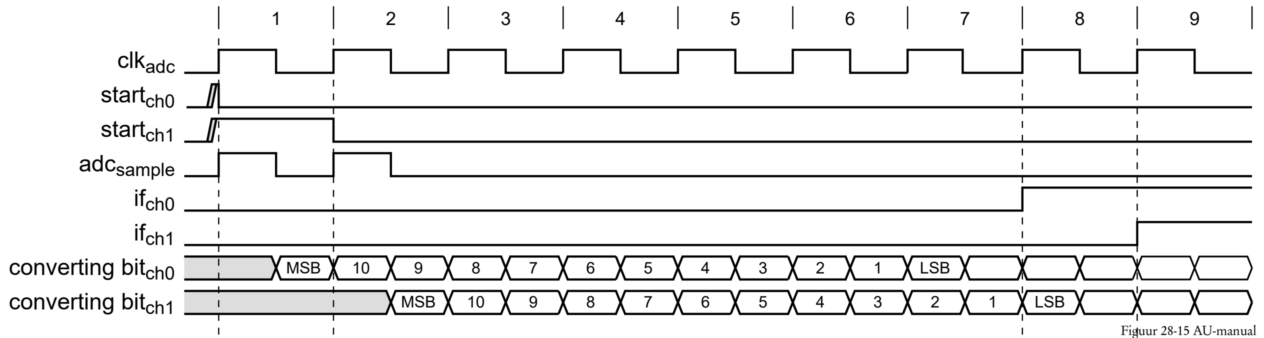
`PRESCALER[2..0]`	prescaling	systeemklok	
		2 MHz	32 MHz
000	4	500k	
001	8	250k	
010	16	125k	2M
011	32	†	1M
100	64		500k
101	128		250k
110	256		125k
111	512		†

† Hoewel de datasheet aangeeft dat de minimale frequentie 100 kHz is, melden diverse gebruikers op internet dat 62,5 kHz nauwkeuriger resultaten oplevert.

De conversietijd t_{conv} van de ADC hangt af van de kloksnelheid, de gewenste resolutie en of de versterking wordt gebruikt. In formule 20.6 kan de resolutie R 8 of 12 zijn.

$$t_{conv} = \begin{cases} \left(\frac{R}{2}+1\right) T_{adc_clock} & \text{zonder } \textit{gain} \\ \left(\frac{R}{2}+2\right) T_{adc_clock} & \text{met } \textit{gain} \end{cases} \tag{20.6}$$

Bij een klokfrequentie van 2 MHz en een prescaling van 16 is periodetijd T_{adc_clock} van de ADC 8 μs. Voor een 12-bits conversie zonder versterking is de conversietijd dan 56 μs.

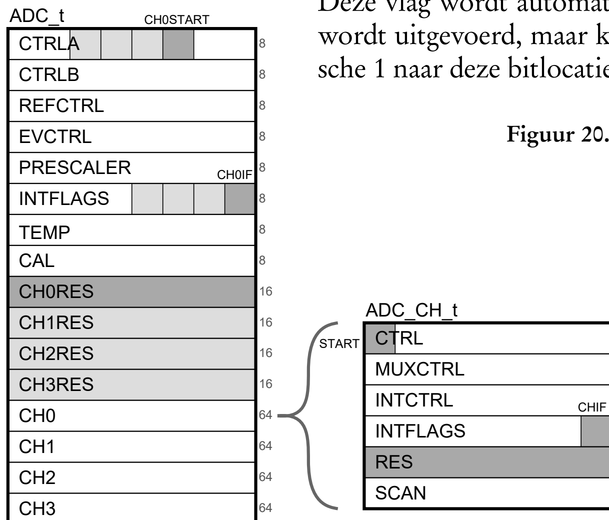


Figuur 20.11 : **Het tijdsgedrag van de ADC met twee ADC-kanalen.** De conversies voor channel `0` en 1 worden gelijktijdig gestart. Eerst wordt channel `0` gesampled en zeven klokslagen na de start is de conversie klaar en wordt de interrupt flag hoog. Channel 1 start één klokslag later en is ook één klokslag later klaar.

Bij de pipelined-ADC kan de gemiddelde verwerkingstijd sneller zijn, omdat er vier kanalen tegelijkertijd gebruikt kunnen worden. In figuur 20.11 staat een voorbeeld van het tijdsgedrag uit de AU-manual van twee aparte 12-bits conversies met een eigen kanaal.

20.3 De conversiemethoden

De ADC van de Xmega kan op verschillende manieren gestart worden: handmatig, automatisch via het eventmechanisme of automatisch met de *freerunning mode*. Na de conversie wordt de interruptvlag van het betreffende kanaal hoog. Deze vlag wordt automatisch laag gemaakt als de bijbehorende interruptfunctie wordt uitgevoerd, maar kan ook handmatig laag worden gemaakt door een logische 1 naar deze bitlocatie te schrijven.



Figuur 20.12 : **De datastructuren voor de ADC.** Links staat een grafische weergave van de datastructuur `ADC_t` van de ADC en de datastructuur `ADC_CH_t` van een kanaal. De getekende velden komen overeen met de registers van de ADC. Ongebruikte registers zijn niet getekend. De resultaatregisters zijn 16-bits. De velden `CHn` van `ADC_t` bevatten alle vier de datastructuur van een kanaal.
De resultaatregisters, startbits en de interruptvlaggen staan zowel in `ADC_t` als in `ADC_CH_t`. In de figuur zijn deze onderdelen licht grijs gekleurd voor de kanalen 1, 2 en 3 en donker grijs voor kanaal `0` . De startbit van kanaal `0` staat dus zowel in veld `CTRLA` van `ADC_t` als in veld `CTRL` van `ADC_CH_t`. In het eerste geval is dat `CH0START` en in het tweede geval is dat `START`.

De startbits, de interruptvlaggen en de resultaatregisters zijn dubbel uitgevoerd, althans ze zijn op twee verschillende manieren bereikbaar. De definities in `io.h` sluiten bij deze registerorganisatie aan en kent twee datastructuren: `ADC_t` en `ADC_CH_t`. In figuur 20.12 zijn deze datastructuren getekend. Datastructuur `ADC_t` bevat alle registers van de ADC, inclusief de datastructuren `ADC_CH_t` voor de vier kanalen.

Figuur 20.12 laat zien dat de startbits, de interruptvlaggen en de resultaatregisters van de kanalen steeds op twee plaatsen terug te vinden zijn. Daardoor kan een conversie op twee manieren beschreven worden: zonder ADC_CH_t met alleen de datastructuur ADC_t of met ADC_CH_t.

Handmatig converteren

Een conversie wordt handmatig gestart door de startbit hoog te maken, te wachten totdat de interruptvlag hoog is, daarna het resultaat te verwerken en tenslotte de interruptvlag te resetten. Onderstaand codefragment converteert op deze manier het analoge signaal dat op kanaal 0 is aangesloten en gebruikt bij de conversie alleen de datastructuur ADC_t:

```
ADCA.CTRLA |= ADC_CH0START_bm;                      // start ADC conversion
while ( !(ADCA.INTFLAGS & ADC_CH0IF_bm) ) ;         // wait until it’s ready
res = ADCA.CH0RES;                                  // do something
ADCA.INTFLAGS |= ADC_CH0IF_bm;                      // reset interrupt flag
```

Dezelfde conversie kan ook gedaan worden door de startbit, de interruptvlag en het resultaatregister uit ADC_CH_t te gebruiken:

```
ADCA.CH0.CTRL |= ADC_CH_START_bm;                   // start ADC conversion
while ( !(ADCA.CH0.INTFLAGS & ADC_CH_CHIF_bm) ) ;   // wait until it’s ready
res = ADCA.CH0.RES;                                 // do something
ADCA.CH0.INTFLAGS |= ADC_CH_CHIF_bm;                // reset interrupt flag
```

Omdat bij ADC_CH_t de interruptvlag de enige bit is van het register INTFLAGS dat gebruikt wordt, is bij het wachten en bij het resetten de bitmaskering niet noodzakelijk:

```
ADCA.CH0.CTRL |= ADC_CH_START_bm;                   // start ADC conversion
while (! ADCA.CH0.INTFLAGS) ;                       // wait until it’s ready
res = ADCA.CH0.RES;                                 // do something
ADCA.CH0.INTFLAGS = 0;                              // reset interrupt flag
```

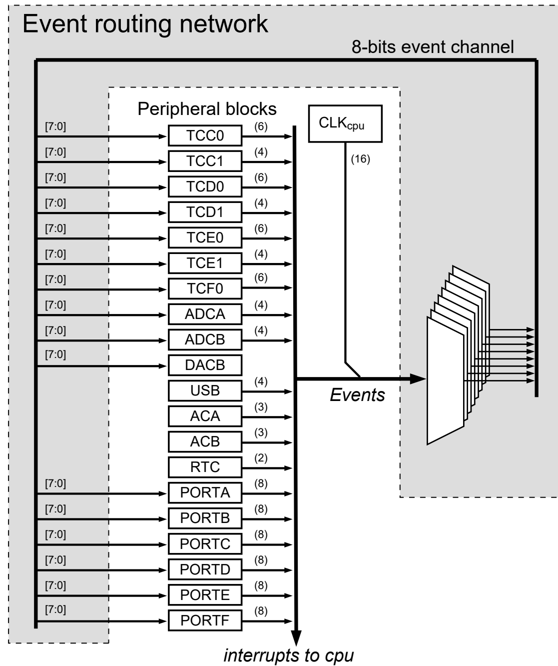
De voorbeeldprogramma’s in dit boek gebruiken overwegend de methode met ADC_CH_t inclusief bitmaskering.

Automatisch converteren

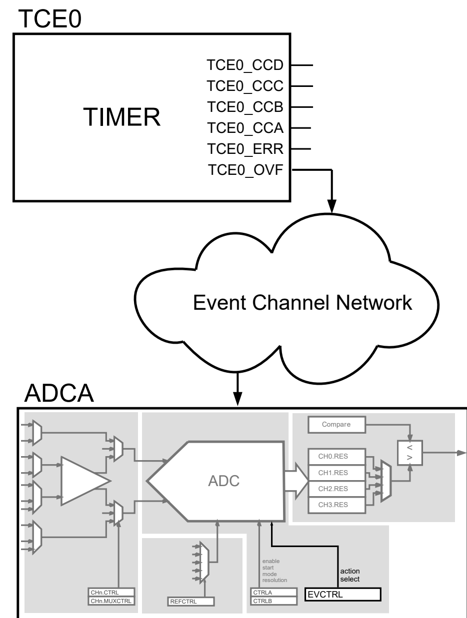
De ADC’s kunnen de analoge signalen ook automatisch converteren naar digitale waarden. Register CTRLB van de ADC bevat een bit FREERUN waardoor de ADC in de *freerunning mode* komt en de kanalen, die met het EVCTRL-register gedefinieerd zijn, voortdurend worden geconverteerd.

Het eventmechanisme van de Xmega kan ook worden gebruikt om — zonder tussenkomst van de CPU — automatisch AD-conversies uit te voeren. In figuur 20.13 staat het eventnetwerk van de Xmega. Via dit netwerk kunnen interrupts van een groot aantal perifere blokken rechtstreeks naar andere perifere blokken geleid worden.

In figuur 20.14 is de *timer overflow* van timer/counter 0 van poort E via het eventnetwerk verbonden met de trigger-ingang van de ADC. Dit wordt ingesteld met de CHnMUX-bits van het EVSYS-register van het eventsysteem. Iedere keer als de timer een overflow heeft, wordt er een AD-conversie uitgevoerd.



Figuur 20.13 : **Het eventnetwerk met de eventverbindingen tussen de perifere blokken**



Figuur 20.14 : **Een eventverbinding tussen timer `TCE0` en ADC `ADCA`.**

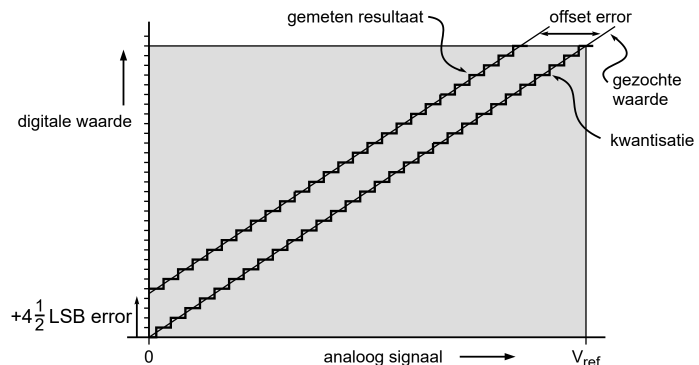
20.4 Fouten bij AD-conversie

Bij een ADC-meting kunnen verschillende soorten fouten gemaakt worden. Deze zijn te verdelen in vier groepen:

- *offset error* of offsetfout
- *gain error* of versterkingsfout
- *noise*
- niet lineariteiten

De GND van de *core* van de geïntegreerde schakeling wijkt vaak een paar mV af van de GND van IO-cellen, deze verschilt vaak weer een paar mV met de GND van het PCB.

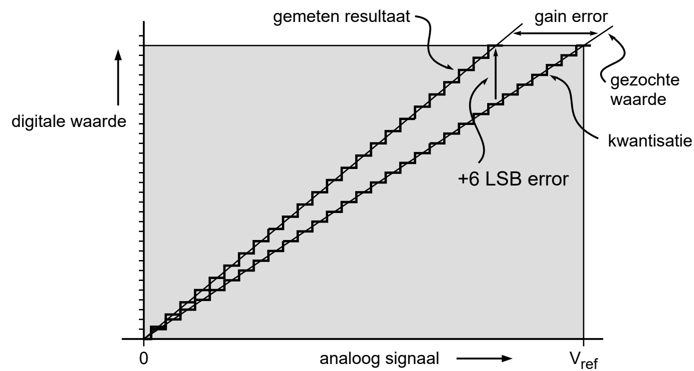
De offsetfout, zie figuur 20.15, is het verschil tussen de werkelijke en de gemeten waarde bij een ingangsspanning van nul volt. Deze fout kan positief of negatief zijn. De offsetfouten ontstaan meestal doordat de referentie van GND niet overal exact hetzelfde is.



Figuur 20.15 : **Een voorbeeld van een positieve offsetfout.**

De offset is op verschillende manieren te meten en kan dus worden gecompenseerd. Welke methode het beste is, hangt af van de manier waarop de ADC gebruikt wordt: *single-ended* of differentieel, één kanaal of een *sweep* met vier kanalen. De meest gebruikte methode bij de *single-ended* metingen is om tijdens de initialisatie de offset te bepalen door de ingang van de ADC tijdelijk met nul volt te verbinden en daarna de uitkomsten met deze offset te corrigeren. Bij de differentiële metingen kan de offset eenvoudig worden gevonden door de positieve en negatieve ingang te verwisselen.

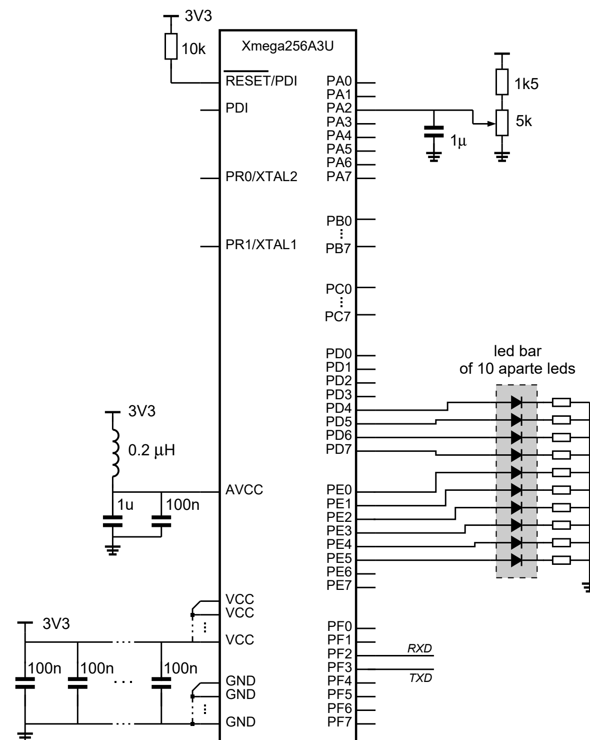
De *gain error* of versterkingsfout, zie figuur 20.16, is het verschil tussen de helling van de gevonden waarden en de werkelijke waarden. Deze fout kan eveneens positief of negatief zijn. De versterkingsfout kan worden gecompenseerd door vooraf de afwijking te meten en de gevonden waarden daarmee te corrigeren.



Figuur 20.16 : Een voorbeeld van een versterkingsfout.

Een onnauwkeurig gedefinieerde referentiespanning levert een versterkingsfout op. Daarom is het belangrijk om bij de AD-conversie een nauwkeurige, stabiele referentie te gebruiken. Als de referentiespanning VCC/1,6 V of VCC/2 V gebruikt wordt, moet de voedingsspanning stabiel zijn.

Naast de ruis of *noise* die de ADC zelf genereert, zijn er veel andere ruisbronnen, zoals de voedingsspanning, EMI-bronnen, overspraak en het snelle schakelen van de digitale componenten van de microcontroller. Een goede ontkoppeling van de ADC-ingangen is noodzakelijk om de ruis te onderdrukken. Bij een *single-ended* meting is een capaciteit van 1 μF tussen de analoge ingang en GND aan te bevelen. Bij een differentiële meting moet de capaciteit tussen de positieve en de negatieve ingang worden aangebracht. Bij het meten van snel wisselende signalen zal een capaciteit van 1 μF te groot zijn. De waarde van de ontkoppelcondensator moet zo hoog mogelijk zijn zonder dat de stijg- en daaltijd beïnvloed wordt.



Figuur 20.17 : Het schema voor het meten van een analoog ingangssignaal met ADCA. Het analoge signaal is aangesloten op pin 2 van poort A en ontkoppelt met een condensator. De referentie is intern aangesloten op `VCC`/1,6. De uitkomst van de conversie wordt op een 10-bits ledbar afgebeeld, waarmee de analoge waarde van het ingangssignaal gevisualiseerd wordt.

Volgens tabel 36-104 uit de datasheet moet de referentiespanning V_{REF} liggen tussen 1 V en AVCC − 0,6 V. Omdat AVCC hier gelijk is aan 3,3 V, moet V_{REF} kleiner zijn dan 2,7 V.

De ingangsspanning van de ADC moet liggen tussen −0,1 V en AVCC + 0,1 V

Bij een referentiespanning van $AVCC/1{,}6$ ligt voor een AVCC van 3,3 het bereik van de ingang tussen 0 en ruim 2 V.

20.5 Toepassing: handmatige *unsigned single-ended* conversie

De toepassingen in deze en volgende paragrafen tonen diverse voorbeelden van AD-conversies. Het voorbeeld van deze paragraaf gebruikt de *unsigned single-ended* configuratie en het basisalgoritme voor de handmatige conversie — of *single conversion* — van pagina 334. Het resultaat van de conversie wordt afgebeeld op een 10-bits ledbar met de methode die besproken is in paragraaf 17.1.

Het bijbehorende schema staat in figuur 20.17. Het analoge signaal is verbonden met pin 2 van poort A. In dit geval is het een regelbare spanning. Met een weerstand van 1k5 en een regelbare weerstand van 5k ligt de analoge spanning tussen 0 en 2,54 V. Om de stabiliteit te verbeteren is de ingang ontkoppeld met een capaciteit van 1μF.

Een ferrietkraal of *bead* is een spoel met een ferriet kern en reageert snel op hoogfrequente signalen. Een ferrietkraal heeft betere filtereigenschappen dan een gewone spoel. Meestal wordt niet de zelfinductie opgegeven maar de impedantie bij een bepaalde frequentie, bijvoorbeeld 50 Ω bij 20 MHz of 300 Ω bij 100 MHz.

De analoge voedingsspanning is ontkoppeld met een condensator van 1 μF en een condensator van 100 nF. De ferrietkraal met een zelfinductie van 0,5μH reduceert de ruis van de analoge voedingsspanning.
De referentie van de ADC is softwarematig aangesloten met de interne referentie van `VCC`/1,6.

De 10-bits ledbar is — net als in figuur 17.3 — aangesloten op pin 4 tot en met 7 van poort D en pin 0 tot en met 5 van poort E. De afbeelding van een analoog signaal op een 10-bits ledbar is geen nauwkeurige meting. Voor nauwkeurig onderzoek kan de waarde ook via de USART0 van poort F naar een pc worden gestuurd.

Code 20.1 : Handmatig converteren in de unsigned single-ended-modus.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define NUM_LEDS    10
7  #define MAX_VALUE   4095
8  #define RES_OFFSET  192
9
10 void init_ledbar(void);
11 void display_level(uint8_t level);
12
13 void init_adc(void)
14 {
15   PORTA.DIRCLR     = PIN2_bm;                          // configure PA2 as input for ADCA
16   ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc;            // PA2 to channel 0
17   ADCA.CH0.CTRL    = ADC_CH_INPUTMODE_SINGLEENDED_gc;  // channel 0 single-ended
18   ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;             // internal VCC/1.6 reference
19   ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc;          // 12 bits conversion, unsigned, no freerun
20   ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;           // 2MHz/16 = 125kHz
21   ADCA.CTRLA       = ADC_ENABLE_bm;                    // enable adc
22 }
23
24 uint16_t read_adc(void)
25 {
26   uint16_t res;
27
28   ADCA.CH0.CTRL |= ADC_CH_START_bm;                    // start ADC conversion
29   while ( !(ADCA.CH0.INTFLAGS & ADC_CH_CHIF_bm) ) ;    // wait until it's ready
30   res = ADCA.CH0.RES;
31   ADCA.CH0.INTFLAGS |= ADC_CH_CHIF_bm;                 // reset interrupt flag
32
33   return res;                                          // return measured value
34 }
35
36 int main(void)
37 {
38   uint16_t res;
39   uint8_t  level;
40
41   init_ledbar();
42   init_adc();
43
44   while (1) {
45     res = read_adc();
46     if (res < RES_OFFSET) {                            // res - RES_OFFSET must be >= 0
47       level = 0;
48     } else {
49       level = (res - RES_OFFSET) * ((NUM_LEDS) + 1)  / (MAX_VALUE - RES_OFFSET + 1);
50     }
51     display_level(level);
52     _delay_ms(200);
53   }
54 }
```

Het bijbehorende programma staat in code 20.1. De functie `display_level` staat in code 17.4 en beeldt het resultaat af op de ledbar. De functie `init_ledbar` bevat regel 30 en regel 31 uit code 17.1 en definieert de aansluitingen voor de ledbar als uitgang.

De functie `read_adc` voert de handmatige conversie van pagina 334 uit en geeft het gemeten resultaat terug. De functie `init_adc` initialiseert de ADC. Deze functie definieert pin A2 als ingang, verbindt deze met kanaal `0` en maakt de omzetting voor dit kanaal single-ended. Verder verbindt deze functie de referentie van de ADC met `VCC`/1,6 en stelt het de ADC in op de 12-bits modus en een prescaling van 16. Tenslotte zet de functie de ADC aan.

Sommige eigenschappen van ADC worden door `init_adc` impliciet ingesteld. Zo wordt bij de toekenning op regel 19 de `CONVMODE`-bit van register `CTRLB` ook `0` gemaakt, zodat de conversies *unsigned* zijn. Om een zelfde reden wordt de freerunningmodus niet gebruikt omdat `FREERUN` eveneens automatisch laag wordt. In `io.h` staat alleen een macro voor `ADC_FREERUN_bm`, die freerunningmodus instelt. Er is geen macro die de freerunningmodus expliciet niet instelt. Dit is te ondervangen met de logische inverse, `!`. De uitdrukking `!ADC_FREERUN_bm` is altijd nul en laat bij de onderstaande toewijzingen de betreffende bit ongemoeid.

```
ADCA.CTRLB  =  ADC_RESOLUTION_12BIT_gc;         // 12 bits conversion
```

```
ADCA.CTRLB  =  ADC_RESOLUTION_12BIT_gc;         // 12 bits conversion
                                                // unsigned,
                                                // no free run
```

```
ADCA.CTRLB  =  ADC_RESOLUTION_12BIT_gc |        // 12 bits conversion
              !ADC_CONMODE_bm          |        // unsigned
              !ADC_FREERUN_bm;                  // no free run
```

```
ADCA.CTRLB  =  ADC_RESOLUTION_12BIT_gc;         // 12 bits conversion
ADCA.CTRLB |= !ADC_CONMODE_bm;                  // unsigned
ADCA.CTRLB |= !ADC_FREERUN_bm;                  // no free run
```

De naam van het bitmasker is `ADC_CONMODE_bm`, terwijl de AU-manual over de `CONVMODE`-bit spreekt.

De bovenstaande vier methoden van toewijzen doen exact hetzelfde. Bij de laatste twee is expliciet aangegeven dat de `CONMODE`- en de `FREERUN`-bit laag zijn. Het commentaar bij de tweede toewijzing vermeldt dat de conversie *unsigned* is en dat de *free run mode* niet gebruikt wordt. Bij de eerste toewijzing is alleen vermeld dat de conversie 12-bits is.

Het expliciet vermelden van de status van alle bits of het toevoegen van commentaar over de status van een impliciete instelling voorkomt misverstanden. Dit wordt echter meestal niet gedaan omdat er eigenlijk teveel instellingen zijn die dan beschreven moeten worden. Register `CTRLB` bevat nog een bit waarmee de ingangsimpedantie aangepast kan worden en een 2-bits bitgroep, die de stroom door de ADC beperkt. Daarnaast heeft de ADC veel meer registers dan de zes die door `init_adc` gebruikt worden, zoals ook in figuur 20.12 te zien is. In de code zijn alleen de belangrijkste instellingen in het commentaar genoemd.

Het hoofdprogramma initialiseert de ledbar en de ADC en meet daarna voortdurend met de functie `read_adc` de waarde `res` van het analoge signaal, zet daarbij deze waarde om naar een niveau `level`, beeldt met de functie `display_level` dit niveau op de ledbar af en wacht 200 ms.

De offset bij de *unsigned single-ended* configuratie is ongeveer $0{,}05V_{\text{ref}}$. Omdat in dit geval de gemeten spanning niet relevant is, is de offset in het resultaat `RES_OFFSET` vooraf bepaald door de ingang van de ADC met `GND` te verbinden. Het resultaat is dan 192 in plaats van 0, zodat de offset 192 is. Bij de berekening van het level wordt deze offset van het resultaat `res` en van de maximale waarde `MAX_VALUE` afgetrokken. De test op regel 46 is nodig voor het geval het gecorrigeerde resultaat negatief is. In dat geval wordt `level` gelijk aan nul gemaakt.

Code 20.1 met uitvoer via een UART in plaats van een ledbar

Uit de gemeten waarde kan de bijbehorende spanning worden berekend en via een UART naar een pc worden verzonden. Uit formule 20.4 volgt dat:

$$V_{\text{INP}} = \frac{\texttt{RES}}{\texttt{TOP}+1} V_{\text{REF}} - \Delta V \tag{20.7}$$

In dit voorbeeld is de referentiespanning V_{REF} gelijk aan $V_{\text{CC}}/1{,}6$ en is de offsetspanning ΔV gelijk aan $0{,}05V_{\text{REF}}$. De waarden van deze spanningen zijn als definities op regel 8 tot en met 10 aan code 20.2 toegevoegd.

In code 20.2 wordt op regel 26 met deze definities en met formule 20.7 de spanning `vinp` van het analoge ingangssignaal berekend en het resultaat daarna met `printf` via de USART0 van poort F naar een pc doorgestuurd.

Code 20.2 : Handmatig converteren in de unsigned single-ended-modus met uitvoer via UART.

```
 1 #define F_CPU 2000000UL
 2 #include <avr/io.h>
 3 #include <avr/interrupt.h>
 4 #include <util/delay.h>
 5 #include "stream.h"
 6
 7 #define MAX_VALUE   4095
 8 #define VCC         3.30
 9 #define VREF        (((double) VCC) / 1.6)  // is 2.06125
10 #define VOFFSET     ((0.05)*(VREF))         // is 0.103125
11
12 void init_adc(void);                        // is identical with function init_adc() from code 20.1
13 uint16_t read_adc(void);                    // is identical with function read_adc() from code 20.1
14
15 int main(void)
16 {
17   uint16_t res;
18   double   vinp;
19
20   init_stream(F_CPU);
21   init_adc();
22   sei();
23
24   while (1) {
25     res  = read_adc();
26     vinp  = ((double) res) * VREF / (MAX_VALUE + 1) - VOFFSET;          // formula 20.7
27     printf("res: %4d  voltage: %5.3f V\n", res, vinp);
28     _delay_ms(200);
29   }
30 }
```

De functie `init_stream` uit paragraaf 19.12 initialiseert op regel 20 de UART. Deze functie zorgt ervoor dat de `stdout` van de `printf` gekoppeld is aan USART0 van poort F. Om gebroken getallen af te drukken moet de bibliotheek `printf_flt` toegevoegd worden, zoals al eerder in paragraaf 18.8 bij het afdrukken van gebroken getallen op een LCD is uitgelegd.

```
COM1 - PuTTY
res: 2492 voltage: 1.152 V
res: 2491 voltage: 1.151 V
res: 2492 voltage: 1.152 V
res: 2493 voltage: 1.152 V
res: 2494 voltage: 1.153 V
res: 2493 voltage: 1.152 V
```

Figuur 20.18 : De uitvoer op de pc voor een analoge ingangsspanning van 1,154 V.

In figuur 20.18 staat een aantal uitkomsten voor een ingangssignaal van 1,154 V. De met de ADC gemeten spanning wijkt af van de werkelijke waarde. Omdat de voedingsspanning en de deelfactor niet exact 3,30 V en 1,6 zijn, zal de werkelijke referentiespanning nooit gelijk zijn aan $2,0625$ V. Bovendien is de werkelijke offsetspanning niet gelijk aan de berekende waarde omdat de referentiespanning en de factor 0,05 niet exact zijn.
De volgende paragraaf toont een methode om de offsetspanning bij *signed single-ended* te compenseren, die ook bruikbaar is bij *unsigned single-ended*.

20.6 Toepassing: handmatige *signed single-ended* conversie

In code 20.3 staat een programma dat de *signed single-ended* gebruikt. De configuratie lijkt sterk op de configuratie, die gebruikt is in code 20.1 voor *unsigned single-ended*. Het voordeel is dat er bij *signed* geen offset ΔV nodig is. Het nadeel is dat er effectief maar elf bits gebruikt worden. De functies `init_adc` en `read_adc` uit code 20.3 zijn geschikt voor de *signed single-ended*. De functie `init_adc` is ten opzichte van die uit code 20.1 op een aantal punten gewijzigd: de negatieve ingang van de ADC is nu verbonden met `GND`, de inputmodus is *differential* en de conversiemodus is *signed*. Omdat de uitkomst van de conversie *signed* is, is de retourwaarde van `read_adc` ook *signed*.

De ADC kent in de differentiële modus verschillende interne aansluitingen voor `GND`, zie tabel 20.3. De zogenoemde *pad* `GND` is identiek aan of ligt vlakbij de externe `GND`. De interne `GND` wordt gebruikt bij metingen van interne signalen in de *signed single-ended*-modus. Het voorbeeldprogramma uit code 20.3 gebruikt `ADC_CH_MUXNEG_GND_MODE3_gc` als `GND`.

Tabel 20.3 : De interne GND-aansluitingen voor de ADC.

MUXNEG[2:0]	groepsconfiguratie	type	gebruikt bij inputmodus
101	ADC_CH_MUXNEG_GND_MODE3_gc	pad GND	ADC_CH_INPUTMODE_DIFF_gc
111	ADC_CH_MUXNEG_INTGND_MODE3_gc	interne GND	ADC_CH_INPUTMODE_DIFF_gc
100	ADC_CH_MUXNEG_GND_MODE4_gc	pad GND	ADC_CH_INPUTMODE_DIFFWGAIN_gc
111	ADC_CH_MUXNEG_INTGND_MODE4_gc	interne GND	ADC_CH_INPUTMODE_DIFFWGAIN_gc

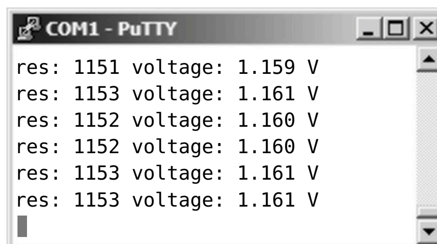
Code 20.3 : **Handmatige conversie met de signed single-ended-modus.** Deze code gebruikt de configuratie uit figuur 20.8 met de differentiële ingangsmodus, waarbij de negatieve ingang verbonden is met de interne GND.

```
1  #define F_CPU 2000000UL
2  #include <avr/io.h>
3  #include <avr/interrupt.h>
4  #include <util/delay.h>
5  #include "stream.h"
6
7  #define MAX_VALUE   2047                                  // only 11 bits are used
8  #define VCC         3.30
9  #define VREF        (((double) VCC) / 1.6)                // is 2.06125
10
11 void init_adc(void)
12 {
13   PORTA.DIRCLR     = PIN2_bm|PIN3_bm;                     // PA3 can be used for offset
14   ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc |              // PA2 to + channel 0
15                      ADC_CH_MUXNEG_GND_MODE3_gc;          // GND to - channel 0
16   ADCA.CH0.CTRL    = ADC_CH_INPUTMODE_DIFF_gc;            // channel 0 differential
17   ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;
18   ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc |
19                      ADC_CONMODE_bm;                      // signed conversion
20   ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;
21   ADCA.CTRLA       = ADC_ENABLE_bm;
22 }
23
24 int16_t read_adc(void)                                    // return a signed
25 {
26   int16_t res;                                            // is also signed
27
28   ADCA.CH0.CTRL |= ADC_CH_START_bm;
29   while ( !(ADCA.CH0.INTFLAGS & ADC_CH_CHIF_bm) ) ;
30   res = ADCA.CH0.RES;
31   ADCA.CH0.INTFLAGS |= ADC_CH_CHIF_bm;
32
33   return res;
34 }
35
36 int main(void)
37 {
38   int16_t res;                                            // is also signed
39   double   vinp;
40
41   init_stream(F_CPU);
42   init_adc();
43   sei();
44
45   while (1) {
46     res = read_adc();
47     vinp = (double) res * VREF / (MAX_VALUE + 1);         // no offset
48     printf("res: %4d  spanning: %5.3f V\n",  res, vinp);
49     _delay_ms(200);
50   }
51 }
```

De gemeten spanning volgt uit formule 20.5. De negatieve ingang is met `GND` verbonden. V_{INN} is dus gelijk aan `0` V. Omdat de ingangsversterker niet gebruikt wordt, is de `GAIN` gelijk aan 1. Voor V_{INP} geldt zodoende:

$$V_{\mathrm{INP}} = \frac{\mathtt{RES}}{\mathtt{TOP}+1} V_{\mathrm{REF}} \tag{20.8}$$

De waarde van `TOP`, in code 20.3 is dat macro `MAX_VALUE`, is in dit geval 2047.



Figuur 20.19 : De gemeten waarde voor een analoge ingangsspanning van 1,154 V bij de *signed single-ended* modus.

Figuur 20.19 toont de gemeten waarde van het analoge ingangssignaal. De met formule 20.8 berekende waarde wijkt ongeveer 0,6% af. Interessant is dat met de ADC gemeten waarde `res` overeenkomt met de gezochte waarde 1,154. Dit komt doordat de factor 1,6 niet exact 1,6 is, maar door Atmel bewust iets groter is gekozen, zodat voor een voedingsspanning van 3,3 V de referentiespanning ongeveer 2,048 V is. De gemeten ADC-waarde bij *signed single-ended* is daardoor ongeveer gelijk aan de spanning in mV.

Offsetcompensatie

Een kleine offset van een paar mV kan onder andere ontstaan doordat de potentiaal van `GND` in de microcontroller niet overal exact hetzelfde is.

Code 20.4 : De functie `offset_adc`.

```
36  int16_t offset_adc(int n)
37  {
38    uint8_t save_muxctrl;
39    int16_t offset = 0;
40
41    save_muxctrl = ADCA.CH0.MUXCTRL;                    // save MUXCTRL
42    ADCA.CH0.MUXCTRL = (ADCA.CH0.MUXCTRL & ~ADC_CH_MUXPOS_gm) |
43                       (ADC_CH_MUXPOS_PIN3_gc & ADC_CH_MUXPOS_gm);
44    for(int i=0; i<n; i++) {                            // measure offset
45      offset += read_adc();
46    }
47    ADCA.CH0.MUXCTRL = save_muxctrl;                    // restore MUXCTRL
48
49    return offset/n;
50  }
```

De uitkomst is te corrigeren door de offset vooraf bij de initialisatie te meten, en deze van het resultaat af te trekken. Het meten van de offset kan door de positieve ingang van de ADC tijdelijk via een andere pin op een externe `GND` aan te sluiten.

De naam *signed single-ended* is in dit geval ongelukkig gekozen. De gebruikte ingangsmodus is *differential* en bij het bepalen van de offset worden er twee ingangen gebruikt.

Deze implementatie met de differentiële modus is flexibeler dan de gewone *signed single ended*.

Functie `offset_adc` uit code 20.4 bewaart de huidige status van register `MUXCTRL`, verbindt de positieve ingang met pin 3 van poort A, meet de ingangswaarde en zet de oude status van `MUCXCTRL` weer terug.
De functie heeft een ingangsparameter `n`. Bij één enkele meting is de kans op een toevallige afwijking groot. Dat is de reden dat de functie de offset `n` keer bepaalt en daarna het gemiddelde van deze metingen teruggeeft.

Het hoofdprogramma uit code 20.5 bepaalt, direct nadat de ADC geïnitialiseerd is, met de functie `offset_adc` de offset en corrigeert de gemeten waarde met deze offset. Het resultaat wordt afgebeeld op de ledbar. Bij de berekening van het niveau wordt de offset ook van `MAX_VALUE` afgetrokken.

Code 20.5 : Handmatig converteren in de signed single-ended-modus met offsetcorrectie.

```
52  int main(void)
53  {
54    int16_t res;               // is here signed (int16_t)
55    double  vinp;
56
57    init_stream(F_CPU);
58    init_adc();
59    sei();
60
61    int16_t offset = offset_adc(16);
62
63    while (1) {
64      res = read_adc() - offset;
65      vinp = ((double) res) * VREF / (MAX_VALUE + 1);
66      printf("res: %4d  voltage: %5.3f V %d\n",  res, vinp, offset);
67      _delay_ms(200);
68    }
69  }
```

20.7 Toepassing: handmatige conversie met differentiële modus

Hoewel in de vorige paragraaf de ingangsmodus van het kanaal differentieel was, werd bij de meting feitelijk één ingang gebruikt. De toepassing in deze paragraaf gebruikt voor de feitelijke meting twee ingangen en voert de meting in ieder geval altijd twee keer uit. De eerste keer is de pin 2 verbonden met de positieve ingang en pin 3 met de negatieve ingang. Bij de tweede meting is pin 3 juist met de positieve en pin 2 met de negatieve ingang verbonden. Het resultaat van de eerste meting wordt van de tweede meting afgetrokken. Het effect is dat het resultaat het dubbele is en dat de offset wegvalt.

In code 20.6 is deze methode gebruikt om het spanningsverschil tussen pin 2 en pin 3 te meten. De dubbele meting wordt vier keer uitgevoerd. Het gezochte resultaat is dan acht keer zo klein.
De reden om meer samples te nemen voor één meting is dat de toevallige variaties gemiddeld worden en de uitkomst nauwkeuriger en stabieler is. Het nadeel is dat er voor de totale meting meer tijd nodig is en dat dit bij snel wisselende ingangssignalen niet bruikbaar is. In dat geval kan het verstandig zijn de prescaling van ADC

Code 20.6 : **Handmatig converteren met de differentiële ingangsmodus.**

```
36  int main(void)
37  {
38    uint8_t  level;
39    int16_t  res;
40
41    init_stream(F_CPU);   // see also code  19.16
42    init_ledbar();
43    init_adc();
44
45    PMIC.CTRL |= PMIC_LOLVLEN_bm;
46    sei();
47
48    while(1) {
49      res = 0;
50      for (int i=0; i<4; i++) {
51        ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc | ADC_CH_MUXNEG_PIN3_gc;
52        res += read_adc();
53        ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN3_gc | ADC_CH_MUXNEG_PIN2_gc;
54        res -= read_adc();
55      }
56      res = res/8;
57
58      level =  abs(res) * ((NUM_LEDS) + 1) / (MAX_VALUE + 1);
59      display_level(level);
60      printf("res: %4d mV\n",  res);
61
62      _delay_ms(200);
63    }
64  }
```

zo klein mogelijk te maken. Bij vier iteraties wordt er acht keer een ADC-meting gedaan. Omdat het een *signed* conversie is, is de maximale waarde van één losse meting 2047 en heeft `res` na de **`for`**-lus maximaal 16376. Dit past royaal binnen het bereik van een 16-bits integer.

Bij 3,30 V is de referentiespanning theoretisch gelijk aan 2063,5 mV. In de praktijk blijkt deze waarde veel dichter bij 2048 mV te liggen.

Er zijn ook zeer nauwkeurige spanningsreferenties van 2,048 en 4,096 V verkrijgbaar, die als externe referentie gebruikt kunnen worden.

Mits de voedingsspanning gelijk is aan 3,30 V en de referentiespanning ingesteld is op `VCC`/1,6 is het interessant te zien dat de uitkomst van de ADC-conversie bij deze methode ongeveer gelijk is aan de werkelijke ingangsspanning in millivolts. Daarom wordt op regel 60 de uitkomst onbewerkt als mV naar de pc gestuurd.

20.8 Toepassing: differentiële conversie met behulp van een interrupt

De functies `read_adc` uit code 20.1 en code 20.3 gebruiken beide de interruptvlag `ADC_CH_CHIF_bm` uit register `INTFLAGS` om te bepalen of de conversie klaar is. De interruptvlag kan ook automatisch een interruptroutine starten die het gemeten resultaat verwerkt.

In code 20.7 is op regel 13 voor kanaal `0` het interruptmechanisme ingesteld op het lage niveau. Het hoofdprogramma zet op regel 35 het interruptmechanisme aan en start op regel 38 de ADC. Nadat de AD-conversie klaar is, wordt de interruptfunctie van regel 23 uitgevoerd. Deze kent het resultaat toe aan de globale

Code 20.7 : Differentiële conversie met interrupt.

```
#define F_CPU  2000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "stream.h"

void init_adc(void)
{
  PORTA.DIRCLR     = PIN2_bm|PIN3_bm;
  ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc |
                     ADC_CH_MUXNEG_PIN3_gc;
  ADCA.CH0.CTRL    = ADC_CH_INPUTMODE_DIFF_gc;
  ADCA.CH0.INTCTRL = ADC_CH_INTLVL_LO_gc;     // low level interrupt CH0
  ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;
  ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc |
                     ADC_CONMODE_bm;
  ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;
  ADCA.CTRLA       = ADC_ENABLE_bm;
}

volatile int16_t res = 0;

ISR(ADCA_CH0_vect)
{
  res = ADCA.CH0.RES;                         // save result

  ADCA.CH0.CTRL |= ADC_CH_START_bm;           // starts conversion again
}

int main(void)
{
  init_stream(F_CPU);
  init_adc();

  PMIC.CTRL |= PMIC_LOLVLEN_bm;
  sei();

  ADCA.CH0.CTRL |= ADC_CH_START_bm;           // starts first conversion
  while(1) {
    printf("Voltage: %d mV\n", res);
    _delay_ms(200);
  }
}
```

variable `res` en start de AD-conversie opnieuw. De ADC blijft zo voortdurend nieuwe waarde toekennen aan `res`. Het hoofdprogramma stuurt de waarde van `res` door naar de UART.

De variabele `res` moet **`volatile`** zijn, omdat de compiler deze variabele anders wegoptimaliseert. In het hoofdprogramma verandert `res` immers niet. De compiler zal bij de optimalisatie daarom `res` behandelen als een constante. Het sleutelwoord **`volatile`** vertelt de compiler dat `res` toch kan veranderen.

Deze toepassing corrigeert niet voor de offset. De gemeten waarde zal dus enigszins afwijken van de werkelijke waarde. Een belangrijker nadeel van de gebruikte methode is dat de microcontroller heel veel metingen uitvoert. Iedere keer als er een AD-meting gedaan is, wordt er automatisch de volgende meting gestart. Vooral bij een kleine prescaling is de meettijd kort. Bij een prescaling van 4 zijn er voor een 12-bits conversie slechts 28 klokslagen nodig. Het gevolg is dat de microcontroller bijna voortdurend bezig is de interruptfunctie uit te voeren. Daarom zal het meestal beter zijn de metingen volledig automatisch uit te voeren met behulp van de freerunningmodus of door een timer te gebruiken.

20.9 Toepassing: differentiële conversie op vaste tijdstippen

Met een timer en het *event routing network* van de Xmega kan de AD-conversie automatisch op vaste tijdstippen worden uitgevoerd. Het voorbeeld van figuur 20.14 is geïmplementeerd in code 20.8. De overflow van timer `0` van poort E triggert via het eventmechanisme de ADC. Als de meting klaar is, wordt automatisch de interruptfunctie van de ADC uitgevoerd en krijgt variabele `res` een nieuwe waarde. Het voordeel van deze opzet is dat de ADC op vaste tijdsstippen de waarde van het analoge ingangssignaal meet en de interruptfunctie niet voortdurend actief is.

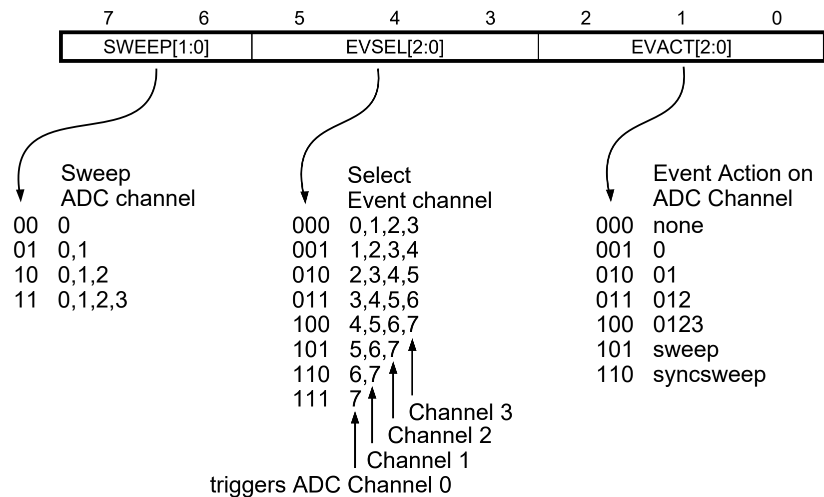
In code 20.8 wordt op regel 19 het eventcontrolregister `EVCTRL` ingesteld. Dit register, zie figuur 20.20, heeft drie bitgroepen: `SWEEP`, `EVSEL` en `EVACT`. De `EVSEL`-bits selecteren de eventkanalen waarmee de kanalen van de ADC verbonden zijn. In dit voorbeeld zijn de eventkanalen `0`, 1, 2 en 3 verbonden met respectievelijk de ADC-kanalen `0`, 1, 2 en 3. De `EVACT`-bits geven aan hoeveel en welke van de geselecteerde eventkanalen er worden gebruikt. Bovendien kan hiermee worden ingesteld dat alle ADC-kanalen, die door de `SWEEP`-bits zijn benoemd, direct achter elkaar worden gemeten. De `SWEEP`-bits geven aan welke ADC-kanalen bij de zogenoemde `sweep` en bij de freerunningmodus automatische worden gemeten.

Op regel 22 is kanaal `0` van het eventsysteem verbonden met de overflow van timer `0` van poort E. Het effect van deze en vorige regel is dat de overflow van de timer een gebeurtenis genereert op eventkanaal `0` en dat de ADC hierop een meting uitvoert met ADC-kanaal `0`. De interruptfunctie kent na afloop van de meting het resultaat toe aan `res`.

Op regel 39 staat de aanroep van de initialisatiefunctie `init_timer`, die de timer instelt op een periodetijd van `0`,125 s. Er is zodoende acht keer per seconde een overflow en een AD-conversie. De timer gebruikt de normale modus en zet op regel 32 expliciet de overflowinterrupt uit. Het eventmechanisme reageert op de overflowbit van de timer. Het interruptmechanisme van de timer hoeft daarvoor niet aan te staan. Als het interruptmechanisme wel aangezet wordt, moet er ook een bijbehorende interruptfunctie zijn.

Code 20.8 : Differentiële conversie op vaste tijdstippen met timer.

```
1  #define F_CPU  2000000UL
2  #include <avr/io.h>
3  #include <avr/interrupt.h>
4  #include <util/delay.h>
5  #include "stream.h"
6
7  void init_adc(void)
8  {
9    PORTA.DIRCLR     = PIN2_bm|PIN3_bm;
10   ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc |
11                      ADC_CH_MUXNEG_PIN3_gc;
12   ADCA.CH0.CTRL    = ADC_CH_INPUTMODE_DIFF_gc;
13   ADCA.CH0.INTCTRL = ADC_CH_INTLVL_LO_gc;
14   ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;
15   ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc |
16                      ADC_CONMODE_bm;
17   ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;
18   ADCA.CTRLA       = ADC_ENABLE_bm;
19   ADCA.EVCTRL      = ADC_SWEEP_0_gc |         // sweep CH0
20                      ADC_EVSEL_0123_gc |      // select event CH0,1,2,3
21                      ADC_EVACT_CH0_gc;        // event triggers ADC CH0
22   EVSYS.CH0MUX     = EVSYS_CHMUX_TCE0_OVF_gc; // event overflow timer E0
23 }
24
25 volatile int16_t res = 0;
26
27 void init_timer(void)
28 {
29   TCE0.PER      = 31249;     // Tper =  8 * (31249 +1) / 2M = 0.125 s
30   TCE0.CTRLA    = TC_CLKSEL_DIV8_gc;          // Prescaling 8
31   TCE0.CTRLB    = TC_WGMODE_NORMAL_gc;        // Normal mode
32   TCE0.INTCTRLA = TC_OVFINTLVL_OFF_gc;        // Interrupt overflow off
33 }
34
35 int main(void)
36 {
37   init_stream(F_CPU);
38   init_adc();
39   init_timer();                               // initialize timer
40
41   PMIC.CTRL |= PMIC_LOLVLEN_bm;
42   sei();
43
44   while(1) {
45     printf("Voltage: %d mV\n", res);
46     _delay_ms(200);
47   }
48 }
49
50 ISR(ADCA_CH0_vect)
51 {
52   res = ADCA.CH0.RES;
53 }
```



Figuur 20.20 : Het event-control-register `EVCTRL` zijn. De `EVSEL`-bits selecteren de eventkanalen waarmee de kanalen van de ADC verbonden. De `SWEEP`-bits geven de ADC-kanalen aan, die bij een *sweep* of bij de freerunningmodus automatisch worden gebruikt. De `EVACT`-bits geven het aantal kanalen aan die gebruikt worden of stellen in dat er een zogenoemde *sweep* wordt gebruikt.

In code 20.9 staat een alternatieve interruptfunctie, die met de methode van code 20.6 de offset meet en de gemeten waarden corrigeert. De functie gebruikt twee statische variabelen voor de som `sum` en het aantal metingen `n`. De interruptfunctie telt het resultaat bij `sum` op als `n` even is en trekt het af als `n` oneven is. In beide situaties worden ook de ingangen omgewisseld. Vervolgens wordt het aantal metingen `n` opgehoogd en als er acht keer gemeten is, krijgt `res` de gemiddelde waarde en worden `sum` en `n` nul gemaakt.

Code 20.9 : Alternatieve interruptfunctie voor AD-conversie, die de offset corrigeert.

```
50  ISR(ADCA_CH0_vect)
51  {
52    static uint8_t n = 0;
53    static int16_t sum = 0;
54
55    if (n & 0x01) {                  //  second (even) measurement
56      sum -= ADCA.CH0.RES;
57      ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN2_gc | ADC_CH_MUXNEG_PIN3_gc;
58    } else {                         //  first (odd) measurement
59      sum += ADCA.CH0.RES;
60      ADCA.CH0.MUXCTRL = ADC_CH_MUXPOS_PIN3_gc | ADC_CH_MUXNEG_PIN2_gc;
61    }
62
63    n++;
64    if ( n == 8 ) {
65      res = sum/8;
66      sum = 0;
67      n = 0;
68    }
69  }
```

Code 20.10 : Differentiële conversie van vier ingangen in freerunningmodus.

```
#define F_CPU  2000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "stream.h"

void set_adcch_input(ADC_CH_t *ch, uint8_t pos_pin_gc, uint8_t neg_pin_gc)
{
  ch->MUXCTRL = pos_pin_gc | neg_pin_gc;
  ch->CTRL    = ADC_CH_INPUTMODE_DIFF_gc;
}

void init_adc(void)
{
  PORTA.DIRCLR     = PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;     // PA3..0 are input
  set_adcch_input(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc, ADC_CH_MUXNEG_GND_MODE3_gc);
  set_adcch_input(&ADCA.CH1, ADC_CH_MUXPOS_PIN1_gc, ADC_CH_MUXNEG_GND_MODE3_gc);
  set_adcch_input(&ADCA.CH2, ADC_CH_MUXPOS_PIN2_gc, ADC_CH_MUXNEG_GND_MODE3_gc);
  set_adcch_input(&ADCA.CH3, ADC_CH_MUXPOS_PIN3_gc, ADC_CH_MUXNEG_GND_MODE3_gc);
  ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc |
                     ADC_CONMODE_bm |
                     ADC_FREERUN_bm;                      // free running mode
  ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;
  ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;
  ADCA.CTRLA       = ADC_ENABLE_bm;
  ADCA.EVCTRL      = ADC_SWEEP_0123_gc|                   // sweep ch. 0,1,2,3
                     ADC_EVSEL_0123_gc|                   // default, no trigger only sweep
                     ADC_EVACT_NONE_gc;                   // no internal or external trigger
}

int main(void)
{
  int16_t  res = 0;
  uint8_t  i = 0;

  init_adc();
  init_stream(F_CPU);
  sei();

  while(1) {
    switch( i & 0x03 ) {
      case 0 :  res = ADCA.CH0.RES; break;
      case 1 :  res = ADCA.CH1.RES; break;
      case 2 :  res = ADCA.CH2.RES; break;
      case 3 :  res = ADCA.CH3.RES; break;
    }

    printf("Voltage %d: %d mV\n", i & 0x03, res);
    i++;
    _delay_ms(1000);
  }
}
```

20.10 Toepassing: differentiële conversie in de freerunningmodus

Met de freerunningmodus worden de ADC-metingen automatisch uitgevoerd. Nadat de AD-conversie klaar is, begint direct daarna de volgende meting. De waarde in het resultaatregister wordt voortdurend aangepast. Het microcontrollerprogramma beschikt dan altijd over de meest actuele waarde.

De toepassing uit code 20.10 meet met behulp van de freerunningmodus vier verschillende analoge signalen. Het hoofdprogramma toont voortdurend de waarde van één van deze signalen op de ledbar. Het `switch`-statement kent — afhankelijk van `i` — aan de variabele `res` het resultaat toe van kanaal 0, 1, 2, of 3. Variabele `i` wordt iedere seconde met één verhoogd.
De initialisatiefunctie gebruikt de functie `set_adcch_input` om de aansluitingen 0, 1, 2, en 3 van poort A te verbinden met respectievelijk kanaal 0, 1, 2, en 3. De negatieve ingang van ADC is in alle vier de gevallen verbonden met `GND`.
Deze toepassing gebruikt geen functie `read_adc`, heeft geen interruptfunctie en ook geen timer nodig. Op regel 22 is het `FREERUN`-bit actief gemaakt. Er is geen interne of externe trigger nodig. Daarom zijn `EVACT`-bits van register `EVCTRL` alle drie laag. De `SWEEP`-bits zijn ingesteld op `0123`, zodat alle vier de kanalen automatisch na elkaar worden uitgelezen.

20.11 Kalibratie van de ADC

De Xmega wordt door Atmel bij de productie getest en verschillende onderdelen worden daarbij gekalibreerd. De kalibratiewaarden worden in het NVM, *non volatile memory*, weggeschreven. De gebruiker kan deze waarden lezen en in de betreffende kalibratieregisters plaatsen.

Code 20.11 definieert een functie `readCalibrationWord` die twee bytes als een 16-bits getal uit het geheugen leest.

Code 20.11 : De functie `readCalibrationWord` leest een 16-bits getal uit het NVM.

```
 1  #include <avr/pgmspace.h>
 2  #include <stddef.h>            // definition of offsetof
 3
 4  uint16_t readCalibrationWord(uint8_t index)
 5  {
 6    uint16_t result;
 7
 8    NVM_CMD = NVM_CMD_READ_CALIB_ROW_gc;
 9    result  = pgm_read_word(index);
10    NVM_CMD = NVM_CMD_NO_OPERATION_gc;
11
12    return result;
13  }
```

Bij de initialisatie van de ADC kan deze functie worden gebruikt om de kalibratiegegevens van de ADC uit het geheugen te lezen en in het kalibratieregister van de ADC te plaatsen:

```
ADCA.CAL = readCalibrationWord( offsetof(NVM_PROD_SIGNATURES_t,ADCACAL0) );
```

De kalibratie betreft volgens Atmel alleen eventuele niet-lineariteiten en is geen correctie voor de offset en de gain, zie paragraaf 28.13 uit de AU-manual.

De metingen in dit hoofdstuk zijn verricht met een niet-ideale opstelling. Verschillende application notes geven daar adviezen over. De gemeten offset kan ook een gevolg zijn een niet perfecte scheiding van het analoge en het digitale aardvlak.

20.12 Resumé ADC

De toepassingen uit dit hoofdstuk tonen aan dat de ADC van de Xmega veel mogelijkheden heeft. Lang niet alle instellingen zijn gebruikt. Sommige mogelijkheden zijn wel genoemd, maar niet toegepast. Alle conversies zijn met 12-bits uitgevoerd. De 8-bits conversie is niet gebruikt. Andere mogelijkheden zijn helemaal niet aan de orde gekomen, zoals de aanpassing van de ingangsimpedantie, het gebruik van een *gain* en het begrenzen van het stroomverbruik.

Voor een nieuwe gebruiker is het aantal instellingen overweldigend. Daarbij komt dat sommige instellingen betrekking hebben op een individueel kanaal en andere op de feitelijke ADC. De inputmodus wordt per kanaal geregeld, maar de resolutie voor de hele ADC. Tabel 20.21 geeft een samenvatting van de belangrijkste instellingen.

Figuur 20.21 : Samenvatting van de belangrijkste instellingen van de ADC.

Instelling	ADC/channel	Opties
conversion mode	ADC	*signed*, *unsigned*
resolution	ADC	8, 12, 12 *left aligned*
freerun	ADC	aan, uit
reference	ADC	1,00 V, VCC/2, VCC/1,6, AREFA, AREFB
prescaling	ADC	4, 8, 16, 32, 64, 128, 256, 512
input mode	channel	*internal*, *single-ended*, *differential*, *differential with gain*
gain	channel	1/2, 1, 2, 4, 8, 16, 32, 64
mux	channel	afhankelijk van de conversiemodus en de inputmodus

Uit de voorbeelden in dit hoofdstuk zijn toch een aantal conclusies te trekken. Het gebruik van de *unsigned single-ended* inputmodus is af te raden. Eenvoudiger is het om de differentiële inputmodus te gebruiken. Met de methode uit code 20.6 kan een eventuele offset eenvoudig gecorrigeerd worden. Bij een voedingsspanning van 3,3 V en een referentiespanning van VCC/1,6 is de gemeten waarde nagenoeg de spanning in millivolts.

Ondanks dat er gewacht moet worden op het resultaat, zal een handmatige meting in veel gevallen voldoen. Bij de oplossing met de interrupt uit code 20.7 is de microcontroller bijna voortdurend met de interrupt bezig. De methode met de interruptfunctie en de timer is veel interessanter. Doordat de interruptfunctie in code 20.8 slechts één keer per seconde wordt uitgevoerd, levert dit nauwelijks overhead op. De processor heeft voldoende ruimte om ook andere taken uit te voeren. Een alternatief is de freerunningmodus, die de processor helemaal niet belast.

21

Seriële communicatie

Doelstelling	Dit hoofdstuk leert je hoe je serieel via SPI en I^{2}C met een extern EEPROM, een externe Real Time Clock en andere componenten kunt communiceren.

Onderwerpen

De behandelde onderwerpen zijn:

- De SPI, *Serial Peripheral Interface*.
- Het benaderen van een extern EEPROM via de SPI.
- Het benaderen van een 74hc595 schuifregister via de SPI.
- Het gebruik van de UART als SPI.
- De I^{2}C-interface of TWI, *Two-Wire serial Interface*.
- Het benaderen van een *real time clock* in de mastermodus met de eenvoudige bibliotheek en met de TWI-drivers van Atmel.
- Levelshifting bij I^{2}C.
- Het gebruik van de drivers van Atmel in de mastermodus.
- Het gebruik van de drivers van Atmel in de slaafmodus.

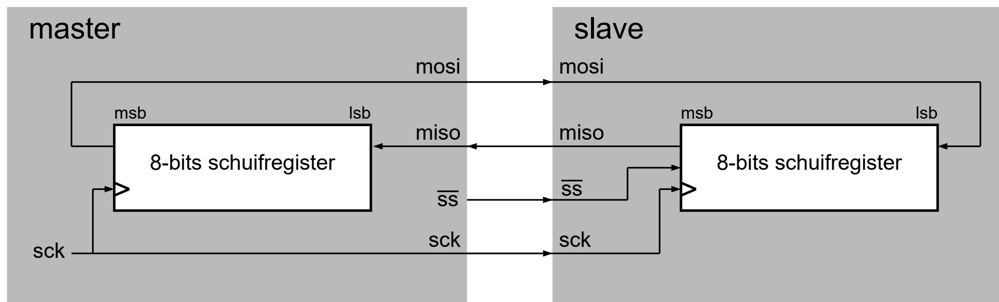
De voorbeelden tonen:

- Een SPI-bibliotheek met functies voor het versturen en ontvangen van gegevens.
- Het schrijven naar en het lezen uit een extern EEPROM via de SPI.
- Het benaderen van een serieel-parallel-schuifregister via de SPI.
- Het benaderen van een serieel-parallel-schuifregister met de UART als SPI.
- De opzet van een eenvoudige bibliotheek voor het lezen van en schrijven via TWI.
- Het gebruik van de eenvoudige bibliotheek bij de *real time clock DS3232*.
- Het gebruik van de TWI-driver van Atmel bij de *real time clock DS3232*.
- Het sturen van gegevens van de master naar de slaaf.
- Het sturen van gegevens van de slaaf naar de master.

Bij de bespreking van de UART in hoofdstuk 19 is al aangegeven dat de Xmega verschillende mogelijkheden voor seriële communicatie heeft. Dit hoofdstuk bespreekt de SPI, *Serial Peripheral Interface*) en de I^{2}C of TWI *Two-Wire serial Interface*. Deze interfaces zijn vooral bedoeld voor de communicatie tussen geïntegreerde schakelingen op een PCB, zoals voor de verbinding met een externe DAC, een seriële LCD, SD-card, of een andere microcontroller. De USB-interface is — net als de USART — bestemd voor communicatie met een PC.

21.1 SPI

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



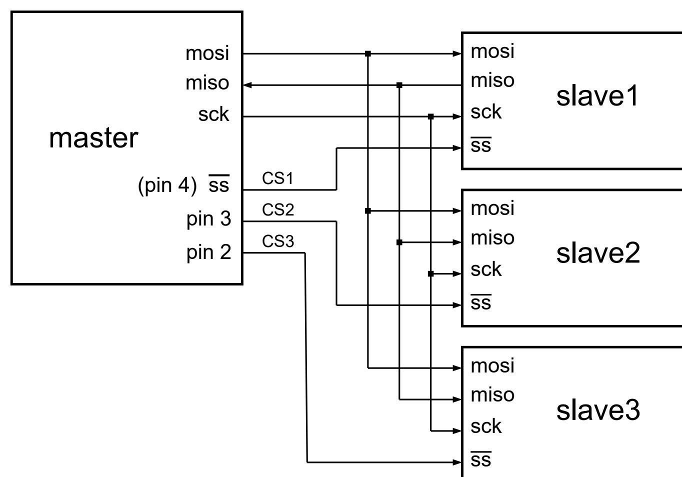
Figuur 21.1 : De SPI met de master en een slave. De `MOSI` van de master is aangesloten aan de `MOSI` van de slave. De `MISO` van de slave is verbonden met de `MISO` van de master. De master genereert de klok voor de slave.

De SPI kent een *master mode* en een *slave mode*. In figuur 21.1 staan de verbindingen tussen de *master* en de *slave*. De kloklijn `SCK` en de selectielijn $\overline{\texttt{SS}}$ zijn uitgangen van de master en ingangen van de slave. De master gebruikt het kloksignaal om waarden in en uit het schuifregister te schuiven. De slave schuift — mits $\overline{\texttt{SS}}$ laag is — de gegevens in het schuifregister bij iedere klokpuls één positie op.

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*).

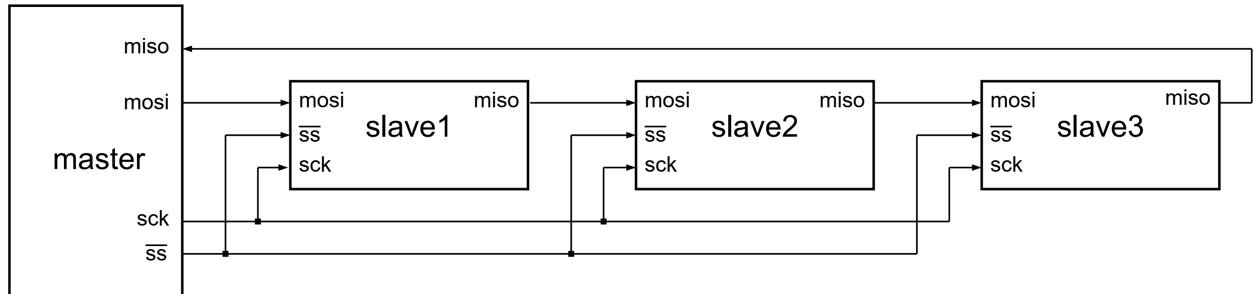
De `MISO`, *Master In Slave Out* van de master is verbonden met de `MISO` van de slave en de `MOSI`, *Master Out Slave In* van de master is verbonden met de `MOSI` van de slave. Hierdoor ontstaat een groot, roterend schuifregister. Na acht klokslagen staat de waarde van de master in het schuifregister 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 21.2 : 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 mastermodus is de selectielijn een gewone uitgang. Elke andere aansluitpin kan eveneens als selectielijn worden gebruikt. Figuur 21.2 geeft een master met drie slaves weer. 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 Xmega één $\overline{\text{SS}}$ heeft, zijn er nog twee andere pinnen nodig, bijvoorbeeld aansluiting 2 en 3 van dezelfde poort.



Figuur 21.3 : De master en drie slaves zijn in een lange seriële ketting geplaatst.

In figuur 21.3 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 één 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 21.1 geeft deze vier modi. De klok kan, als er geen klokpulsen zijn, laag of hoog zijn. Bovendien kan er data worden ingeklokt bij de opgaande en neergaande klokflank. In modus `0` en 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.

Een SPI-signaal kan ook worden verstuurd zonder de SPI-modules van de Xmega. Dat kan ook met een UART, zie paragraaf 21.4, en met *bit banging*.

Bit banging gebruikt geen hardware van de microcontroller. In plaats daarvan worden de uitgangen softwarematig op de juiste momenten hoog en laag gemaakt.

Tabel 21.1 : 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

De SPI van de Xmega gebruikt vier registers: een dataregister `DATA`, een controlregister `CTRL`. een statusregister `STATUS` en een interruptcontrolregister `INTCTRL`. 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 `IF`-flag uit het statusregister hoog als alle acht bits verwerkt zijn.

In code 21.1 en 21.2 staan het bestand `spi.h` en `spi.c` voor de communicatie met de SPI van poort D in de master modus. Bestand `spi.h` bevat een aantal definities en de prototypen van de functies uit `spi.c`. De functie `spi_init` op regel 4 van `spi.c` maakt de aansluiting van de klok, de MOSI en de slave-select uitgang en de MISO ingang. De uitgang slave-select wordt hoog gemaakt en het controlregister krijgt de juiste configuratie.

Code 21.1 : Headerbestand `spi.h` voor communicatie met SPIC in mastermodus.

```
#define SPI_SS_bm      0x10
#define SPI_MOSI_bm    0x20
#define SPI_MISO_bm    0x40
#define SPI_SCK_bm     0x80

#define FOO            0x00

void    spi_init(void);
uint8_t spi_transfer(uint8_t data);
void    spi_write(uint8_t data);
uint8_t spi_read(void);
```

Code 21.2 : C-bestand `spi.c` voor communicatie met SPID in mastermodus.

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

void spi_init(void)
{
  PORTD.DIRSET  =  SPI_SCK_bm|SPI_MOSI_bm|SPI_SS_bm;
  PORTD.DIRCLR  =  SPI_MISO_bm;
  PORTD.OUTSET  =  SPI_SS_bm;
  SPID.CTRL     =  SPI_ENABLE_bm |         // enable SPI
                   SPI_MASTER_bm |         // master mode
                // SPI_CLK2X_bm  |         // no double clock speed
                // SPI_DORD_bm   |         // MSB first
                   SPI_MODE_0_gc |         // SPI mode 0
                   SPI_PRESCALER_DIV4_gc;  // prescaling 4
}

uint8_t spi_transfer(uint8_t data)
{
  SPID.DATA = data;
  while(!(SPID.STATUS & (SPI_IF_bm)));

  return SPID.DATA;
}

void spi_write(uint8_t data)
{
  PORTD.OUTCLR = SPI_SS_bm;
  spi_transfer(data);
  PORTD.OUTSET = SPI_SS_bm;
}

uint8_t spi_read(void)
{
  uint8_t data;

  PORTD.OUTCLR = SPI_SS_bm;
  data = spi_transfer(FOO);
  PORTD.OUTSET = SPI_SS_bm;

  return data;
}
```

De functie `spi_transfer` is de basisfunctie van deze SPI-bibliotheek. De functie kent ingangsparameter `data` toe aan register `DATA` van de SPI en wacht daarna totdat alle acht bits verstuurd zijn via de MOSI. Intussen zijn via de MISO dan acht nieuwe bits vanuit de slave register `DATA` ingeschoven. De functie retourneert na afloop deze acht bits. De functie verstuurt dus de byte die aan de functiewaarde wordt meegegeven en retourneert de ontvangen byte.

De functies `spi_read` en `spi_write` gebruiken de functie `spi_transfer` om een byte te lezen en te schrijven. In beide gevallen wordt de slave-select eerst actief gemaakt en na de overdracht, de transfer, weer inactief. De definitie `FOO` op regel 6 in code 21.1 is een dummy waarde, die bij het lezen van gegevens wordt gebruikt.

21.2 Toepassing: aansturing van een extern EEPROM via de SPI

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.

Code 21.3 : Bestand `spi_eeprom.h` voor de SPI-communicatie met een extern EEPROM.

```
 1 #include "spi.h"
 2
 3 // instruction codes for AT25xxx EEPROM with SPI
 4 #define WREN     6   // set write enable latch
 5 #define WRDI     4   // reset write enable latch
 6 #define RDSR     5   // read status register
 7 #define WRSR     1   // write status register
 8 #define READ     3   // read data from memory array
 9 #define WRITE    2   // write data to memory array
10
11 // function prototypes
12 uint8_t spi_eeprom_read_byte(uint16_t addr);
13 uint8_t spi_eeprom_read_block(uint8_t *dst, uint16_t addr, uint8_t n);
14 void    spi_eeprom_write_byte(uint16_t addr, uint8_t data);
15 void    spi_eeprom_write_block(uint8_t *src, uint16_t addr, uint8_t n);
```

Het EEPROM is als slave op de microcontroller aangesloten, zoals in figuur 21.1 getekend is. In code 21.3 staat een headerbestand `spi_eeprom.h` met een aantal definities. De AT25128 kent zes instructies voor het schrijven en lezen. De definities van deze instructies of opcodes staan vanaf regel 3 in het headerbestand.

In code 21.4 staat een deel van het bestand `spi_eeprom.c` met de routines voor het lezen en schrijven van gegevens naar het EEPROM via de SPI. 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 7 met het selecteren van 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 13 afgesloten met het sturen van de instructie `WRDI` en het deselecteren van de slave.

Code 21.4 : De functies om via een SPI te communiceren met een extern EEPROM.

```
 1  #include <avr/io.h>
 2  #include "spi.h"
 3  #include "spi_eeprom.h"
 4
 5  void spi_eeprom_write_byte(uint16_t addr, uint8_t data)
 6  {
 7    PORTD.OUTCLR = SPI_SS_bm;      // select slave
 8    spi_transfer(WREN);            // send Write Enable
 9    spi_transfer(WRITE);           // send Write
10    spi_transfer(addr>>8);         // send MSB address
11    spi_transfer(addr);            // send LSB address
12    spi_transfer(data);            // send data
13    spi_transfer(WRDI);            // send Write Disable
14    PORTD.OUTSET = SPI_SS_bm;      // deselect slave
15  }
16
17  uint8_t spi_eeprom_read_byte(uint16_t addr)
18  {
19    uint8_t data;
20
21    PORTD.OUTCLR = SPI_SS_bm;      // select slave
22    spi_transfer(READ);            // send Read
23    spi_transfer(addr>>8);         // send MSB address
24    spi_transfer(addr);            // send LSB address
25    data = spi_transfer(F00);      // get data
26    PORTD.OUTSET = SPI_SS_bm;      // deselect slave
27
28    return data;
29  }
```

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 22 de instructie `READ` verstuurd en daarna wordt het adres en de willekeurige byte `F00` 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. De functie `spi_eeprom_read_byte` geeft op regel 28 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. Na het versturen van het adres stuurt de functie op regel 25 de dummywaarde `F00`. Het EEPROM geeft daarop de inhoud van het gestuurde adres 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.

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.

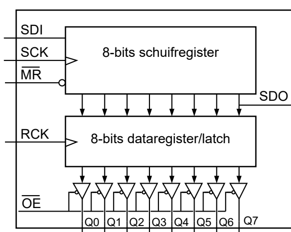
Code 21.5 gebruikt de functies uit code 21.4 om de tafel van veertien in een extern EEPROM op te slaan en vervolgens deze tafel weer te lezen en af te beelden op poort E. Op regel 12 staat de initialisatie van de SPI en op regel 15 worden de waarden uit de tafel van veertien naar het EEPROM geschreven en op regel 22 weer uit het EEPROM gelezen.

Code 21.5 : De communicatie via SPI met behulp van de functies uit code 21.4.

```
1  #define  F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5  #include "spi_eeprom.h"
6
7  int main(void)
8  {
9    uint8_t i;
10
11   PORTE.DIRSET = 0xFF;
12   spi_init();
13
14   for (i=0; i<10; i++) {
15     spi_eeprom_write_byte(i, 14*(i+1));
16   }
17
18   while (1) {
19     if (i==10) {
20       i=0;
21     }
22     PORTE.OUT = spi_eeprom_read_byte(i);
23     _delay_ms(500);
24     i++;
25   }
26 }
```

21.3 Toepassing: aansturing van een schuifregister via een SPI

De SPI heeft in principe vier aansluitingen. Bij veel toepassingen worden drie of zelfs twee aansluitingen gebruikt. Een voorbeeld is de aansturing van een 74hc595. Dit is een 8-bits serieel-in, parallel-uit schuifregister met uitgangsregister en tristate uitgangen. De component heeft twee kloklijnen `SCK` en `RCK`, een actief lage master reset `MR`, een seriële ingang `SDI`, een seriële uitgang `SDO`, een actief lage output-enable `OE` en acht uitgangen.

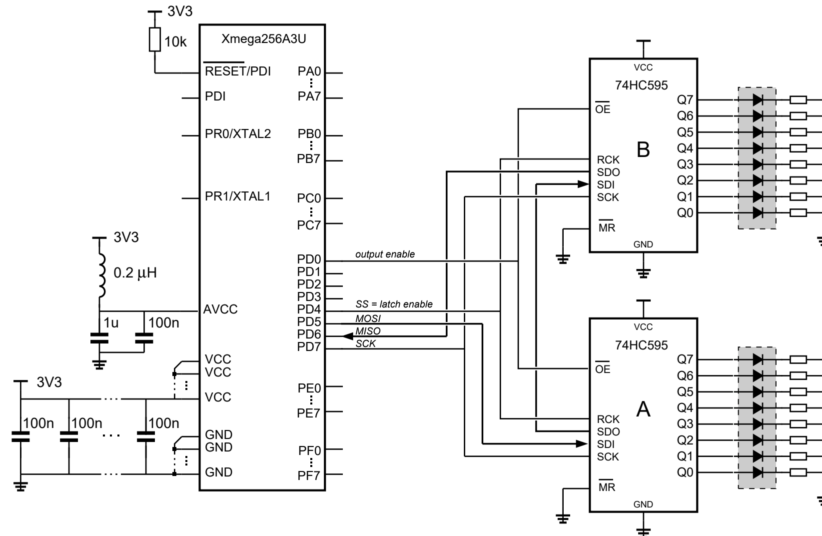


Figuur 21.4 : Het blokschema van de 74hc595.

In figuur 21.4 staat het blokschema van de 74hc595. Bij iedere opgaande flank van kloksignaal `SCK` schuift het schuifregister via signaal `SDI` een databit naar binnen en via `SDO` een databit naar buiten. De opgaande flank van kloksignaal `RCK` plaatst de inhoud van het schuifregister in het dataregister. Als signaal `OE` laag is staat de waarde uit het dataregister op de uitgangen.

De datasheets van deze component gebruiken vaak andere namen voor de verschillende aansluitingen. Ondanks het feit dat het dataregister op een klokflank reageert, wordt het dataregister ook wel latch genoemd en noemt men signaal `RCK` een latch-enable.

De schakeling van figuur 21.5 verbindt de SPI van poort D met een keten van twee 74hc595's. Aan de uitgangen van de 74hc595's zijn zestien leds aangesloten. In code 21.6 staat een programma dat voortdurend de getallen `0` tot en met 65355 via de SPI naar de keten met 74hc595's stuurt. De meest significante byte wordt eerst verstuurd en daarna de minst significante byte. De meest significante byte



Figuur 21.5 : Een schakeling met een keten van twee 74hc595 schuifregisters.
De `MOSI` van de D-poort van de Xmega is verbonden met de `SDI` van register A. De `SDO` van register A is doorverbonden met de `SDI` van register B. Tenslotte is de `SDO` van register B verbonden is met de `MISO` van de Xmega.

komt in register B te staan en daarna de minst significante byte in register A. De leds uit figuur 21.5 tonen voortdurend de binaire waarde van het laatst verzonden getal.

Het programma uit code 21.6 gebruikt de functies `spi_init` en `spi_transfer` uit code 21.2. Na de initialisatie verstuurt in de oneindige lus de functie `spi_transfer` op regel 26 de meest significante byte en op regel 27 de minst significante byte van het getal. Op regel 29 plaatst de functie `latch_data` het getal in de dataregisters van de 74hc595's. De functie `latch_data` doet dit door de `latch_enable` even hoog en direct weer laag te maken. De output-enable van de 74hc595's is bij de initialisatie laag gemaakt, zodat op de uitgangen altijd de inhoud van de dataregisters weergeven.

Bij de declaratie van de functie `latch_data` staat de *qualifier* `inline`. Dit betekent dat de compiler de functie niet als gewone functie implementeert, maar de body — als een soort macrodefinitie — direct in de code invult. Het voordeel is dat er niet elke keer een functie-call nodig is, maar dat de inhoud van de functie direct uitgevoerd wordt.

In plaats van een inline-functie had `latch_data` — net als `output_enable_on()` en `output_enable_off()` — ook als een macro gedefinieerd kunnen worden.
Macro's met meer regels zijn vaak onoverzichtelijk:

```
#define latch_data()    PORTD.OUTSET = SPI_LATCH_bm; \
                        PORTD.OUTCLR = SPI_LATCH_bm
```

De backslash negeert het einde van de regel. De macrodefinitie gaat daardoor verder op de volgende regel. Er staat geen puntkomma na het laatste statement. Deze wordt bij de aanroep van de macro toegevoegd:

```
latch_data();
```

Code 21.6 : Het versturen van een reeks van 256 bytes via de SPI naar een 74HC595.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5  #include "spi.h"
6
7  #define SHFT_LATCH_bm        PIN4_bm   // is SPI_SS_bm
8  #define SHFT_OE_bm           PIN0_bm
9  #define output_enable_on()   PORTD.DIRCLR = SHFT_OE_bm
10 #define output_enable_off()  PORTD.DIRSET = SHFT_OE_bm
11
12 inline void latch_data(void)
13 {
14   PORTD.OUTSET = SHFT_LATCH_bm;
15   PORTD.OUTCLR = SHFT_LATCH_bm;
16 }
17
18 int main(void)
19 {
20   uint16_t i = 0;
21
22   spi_init();
23   output_enable_on();
24
25   while (1) {
26     spi_transfer(i>>8);       // transfer high byte
27     spi_transfer(i&0x00FF);   // transfer low byte
28
29     latch_data();
30
31     _delay_ms(100);
32     i++;
33   }
34 }
```

Bij de inline-functie bepaalt de compiler of de functie direct in de code wordt ingevuld of dat deze toch als een gewone functie wordt geïmplementeerd. Bij de macrodefinitie wordt de tekst altijd letterlijk in de code ingevuld.

De macro's `output_enable_on()` en `output_enable_off()` kunnen gebruikt worden om de leds te dimmen door in code 21.6 de tijdvertraging van regel 31 te vervangen door:

```
  for(int j=0; j<100; j++) {
    output_enable_on();
    _delay_us(100);
    output_enable_off();
    _delay_us(900);
  }
```

De leds zijn dan 100 keer achter elkaar ongeveer 10 μs aan en 90 μs uit. De uitgangen knipperen met een frequentie van ongeveer 1 kHz en met een duty-cycle van 10%.

21.4 De USART als SPI

De USART's van de Xmega kunnen ook als SPI worden gebruikt. De aansturing van een USART in SPI-mode lijkt sterk op de aansturing via een gewone SPI. De initialisatie van de USART in de SPI-modus heeft meer overeenkomsten met de initialisatie van de SPI dan met de initialisatie als gewone UART. In mastermodus lijkt het versturen en ontvangen van gegevens eveneens sterk op de werkwijze bij de gewone SPI. De te verzenden byte wordt nu in het `DATA`-register van de UART geplaatst en daarna wordt er weer gewacht totdat de byte verzonden is.

Naast het feit dat de Xmega256a3u zeven USART's en slechts drie SPI's bevat, is er nog een reden om de SPI van de USART's te gebruiken. De gewone SPI kan alleen als slave DMA, *direct memory access*, gebruiken. Met de USART als SPI kan in de mastermodus wel DMA worden toegepast.

De baudsnelheid hangt af van de waarde van `BSEL` en van de klokfrequentie f_{cpu}:

$$f_{\text{baud}} = \frac{f_{\text{cpu}}}{2\,(BSEL+1)} \tag{21.1}$$

De snelheid is maximaal als `BSEL` gelijk is aan `0` is en dat is gelijk aan de maximale snelheid van de gewone SPI. In beide gevallen is de maximale frequentie de helft van de klokfrequentie. Bij beide methoden zijn er voor het versturen van één bit twee klokslagen nodig.

De SPI van de USART heeft geen slave-select. Als er een slave-select nodig is, kan hiervoor iedere andere pin gebruikt worden. De Xmega heeft bij alle poorten C tot en met F in principe een SPI en twee USART's. De Xmega256a3u heeft bij poort F alleen een USART0. De aansluitpinnen van de SPI en de USART's zijn voor iedere poort gelijk. In tabel 21.2 staat een overzicht met de pinnummers van de verschillende SPI-aansluitingen.

Tabel 21.2 : De aansluitpinnen van de SPI en de SPI van de UART's.

SPI aansluiting	USART0	USART1	SPI	SPI (remap)
MOSI	3 (TxD)	7 (TxD)	5	7
MISO	2 (RxD)	6 (RxD)	6	6
SCK	1 (XCK)	5 (XCK)	7	5
SS			4	4

Bij de SPI is pin 5 de MOSI en pin 7 de klok. Voor USART1 is dat precies andersom: pin 5 is dan de klok en pin 7 de MOSI of `TxD`. Bij de gewone SPI kunnen de aansluitingen MOSI en klok omgewisseld worden. Dit wordt gedaan met het `SPI`-bit uit het `REMAP`-register van de betreffende poort:

```
PORTx.REMAP |= PORT_SPI_bm;
```

Sommige ontwerpers voegen bovenstaande toekenning standaard toe aan de initialisatiefunctie van de SPI om pincompatibel te zijn met de SPI van de USART. Op een later moment in het ontwerpproces kan dan — zonder dat de hardware wijzigt — gekozen worden voor de gewone SPI of een SPI van de USART1.

Code 21.7 definieert voor de SPI van USART1 van poort D een initialisatiefunctie `spi_init` en een transferfunctie `spi_transfer`, die een byte kan versturen en ontvangen. De functie `spi_init` legt de richtingen van de in- en uitgangen vast en selec-

Code 21.7 : `uartspi.c` **voor de communicatie met USART1 van poort D in SPI-modus.**

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

void spi_init(void)
{
  PORTD.DIRSET  = PIN7_bm|PIN5_bm;            // MOSI (txd), SCK (xck)
  PORTD.DIRCLR  = PIN6_bm;                    // MISO (rxd)
  PORTD.DIRSET  = PIN4_bm;                    // slave select
  PORTD.OUTSET  = PIN4_bm;                    // active low

  USARTD1.BAUDCTRLA = 0;                      // max. baudrate: FCPU/2
  USARTD1.BAUDCTRLB = 0;
  USARTD1.CTRLC = USART_CMODE_MSPI_gc;        // MSB first, SPI Mode 0
  USARTD1.CTRLA = 0;                          // no interrupt
  USARTD1.CTRLB = USART_TXEN_bm|USART_RXEN_bm; // enable TX and RX
}

uint8_t spi_transfer(uint8_t data)
{
  USARTD1.DATA = data;                        // send data
  while( !(USARTD1.STATUS & USART_TXCIF_bm) ); // wait for TX complete
  USARTD1.STATUS |= USART_TXCIF_bm;           // clear TX interrupt flag

  return USARTD1.DATA;
}
```

teert pin 4 als slave-select. Regel 12 stelt de `CMODE`-bits van register `CTRLC` in op `MSPI`. In deze masterspi-modus hebben de overige bits van `CTRLC` een andere betekenis dan bij de asynchrone modus van figuur 19.5. In dit geval zijn er twee bits `UDORD` en `UCPHA`, zie figuur 21.6.
De eerste bepaalt de bitvolgorde en de tweede bepaalt de fase (CPHA) van het datasignaal ten opzichte van de klok. De polariteit van de klok (CPOL) is niet instelbaar, zodat bij de USART de spi-modus alleen `0` of 1 kan zijn.

Figuur 21.6 : De bits van het `CTRLC`-register bij de masterspi-modus.

De functie `spi_transfer` zet de te versturen byte in het `DATA`-register en wacht totdat de byte helemaal verstuurd is. Dit is het geval als de interruptvlag `TXCIF` van register `STATUS` hoog is. Op regel 21 wordt deze interruptvlag direct weer laag gemaakt door er een 1 naar toe te schrijven.

De functies `spi_init` en `spi_transfer` zijn gelijkwaardig met de functies `spi_init` en `spi_transfer` uit het bestand `spi.c` van code 21.2. Het voorbeeld van code 21.6 uit paragraaf 21.3 kan zonder aanpassingen direct worden gebruikt in combinatie met de functies `spi_init` en `spi_transfer` uit code 21.7.
Wel moet de hardware uit figuur 21.5 worden aangepast. Bij de gewone SPI is D7 de klok en D5 de MOSI, maar bij USART1 is D5 de klok en D7 de MOSI. In het schema moeten de aansluitingen van pin D7 en pin D5 worden verwisseld.

21.5 I^{2}C

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

Spreek I^{2}C 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.

Binnen het I^{2}C-protocol heeft elke type component een unieke identificatiecode. NXP, voorheen een onderdeel van Philips, stelt deze codes in licentie ter beschikking. Als de microcontroller master is, heeft de I^{2}C-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. Atmel gebruikt — evenals veel andere fabrikanten – in plaats van de naam I^{2}C de afkorting TWI. De afkorting TWI staat voor *two-wire interface* of *two-wire serial interface*.

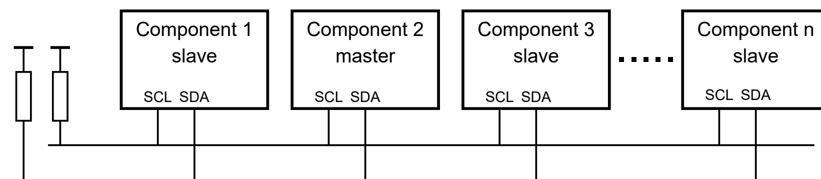
SPI en I^{2}C zijn beide seriële communicatieprotocollen. De voordelen van SPI: de kloksnelheid kan hoger zijn dan I^{2}C, er is geen extra hardware nodig en er hoeven geen extra gegevens, zoals de identificatiecode, verstuurd te worden. De voordelen van I^{2}C: er is een officiële standaard, er zijn minder lijnen nodig, in een systeem met veel componenten blijft het aantal lijnen twee en het is geschikt voor een multi-master systeem.

I^{2}C 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.

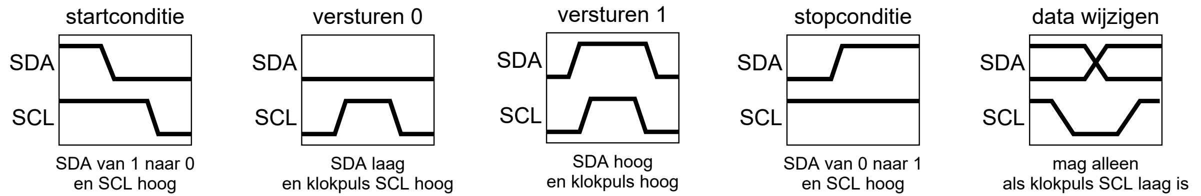
Componenten, die met een I^{2}C-interface verkrijgbaar zijn, zijn onder andere ADC, DAC, EEPROM, flash, SDRAM, port expander, *real time clock*, LCD, temperatuursensor en vele andere sensoren.



Figuur 21.7 : Een I^{2}C-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.

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.

In figuur 21.7 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 I^{2}C-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 21.8 : De betekenis van de verschillende signaalcondities met de `SDA`- en de `SCL`-lijn.

In figuur 21.8 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.

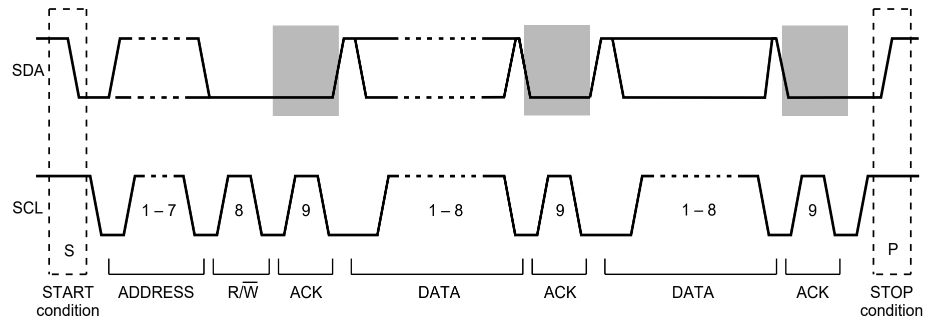
Figuur 21.9 : Het protocol voor het versturen van gegevens door de master naar een slave. Na de startconditie `START` verstuurt de master het slave-adres `ADDRESS`, het schrijfbit `W` en de te versturen gegevens `DATA`. Na ontvangst van elke byte antwoordt de slave door het bevestigingsbit `ACK` op de bus te zetten. Tenslotte sluit de master de communicatie met de `STOP`-conditie.

Figuur 21.9 toont het I²C-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 met het schrijfbit `W` met 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 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.

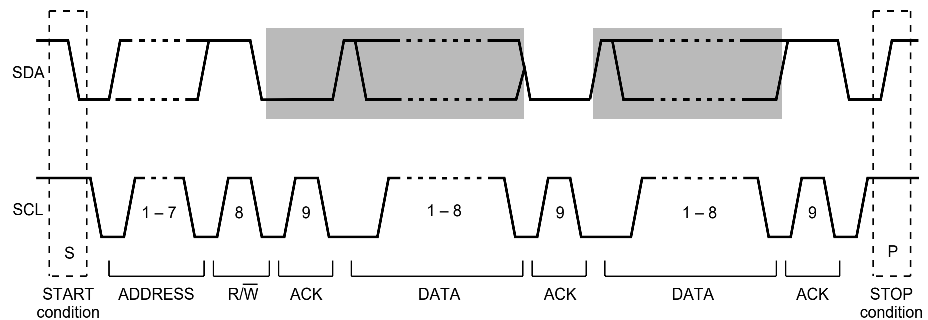
Figuur 21.11 toont de signaalwaarden van het protocol van figuur 21.9. Het kloksignaal wordt door de master gegenereerd. De slave zet de bevestigingsbits op de datalijn. De rest van het datasignaal wordt door de master gemaakt.

Figuur 21.10 : Het protocol voor het ontvangen van gegevens door de master van 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` ($\overline{\mathtt{ACK}}$). Zo weet de slave dat er geen gegevens meer verstuurd hoeven te worden. De master beëindigt de communicatie met de `STOP`-conditie.

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. De master stuurt het slave-adres en het leesbit `R` met de waarde 1. 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 voldoende



Figuur 21.11 : 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.



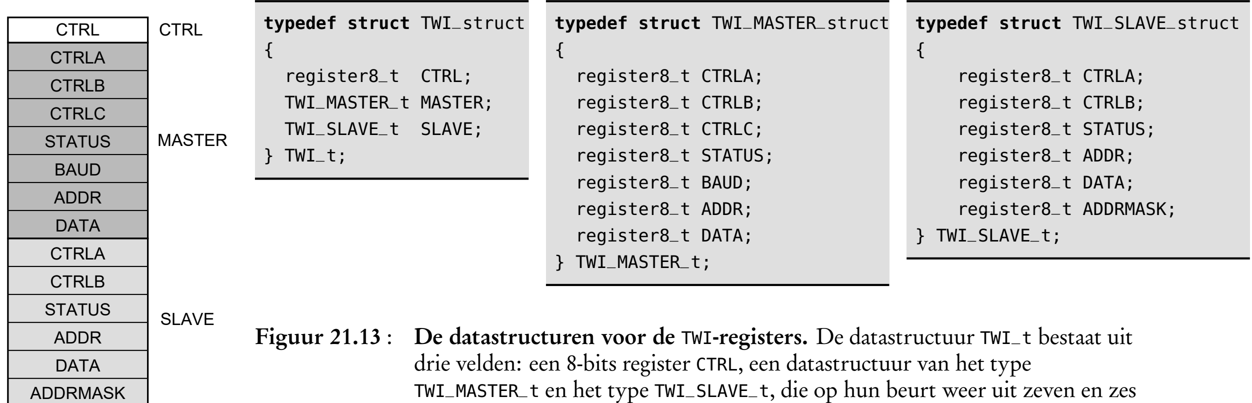
Figuur 21.12 : De signalen bij het ontvangen van twee databytes door de master. De bits, die door de slave gegenereerd worden, hebben een grijze achtergrond.

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

Figuur 21.12 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 overige bevestigingsbits zet de master op de datalijn.

21.6 I²C of TWI voor de Xmega

I²C is bij de Xmega van Atmel geïmplementeerd als TWI, een zogenoemde two-wire serial interface. De Xmega kan zowel master als slave zijn. In de mastermodus gebruikt de Xmega andere registers dan in de slave-modus. In figuur 21.13 is de datastructuur `twi_t` getekend. Deze datastructuur bestaat uit drie velden: een controlregister voor algemene instellingen en een datastructuur van het type `TWI_MASTER_t` en een datastructuur van het type `TWI_SLAVE_t`. De velden `MASTER` en `SLAVE` bevatten de registers voor het gebruik van de TWI als master en als slave. Beide datastructuren hebben een aantal controlregisters, een dataregister `DATA`, een statusregister `STATUS` en een adresregister `ADDR`.




```
typedef struct TWI_struct
{
  register8_t  CTRL;
  TWI_MASTER_t MASTER;
  TWI_SLAVE_t  SLAVE;
} TWI_t;
```

```
typedef struct TWI_MASTER_struct
{
  register8_t CTRLA;
  register8_t CTRLB;
  register8_t CTRLC;
  register8_t STATUS;
  register8_t BAUD;
  register8_t ADDR;
  register8_t DATA;
} TWI_MASTER_t;
```

```
typedef struct TWI_SLAVE_struct
{
    register8_t CTRLA;
    register8_t CTRLB;
    register8_t STATUS;
    register8_t ADDR;
    register8_t DATA;
    register8_t ADDRMASK;
} TWI_SLAVE_t;
```

Figuur 21.13 : De datastructuren voor de `TWI`-registers. De datastructuur `TWI_t` bestaat uit drie velden: een 8-bits register `CTRL`, een datastructuur van het type `TWI_MASTER_t` en het type `TWI_SLAVE_t`, die op hun beurt weer uit zeven en zes 8-bits registers bestaan.

In de mastermodus wordt de communicatie gestart door het *slave address* van de slave naar het adresregister te schrijven en data wordt verstuurd door deze naar het dataregister te schrijven. Data wordt ontvangen door het dataregister uit te lezen. Het statusregister bevat een aantal bits, die aangeven of de data correct is ontvangen en verstuurd.

21.7 Eenvoudige I^{2}C-bibliotheek voor de Xmega in mastermodus

Deze paragraaf bespreekt een eenvoudige I^{2}C-bibliotheek, waarbij de Xmega als master wordt gebruikt en die aansluit bij een stijl, die veel toegepast wordt bij andere microcontrollers. Deze bibliotheken bevatten functies om de communicatie te initialiseren, te starten, te herstarten en te stoppen en bevatten functies voor het lezen en schrijven van databytes.

Het slave-adres is een 7-bits getal. Als 8-bits getal kan deze links of rechts uitgelijnd zijn. Deze bibliotheek gebruikt de rechts uitgelijnde versie om aan te sluiten bij de bibliotheken van Atmel.

In code 21.9 staat het headerbestand `i2c.h` met de prototypen van de functies en een aantal macrodefinities. In code 21.8 staat het bestand `i2c.c` met de functies. Naast een functie voor de initialisatie zijn er vijf functies voor het schrijven, het lezen en het starten, herstarten en stoppen van de communicatie.

De meeste microcontrollers starten de communicatie door een bit in een controlregister te zetten. Bij de Xmega start de communicatie automatisch als het *slave address* en de R/W-bit naar het adresregister `ADDR` van de master worden geschreven. Bij de functie `i2c_start` gebeurt dit in code 21.8 op regel 15, nadat gecontroleerd is of de I^{2}C-bus beschikbaar is. De functie wacht totdat het *slave address* verstuurd is en controleert daarna of het `ACK`-bit ontvangen is.

Het *idle* maken van de bus bij het stoppen, is strikt genomen niet nodig. Dat gebeurt na enige tijd automatisch. Het voordeel is dat er hiermee ook een restart kan worden gedaan door eerst `i2c_stop` en daarna `i2c_start` aan te roepen.

De functie `i2c_restart`, die in code 21.8 ontbreekt, is nagenoeg identiek met `i2c_start`. Alleen de test van regel 13 ontbreekt. De communicatie is immers al aanwezig, de functie `i2c_restart` is bedoeld om deze te herstarten nadat er bijvoorbeeld eerst een data-adres naar de slave is gezonden. De functie `i2c_stop` stopt de communicatie en maakt I^{2}C-bus *idle*.

Code 21.8 : Het bestand `i2c.c` met de I²C-functies voor het versturen en ontvangen.

```
 1  #include "i2c.h"
 2
 3  void i2c_init(TWI_t *twi, uint8_t baudRateRegisterSetting)
 4  {
 5    twi->MASTER.BAUD   = baudRateRegisterSetting;
 6    twi->MASTER.CTRLC  = 0;
 7    twi->MASTER.CTRLA  = TWI_MASTER_ENABLE_bm;
 8    twi->MASTER.STATUS = TWI_MASTER_BUSSTATE_IDLE_gc;
 9  }
10
11  uint8_t i2c_start(TWI_t *twi, uint8_t address, uint8_t rw)
12  {
13    if ( (twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_gm) !=                      // if bus available
14                        TWI_MASTER_BUSSTATE_IDLE_gc ) return I2C_STATUS_BUSY; //
15    twi->MASTER.ADDR = (address << 1) | rw;                                    // send slave address
16    while( ! (twi->MASTER.STATUS & (TWI_MASTER_WIF_bm << rw)) );               // wait until sent
17
18    if ( twi->MASTER.STATUS & TWI_MASTER_RXACK_bm ) {                          // if no ack
19      twi->MASTER.CTRLC = TWI_MASTER_CMD_STOP_gc;
20      return I2C_STATUS_NO_ACK;
21    }
22
23    return I2C_STATUS_OK;
24  }
                ⋮   i2cstart en i2cstop zijn hier weggelaten
45  uint8_t i2c_write(TWI_t *twi, uint8_t data)
46  {
47    twi->MASTER.DATA = data;                                                   // send data
48    while( ! (twi->MASTER.STATUS & TWI_MASTER_WIF_bm) );                       // wait until sent
49
50    if ( twi->MASTER.STATUS & TWI_MASTER_RXACK_bm ) {                          // if no ack
51      twi->MASTER.CTRLC = TWI_MASTER_CMD_STOP_gc;
52      return I2C_STATUS_NO_ACK;
53    }
54
55    return I2C_STATUS_OK;
56  }
57
58  uint8_t i2c_read(TWI_t *twi, uint8_t ack)
59  {
60    uint8_t data;
61
62    while( ! (twi->MASTER.STATUS & TWI_MASTER_RIF_bm) );                       // wait until received
63    data = twi->MASTER.DATA;                                                   // read data
64    twi->MASTER.CTRLC = ((ack==I2C_ACK) ? TWI_MASTER_CMD_RECVTRANS_gc :        // send ack (go on) or
65                              TWI_MASTER_ACKACT_bm|TWI_MASTER_CMD_STOP_gc); //     nack (and stop)
66
67    if ( ack == I2C_NACK ) {
68      while( ! (twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_IDLE_gc) );
69    }
70
71    return data;
72  }
```

Code 21.9 : Het headerbestand `i2c.h` met definities en prototypen.

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

#define TWI_BAUD(F_SYS, F_TWI)   ((F_SYS / (2 * F_TWI)) - 5)

#define I2C_ACK     0
#define I2C_NACK    1
#define I2C_READ    1
#define I2C_WRITE   0

#define I2C_STATUS_OK      0
#define I2C_STATUS_BUSY    1
#define I2C_STATUS_NO_ACK  2

void    i2c_init(TWI_t *twi, uint8_t baudRateRegisterSetting);
uint8_t i2c_start(TWI_t *twi, uint8_t address, uint8_t rw);
uint8_t i2c_restart(TWI_t *twi, uint8_t address, uint8_t rw);
void    i2c_stop(TWI_t *twi);
uint8_t i2c_write(TWI_t *twi, uint8_t data);
uint8_t i2c_read(TWI_t *twi, uint8_t ack);
```

De functie `i2c_write` verstuurt een byte door deze in het dataregister `DATA` van de master te plaatsen. De functie wacht tot de byte verzonden is en controleert vervolgens of het `ACK`-bit ontvangen is. Het onderscheid tussen lezen en schrijven wordt gemaakt doordat bij het versturen van het slave adres de `R/W̅`-bit hoog of laag is. De functie `i2c_write` is alleen zinvol als de schrijfmodus (`W`) actief is en de `i2c_read` is alleen zinvol in de leesmodus (`R`)

De functie `i2c_read` wacht tot er een byte ontvangen is en kent het resultaat toe aan de lokale variabele `data`. Daarna stuurt de functie een `ack` als er nog meer gelezen moet worden en een `nack` als alle bytes gelezen zijn. De functie `i2c_read` geeft na het lezen van de byte automatisch een `ACK` of een `NACK` door in het `CTRLC`-register het `ACKACT`-bit en de `CMD`-bits de juiste waarden te geven. Als `i2c_read` alle bytes gelezen heeft, wacht de functie tot de bus *idle* is en geeft tenslotte de gelezen databyte terug.

De functie `i2c_init` initialiseert de TWI-interface en stelt de juiste baudsnelheid in, stelt de mastermodus in en maakt de busstatus *idle*. De frequentie van de communicatie hangt af van de systeemklok f_{cpu} en de waarde van register `BAUD`:

$$f_{\text{twi}} = \frac{f_{\text{cpu}}}{2\,(5 + \texttt{BAUD})} \tag{21.2}$$

Formule 21.3 geldt voor baudsnelheden tot 100 kHz. Voor een baudsnelheid van 400 kHz meldt de AU-manual dat er mogelijk een hogere waarde voor `BAUD` nodig is.

Voor een bepaalde klokfrequentie is de waarde van `BAUD` dan:

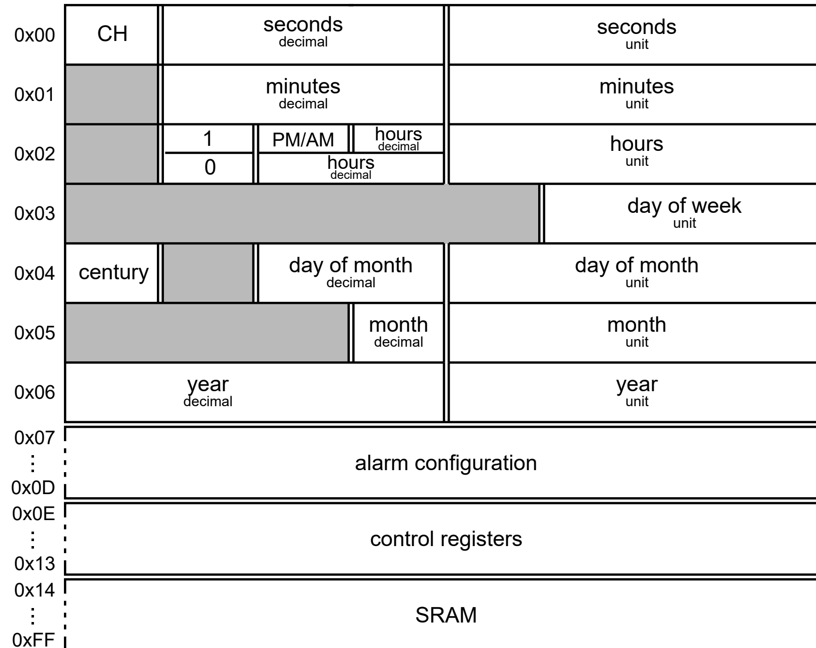
$$\texttt{BAUD} = \frac{f_{\text{cpu}}}{2\,f_{\text{twi}}} - 5 \tag{21.3}$$

In het headerbestand `i2c.h` uit code 21.9 staat op regel 3 de macro `TWI_BAUD` die met formule 21.3 de juiste waarde voor `BAUD` berekent.

21.8 Toepassing: eenvoudige I^{2}C-bibliotheek bij een DS3232

Er bestaat een grote variëteit aan realtimeklokken met I^{2}C. NXP levert de PCF8563 en de PCF8583. De DS1307 van Maxim, voorheen Dallas Semiconductor, was en is zeer populair. Maxim levert veel verschillende typen. De DS3232 is relatief duur, maar heeft een ingebouwde temperatuur-gecompenseerde kristaloscillator en is geschikt voor 3,3 V.

Een voorbeeld van een I^{2}C-component is de DS3232 *real time clock* van Maxim. Deze component bevat een oscillator en houdt de tijd en de datum bij. Deze gegevens worden bewaard in een 256 × 8 NV-RAM, *Non Volatile RAM* en zijn via I^{2}C van buitenaf bereikbaar.



Figuur 21.14 : De geheugenindeling bij de DS3232 van Maxim. De gegevens zijn BCD gecodeerd. De lage nibbles bevatten de eenheden en de hoge nibbles de tientallen. De grijs gekleurde bits zijn ongebruikt.

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

De datastructuur is anders voor de diverse realtimeklokken. De eerste registers uit figuur 21.14 zijn meestal wel aanwezig. Zo heeft de DS1307 geen alarmfuncties en geen `Century`-bit, die verandert als het jaartal van 99 naar 00 gaat.

In figuur 21.14 staat de geheugenindeling. De eerste zeven 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.
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.

Bij de verwerking van de datum en de tijd is het handig om de gegevens van de DS3232 eerst in een datastructuur te plaatsen. Dat kan bijvoorbeeld een array van zeven bytes zijn:

```
uint8_t ds3232[7];
```

Het eerste byte `ds3232[0]` is dan het aantal seconden en het zevende byte `ds3232[6]` het jaar.

Code 21.10 : Deel van `rtc.h` met definities voor de DS3232 real time clock.

```
1  #define RTC_SLAVE_ADDRESS 0x68  // 1101000
2
3  #define RTC_SECOND        0x00
4  #define RTC_MINUTE        0x01
5  #define RTC_HOUR          0x02
6  #define RTC_DAY           0x03
7  #define RTC_DATE          0x04
8  #define RTC_MONTH         0x05
9  #define RTC_YEAR          0x06
10
11 void  rtc_set_time(TWI_t *twi);
12 void  rtc_set_date(TWI_t *twi);
13 int   rtc_get_time(TWI_t *twi);
14 int   rtc_get_date(TWI_t *twi);
15 char *rtc_time_to_string(char *s);
16 void  string_to_rtc_time(char *s);
17
18 struct rtc_time {
19   uint8_t second;
20   uint8_t minute;
21   uint8_t hour;
22 };
23
24 struct rtc_date {
25   uint8_t day;
26   uint8_t month;
27   uint8_t year;
28 };
```

Code 21.11 : Deel van `rtc.c` met de functies `rtc_set_time` en `rtc_get_date`.

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

In code 21.10 en code 21.11 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 verwijzing naar een veld zeer goed leesbaar is: `time.hour` geeft het uur.

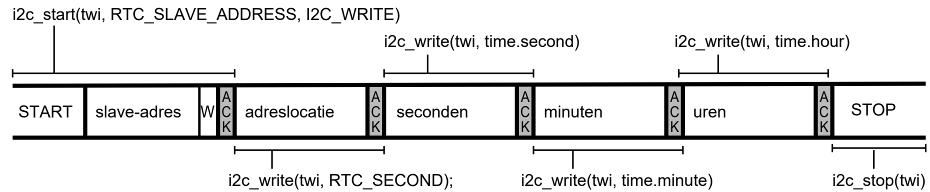
De dag van de week is in dit voorbeeld weggelaten. Dit veld kan worden toegevoegd aan de structuur `date`.

Voor het instellen van een DS3232 real time clock wordt na het slave-adres en het schrijfbit eerst de adreslocatie verstuurd van waar de bytes ingesteld moeten worden. Na het adres van de geheugenlocatie volgen de bytes.

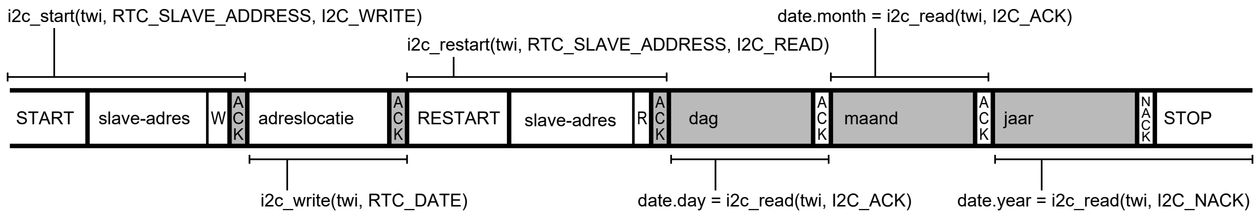
De functie `rtc_set_time` uit code 21.11 stelt de tijd van de DS3232 in. Na het slave-adres volgt eerst het adres `RTC_SECOND` van de locatie waar de seconden staan. Daarna volgen de drie bytes met de nieuwe waarden uit de structuur `time` met de seconden, minuten en uren. Figuur 21.15 toont het dataformaat dat verstuurd wordt samen met de toewijzingen uit de functie `rtc_set_time`.

Het slave-adres is een 7-bits getal. Als 8-bits getal kan deze links of rechts uitgelijnd zijn. Voor de DS3232 is dat respectievelijk `0xD0` of `0x68`. Deze I^2C-bibliotheek gebruikt de rechts uitgelijnde versie om aan te sluiten bij de bibliotheken van Atmel.

Voor het uitlezen van een DS3232 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.
De functie `rtc_get_date` uit code 21.11 leest de datum uit de DS3232. Na het slave-adres met de schrijfbit (`W`) wordt eerst het adres `RTC_DATE`, van de locatie waar de



Figuur 21.15 : Het dataformaat voor het instellen van de tijd bij de DS3232.



Figuur 21.16 : Het dataformaat voor het lezen van de datum bij de DS3232.

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 21.16 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 21.11.

Code 21.12 : Conversiefuncties uit `rtc.c` voor het omzetten van het `rtc_time`.

```
68 char *rtc_time_to_string(char *s)
69 {
70   s[0] = 0x30 | ((time.hour & 0x30) >> 4);
71   s[1] = 0x30 |  (time.hour & 0x0F);
72
73   s[3] = 0x30 | ((time.minute & 0x70) >> 4);
74   s[4] = 0x30 |  (time.minute & 0x0F);
75
76   s[6] = 0x30 | ((time.second & 0x70) >> 4);
77   s[7] = 0x30 |  (time.second & 0x0F);
78
79   return s;
80 }
```

```
82 void string_to_rtc_time(char *s)
83 {
84   time.hour   = ((s[0] & 0x03)<<4) | (s[1] & 0x0F);
85   time.minute = ((s[3] & 0x07)<<4) | (s[4] & 0x0F);
86   time.second = ((s[6] & 0x07)<<4) | (s[7] & 0x0F);
87 }
```

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 21.12 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, vier posities naar rechts geschoven en op dezelfde wijze omgezet naar de ASCII-waarde van het betreffende cijfer. 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 toe 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.

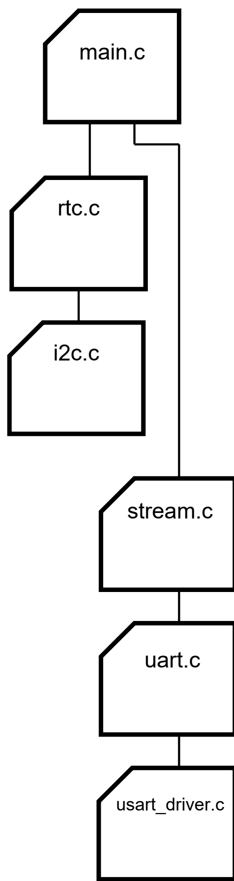
Het voorbeeld uit code 21.13 gebruikt voor de UART de wrapper uit paragraaf 19.9 en de `stream.c` uit paragraaf 19.12. Dit kan ook met alleen `serialF0.c` uit paragraaf 19.12.

De macro `clear_screen()` is gedefinieerd in `stream.h` en maakt het terminalvenster leeg bij een Putty en Tera Term. Bij andere hyperterminals kan het zijn dat dit niet werkt.

Code 21.13 : Programma dat de tijd instelt en iedere seconde een tijdmelding geeft.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6  #include <stdio.h>
7
8  #include "stream.h"                              // or use serialF0.h
9
10 #define BAUD_100K       100000UL
11 #include "i2c.h"
12 #include "rtc.h"
13
14 int main(void)
15 {
16   char t[]="HH:MM:SS";
17
18   i2c_init(&TWIE, TWI_BAUD(F_CPU, BAUD_100K));
19   PORTE.DIRSET    = PIN1_bm|PIN0_bm;            // SDA 0 SCL 1
20   PORTE.PIN0CTRL  = PORT_OPC_WIREDANDPULL_gc;   // Pullup SDA
21   PORTE.PIN1CTRL  = PORT_OPC_WIREDANDPULL_gc;   // Pullup SCL
22   PMIC.CTRL |= PMIC_LOLVLEN_bm;
23
24   init_stream(F_CPU);
25   sei();
26
27   string_to_rtc_time("08:53:38");
28   rtc_set_time(&TWIE);
29   while(1) {
30     clear_screen();
31     rtc_get_time(&TWIE);
32     printf("%s",rtc_time_to_string(t));
33     _delay_ms(1000);
34   }
35 }
```

Het hoofdprogramma uit code 21.13 stelt, na de initialisatie van de I^{2}C-interface en de UART, de tijd van de DS3232 in op 08:53:38. Vervolgens verstuurt het via de UART elke seconde de actuele tijd.

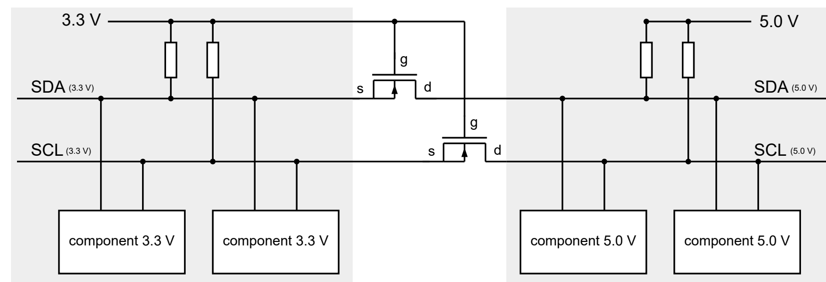


Figuur 21.17 : De organisatie van de bestanden bij het lezen van en schrijven naar de DS3232.

Het hoofdbestand `main.c` maakt gebruik van `stream.c` en `rtc.c` Het bestand `rtc.c` bevat alleen de functies voor het benaderen van de DS3232. 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 DS3232 zit in bestand `rtc.c`. Figuur 21.17 toont de samenhang tussen de diverse bestanden die voor de communicatie met DS3232 nodig zijn.

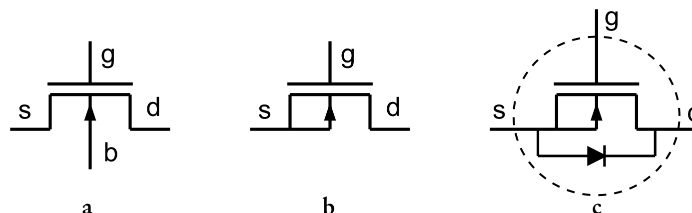
21.9 Levelshifting voor I^{2}C

De Xmega is, zoals al eerder opgemerkt is, niet 5V-tolerant. De klok- en de datalijnen van de Xmega mogen daarom niet direct met een 5V-component verbonden worden. De application note AN97055 van Philips bespreekt dit probleem. Na de verzelfstandiging van NXP is dit document vervangen door de application note AN10441 van NXP. Deze documenten adviseren om de klok- en de datalijn van deze componenten met elkaar te verbinden via een n-kanaal MOSFET.

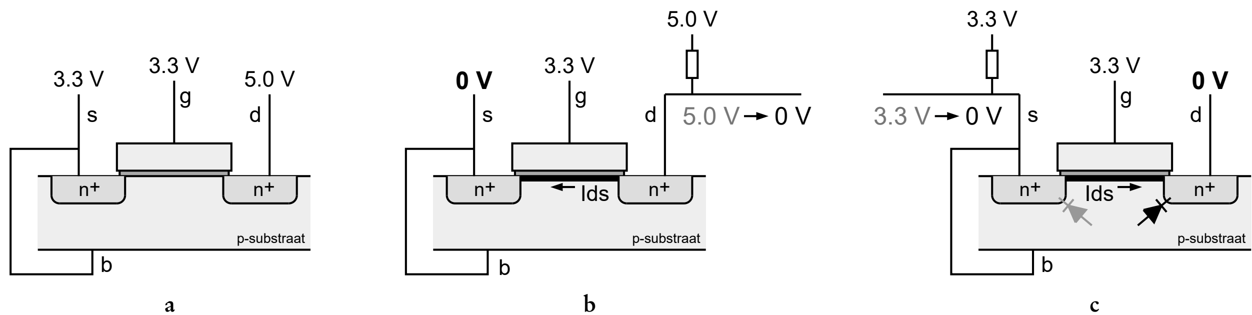


Figuur 21.18 : Bidirectionele levelshifting voor een I^{2}C-systeem met twee verschillende spanningsgebieden. De source en gate van de n-kanaal MOSFET's zijn verbonden met de voeding van de lage spanning en de drain is verbonden met de voeding van de hoge spanning.

Figuur 21.18 geeft een voorbeeld van een component met 3,3 V en een component met 5 V. De drain van de n-kanaal MOSFET's is verbonden met het deel van het I^{2}C-systeem dat een hoge voedingsspanning heeft. De source en de gate zijn verbonden met het deel dat een lage voedingsspanning heeft.



Figuur 21.19 : Symbolen voor een n-kanaal MOSFET. Figuur a geeft het symbool met vier aansluitingen voor de source, gate, drain en back-gate. Figuur b heeft drie aansluitingen; de source en de back-gate zijn intern doorverbonden. Figuur c is het symbool dat NXP en Philips gebruiken. De diode is de pn-overgang tussen de drain en de back-gate.



Figuur 21.20 : **Uitleg werking levelshifting met behulp van een n-kanaal MOSFET.** De gate en source van de transistor zijn verbonden met de lage voedingsspanning. In figuur a staat de situatie als beide lijnen hoog zijn. De U_{gs} is dan nul en is er geen kanaal tussen de source en drain. In figuur b is de zijde met de lage voedingsspanning laag. U_{gs} is dan groter dan de drempelspanning. Er is nu wel een kanaal tussen de source en drain. Er loopt een stroom I_{ds} die de kant met hoge spanning omlaag trekt, ondanks de pullupweerstand. In figuur c is de zijde met de hoge voedingsspanning laag. Aanvankelijk is $U_s (= U_b) > U_d$, zodat de diode tussen drain en back-gate in geleiding komt. U_s neemt af, U_{gs} wordt hoger, zodat er een kanaal ontstaat. Er loopt een stroom van de source naar de drain ($I_{ds} < 0$), die de kant met lage spanning omlaag trekt.

Voor NMOS-transistoren bestaan verschillende symbolen. NXP en Philips gebruiken beide in de datasheets een symbool waarin ook een diode is getekend, zie figuur 21.19. Deze diode is de pn-overgang tussen de drain en de backgate van de n-kanaal MOSFET en ze gebruiken dit symbool omdat deze pn-overgang essentieel is voor de werking van de levelshifting.

Er zijn ook speciale levelshifters voor I²C verkrijgbaar, zoals de PCA9306 van NXP.

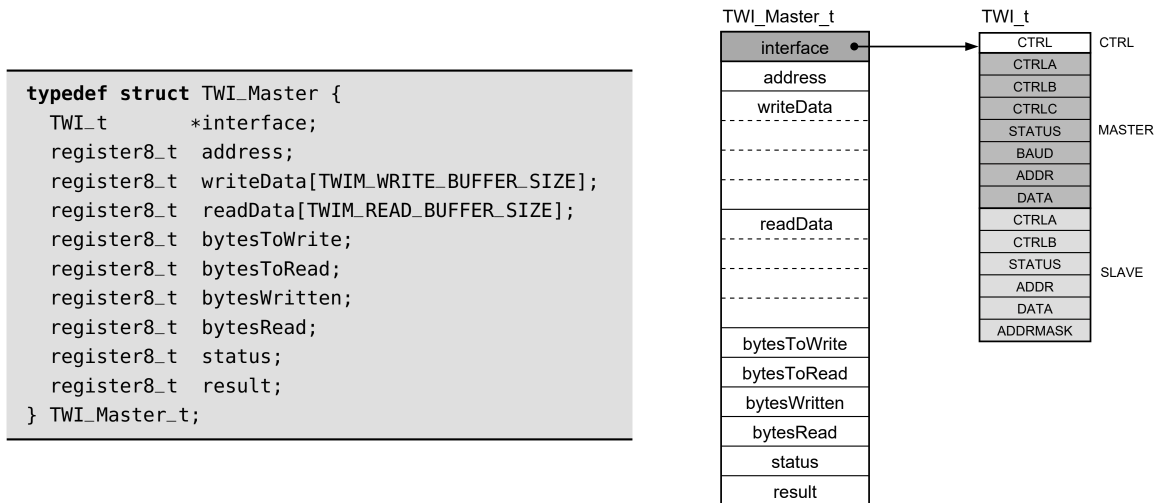
Figuur 21.20 geeft de uitleg van deze levelshifting. Figuur a toont dat, als beide zijden *idle* zijn, de transistor niet geleidt en dat de kant met de lage en met de hoge voedingsspanning van elkaar gescheiden zijn. Figuur b demonstreert dat, als de zijde met de lage voedingsspanning laag is, de zijde met de hoge voedingsspanning eveneens laag wordt. Tenslotte laat figuur c laat zien dat, als de zijde met de hoge voedingsspanning laag is, de pn-overgang tussen de back-gate en de drain ervoor zorgt dat de zijde met de lage voedingsspanning ook laag wordt.

21.10 De TWI-masterdriver van Atmel

De I²C-bibliotheek uit paragraaf 21.6 gebruikt een stijl, die niet zo goed past bij de Xmega. De Xmega kent geen aparte startoperatie. De communicatie wordt gestart door het slave-adres naar het `ADDR`-register te schrijven. Atmel heeft een TWI-driver voor de mastermodus en de slave-modus. Deze paragraaf beschrijft de TWI-driver voor de mastermodus.

De application note AVR1308, *Using the XMEGA TWI*, beschrijft de beide TWI-drivers. Bij dit document hoort een zip-bestand met de drivers.

De TWI-driver van Atmel bestaat uit drie bestanden, namelijk: een algemene bestand `avr_compiler.h` en de bestanden `twi_master_driver.c` en `twi_master_driver.h`. Het c-bestand bevat onder andere de functies `TWI_MasterWrite`, `TWI_MasterRead` en `TWI_MasterWriteRead`, die één of meer bytes kunnen versturen en/of ontvangen. Met name de eerst- en laatstgenoemde functie sluiten goed aan bij de wijze zoals I²C toegepast wordt. De functie `i2c_set_time` uit paragraaf 21.8 verstuurt een pakket met drie bytes en de functie `i2c_get_date` stuurt een adresbyte en ontvangt daarna 3 bytes.



```
typedef struct TWI_Master {
  TWI_t        *interface;
  register8_t  address;
  register8_t  writeData[TWIM_WRITE_BUFFER_SIZE];
  register8_t  readData[TWIM_READ_BUFFER_SIZE];
  register8_t  bytesToWrite;
  register8_t  bytesToRead;
  register8_t  bytesWritten;
  register8_t  bytesRead;
  register8_t  status;
  register8_t  result;
} TWI_Master_t;
```


Figuur 21.21 : De datastructuur `TWI_Master_t`. Links staat de typedefinitie uit `twi_master_driver.h` en rechts een grafische weergave van deze datastructuur.

In het bestand `twi_master_driver.h` staat de typedefinitie van de datastructuur `TWI_Master_t`. Deze datastructuur is weergegeven in figuur 21.21 en bevat twee arrays voor de te versturen en voor de te ontvangen bytes. De grootte van deze arrays hangt af van de macro's `TWIM_WRITE_BUFFER_SIZE` en `TWIM_READ_BUFFER_SIZE`. De velden `bytesToWrite` en `bytesToRead` bevatten het aantal bytes dat geschreven en gelezen moet worden en de velden `bytesWritten` en `bytesRead` het aantal bytes dat inmiddels geschreven en gelezen is. Het `result` bevat de status van TWI-interface en het veld `status` kent twee toestanden: `TWIM_STATUS_READY` en `TWIM_STATUS_BUSY`. De pointer `interface` wijst naar de structuur van de TWI die gebruikt wordt.

Code 21.14 : Deel van de `rtc.c`, die gebaseerd is op de TWI-master-drivers van Atmel.

```
 1 #include "twi_master_driver.h"
 2 #include "rtc.h"   // rtc.h is not code 21.10
 3
 4 struct rtc_time time;
 5 struct rtc_date date;
 6
 7 void rtc_set_time(TWI_Master_t *twim)
 8 {
 9   uint8_t data[4];
10   data[0] = RTC_SECOND;
11   data[1] = time.second;
12   data[2] = time.minute;
13   data[3] = time.hour;
14
15   TWI_MasterWriteRead(twim,
16                  RTC_SLAVE_ADDRESS, data, 4, 0);
17   while (twim->status != TWIM_STATUS_READY);
18 }
```

```
20 void rtc_get_date(TWI_Master_t *twim)
21 {
22   uint8_t data[1];
23   data[0] = RTC_DATE;
24
25   TWI_MasterWriteRead(twim,
26                  RTC_SLAVE_ADDRESS, data, 1, 3);
27   while (twim->status != TWIM_STATUS_READY);
28   date.day   = twim->readData[0];
29   date.month = twim->readData[1];
30   date.year  = twim->readData[2];
31 }
```

De **`while`**-lussen op regel 17 en regel 27 zijn niet noodzakelijk. Deze garanderen echter wel dat de gegevens verwerkt zijn voordat de volgende gegevens verstuurd of ontvangen worden.

In code 21.14 staat een deel van de bibliotheek voor een DS3232, die gebruik maakt van TWI-drivers van Atmel. De functies `rtc_set_time` en `rtc_get_date` komen functioneel overeen met die uit code 21.11. De functie `rtc_set_time` bevat

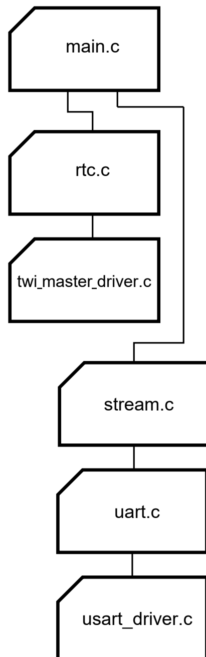
een array `data` die gevuld wordt met het slave-adres en met de seconden, minuten en uren van de tijd die moet worden ingesteld. De functie `TWI_MasterWriteRead` verstuurt deze vier bytes naar de DS3232 waarna er gewacht wordt totdat de informatie is verstuurd.

Bij de functie `rtc_get_date` bevat de array `data` alleen het adres `RTC_DATE` van de datumgegevens. De functie `TWI_MasterWriteRead` ontvangt — nadat dit adres is verstuurd — drie bytes met de dag, de maand en het jaar en plaatst deze bytes in de datastructuur `date`.

Code 21.15 : Programma identiek aan code 21.13 op basis van de TWI-drivers van Atmel.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6  #include <stdio.h>
7  #include "stream.h"                             // or use serialF0.h
8
9  #define  BAUD_100K        100000UL
10 #include "twi_master_driver.h"
11 #include "rtc.h"          // this rtc.h differs from code 21.10, the prototypes are changed
12
13 TWI_Master_t twiMaster;
14
15 int main(void)
16 {
17   char t[] = "HH:MM:SS";
18
19   TWI_MasterInit(&twiMaster, &TWIE, TWI_MASTER_INTLVL_LO_gc, TWI_BAUD(F_CPU, BAUD_100K));
20   PORTE.DIRSET    = PIN1_bm|PIN0_bm;            // SDA 0 SCL 1
21   PORTE.PIN0CTRL  = PORT_OPC_WIREDANDPULL_gc;   // Pullup SDA
22   PORTE.PIN1CTRL  = PORT_OPC_WIREDANDPULL_gc;   // Pullup SCL
23   PMIC.CTRL |= PMIC_LOLVLEN_bm;
24
25   init_stream(F_CPU);
26   sei();
27
28   string_to_rtc_time("08:53:38");
29   rtc_set_time(&twiMaster);
30   while(1) {
31     clear_screen();
32     rtc_get_time(&twiMaster);
33     printf("%s", rtc_time_to_string(t));
34     _delay_ms(1000);
35   }
36 }
37
38 ISR(TWIE_TWIM_vect)
39 {
40   TWI_MasterInterruptHandler(&twiMaster);
41 }
```

Het programma uit code 21.15 stelt met de functies uit het bestand `rtc.c` de tijd in en verstuurt daarna iedere seconde de actuele tijd via de USART0 van poort F. De

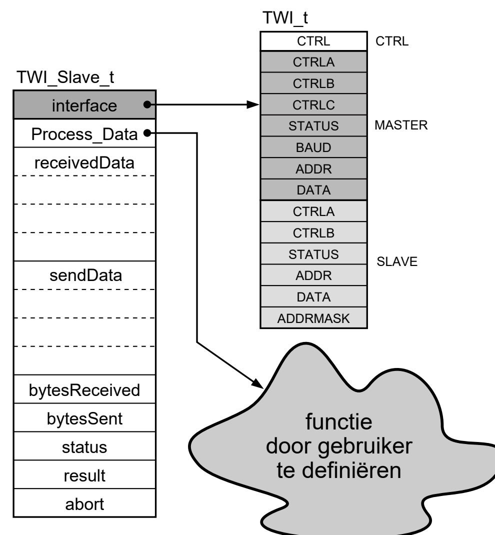


Figuur 21.22 : De bestanden bij code 21.15.

verschillen met code 21.13 zijn klein. Bij de functies `rtc_get_time` en `rtc_set_time` is de eerste parameter nu niet het adres van `TWIE`, maar het adres van de datastructuur `twiMaster` die op regel 13 is gedefinieerd. De functie `TWI_MasterInit` initialiseert op regel 19 de TWI-bus als master.

Het belangrijkste verschil is dat bij deze oplossing interrupts worden gebruikt. Bij initialisatie is het interruptniveau op *low* ingesteld en op regel 38 is de interruptfunctie toegevoegd. Deze functie roept een *handler* aan die de interrupt op de juiste manier afhandelt. Deze *handler* test eerst of de *write interrupt flag* hoog is. Als dat het geval is, wordt er gekeken of alle bytes verstuurd zijn. Dan is het veld `bytesWritten` van de datastructuur gelijk aan het veld `bytesToWrite`. Als niet alle bytes verstuurd zijn, wordt `bytesWritten` opgehoogd en zal de TWI het volgende byte versturen. De *handler* checkt daarna of de *read interrupt flag* hoog is. Dat is het geval als het veld `bytesRead` van de datastructuur gelijk is aan het veld `bytesToRead`. Als ook alle bytes ontvangen zijn, wordt de communicatie afgesloten met de stopcode.

```
typedef struct TWI_Slave {
  TWI_t       *interface;
  void         (*Process_Data) (void);
  register8_t  receivedData[TWIS_RECEIVE_BUFFER_SIZE];
  register8_t  sendData[TWIS_SEND_BUFFER_SIZE];
  register8_t  bytesReceived;
  register8_t  bytesSent;
  register8_t  status;
  register8_t  result;
  bool abort;
} TWI_Slave_t;
```



Figuur 21.23 : **De datastructuur `TWI_Slave_t`. Links staat de typedefinitie uit `twi_slaver_driver.h` en rechts een grafische weergave van deze datastructuur.** Het veld `Process_Data` is een pointer naar een functie die door gebruiker gedefinieerd wordt. Deze functie legt vast hoe de slave gelezen gegevens verwerkt.

21.11 De TWI-slavedriver van Atmel

De bibliotheek van Atmel bevat een TWI-driver om de I^2C-bus te benaderen als slave. In veel gevallen is de microcontroller de master van het systeem en zijn de aangesloten componenten slaaf. Als de microcontroller gebruikt wordt om een slimme sensor te maken, zal deze vaak de slaaf zijn van de centrale microcontroller die het systeem bestuurt. De driver bestaat naast het headerbestand `avr_compiler.h` uit twee bestanden `twi_slave_driver.h` en `twi_slave_driver.c`

Het headerbestand `twi_slave_driver.h` bevat de datastructuur `TWI_Slave_t` en staat in figuur 21.23. De structuur heeft weer een pointer `interface` die naar een TWI-interface wijst en twee arrays `receivedData` en `sendData` voor de te ontvangen en te versturen gegevens.

Het grote verschil met de datastructuur van de master is dat `TWI_Slave_t` een pointer `Process_Data` heeft naar een functie, die de bytes verwerkt die de slaaf ontvangt. Deze functie is alleen nodig bij het ontvangen van gegevens en moet door de gebruiker gedefinieerd worden.

Code 21.16 : De master verstuurt de gegevens.

```
 1 #define F_CPU    2000000UL
 2
 3 #include <avr/io.h>
 4 #include <avr/interrupt.h>
 5 #include <util/delay.h>
 6 #include "twi_master_driver.h"
 7
 8 #define SLAVE_ADDR  0x7C
 9 #define BAUD_100K   100000UL
10 TWI_Master_t twim;
11
12 int main(void)
13 {
14   uint8_t  c = 0;
15   uint8_t  data[2];
16
17   TWI_MasterInit(&twim, &TWIE,
18                  TWI_MASTER_INTLVL_LO_gc,
19                  TWI_BAUD(F_CPU, BAUD_100K));
20   PORTE.DIRSET   = PIN1_bm|PIN0_bm;
21   PORTE.PIN0CTRL = PORT_OPC_WIREDANDPULL_gc;
22   PORTE.PIN1CTRL = PORT_OPC_WIREDANDPULL_gc;
23
24   PMIC.CTRL |= PMIC_LOLVLEN_bm;
25   sei();
26
27   while (1) {
28      data[0] = ':';
29      data[1] = c++;
30      TWI_MasterWrite(&twim, SLAVE_ADDR, data, 2);
31      _delay_ms(1000);
32   }
33 }
34
35 ISR(TWIE_TWIM_vect)
36 {
37   TWI_MasterInterruptHandler(&twim);
38 }
```

Code 21.17 : De slave ontvangt en stuurt de gegevens door.

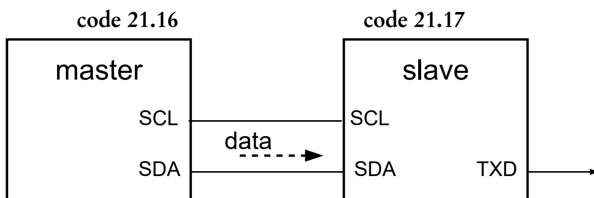
```
 1 #define F_CPU     2000000UL
 2
 3 #include <avr/io.h>
 4 #include <avr/interrupt.h>
 5 #include <util/delay.h>
 6 #include <stdio.h>
 7 #include "stream.h"
 8 #include "twi_slave_driver.h"
 9
10 #define SLAVE_ADDR  0x7C
11 #define BAUD_100K   100000UL
12 TWI_Slave_t twis;
13 volatile uint8_t d[TWIS_RECEIVE_BUFFER_SIZE];
14
15 void SlaveReceiveData(void)
16 {
17   uint8_t index = twis.bytesReceived;
18   d[index]      = twis.receivedData[index];
19 }
20
21 int main(void)
22 {
23   TWI_SlaveInitializeDriver(&twis, &TWIE,
24                                        SlaveReceiveData);
25   TWI_SlaveInitializeModule(&twis, SLAVE_ADDR,
26                                  TWI_SLAVE_INTLVL_LO_gc);
27
28   PMIC.CTRL |= PMIC_LOLVLEN_bm;
29   init_stream(F_CPU);
30   sei();
31
32   while(1) {
33     printf("%c %d\n", (char) d[0], d[1]);
34     _delay_ms(400);
35   }
36 }
37
38 ISR(TWIE_TWIS_vect)
39 {
40   TWI_SlaveInterruptHandler(&twis);
41 }
```

In code 21.17 staat de code van een slaaf, die ongeveer iedere seconde twee bytes ontvangt van de master uit code 21.16. De slaaf ontvangt iedere seconde dus twee bytes en slaat deze op in de array `d` en stuurt iedere 400 ms de actuele waarden door naar USART0 van poort F. Op regel 30 verstuurt de master met de functie `TWI_MasterWrite` de twee bytes. De eerste byte is altijd `':'` en de tweede een 8-bits getal. Het adres van de slaaf `SLAVE_ADDR` is nu een willekeurige waarde. In dit voorbeeld is dat `0x7C`.

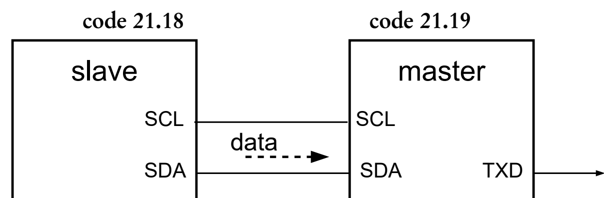
De initialisatie van de slaaf begint op regel 23. De interface voor de driver is `TWIE` en de functie voor het verwerken van de gegevens `SlaveReceiveData`. Het slaafadres van de interface is `SLAVE_ADDR`. Deze waarde moet identiek zijn met die van de master. Bij de master zijn bij de aansluitingen van de TWI-interfaces twee interne pullupweerstanden aangebracht.

De *handler* van de interruptfunctie zorgt ervoor dat de gegevens correct ontvangen worden en gebruikt daarbij de functie `SlaveReceiveData` van regel 15. Nadat het slaafadres ontvangen is, begint de afhandeling. Het aantal ontvangen bytes `twis.bytesReceived` is aanvankelijk nul. De functie `SlaveReceiveData` kent de eerste byte toe aan `d[0]` en de volgende byte `d[1]`. De grootte van de array `d` is niet twee, maar is even groot als de array `receivedData` uit de datastructuur van de slaaf. Dit voorkomt geheugenproblemen als de master meer dan twee bytes verstuurd.

In dit voorbeeld genereert de master de gegevens en geeft deze door aan de slaaf die de gegevens doorstuurt via de UART, zoals dit in figuur 21.24 is getekend. De master gebruikt voor het versturen de functie `TWI_MasterWrite` en de slaaf de functie `SlaveReceiveData` om de gegevens te verwerken.



Figuur 21.24 : De *slave* als ontvanger.



Figuur 21.25 : De *slave* als zender.

In figuur 21.25 is de omgekeerde situatie getekend. De slaaf genereert de gegevens en de master stuurt deze vervolgens door naar de UART. De master heeft het initiatief en gebruikt de functie `TWI_MasterRead` om de gegevens op te halen en de slaaf hoeft de te versturen gegevens alleen in de array `sendData` van de datastructuur `TWI_Slave_t` te zetten.

In code 21.19 staat de beschrijving van de master, die ongeveer eens in de 400 ms met behulp van de functie `TWI_MasterRead` twee bytes van de slaaf ontvangt en deze bytes daarna via de UART doorstuurt. Het programma van de slaaf staat in code 21.18. Deze vult ongeveer iedere seconde twee bytes met nieuwe gegevens.

Code 21.18 : De slave verstuurt de gegevens.

```
1  #define F_CPU    2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6  #include "twi_slave_driver.h"
7
8  #define SLAVE_ADDR  0x7C
9  #define BAUD_100K   100000UL
10 TWI_Slave_t twis;
11
12 int main(void)
13 {
14   uint8_t  c = 0;
15
16   TWI_SlaveInitializeDriver(&twis, &TWIE, NULL);
17   TWI_SlaveInitializeModule(&twis, SLAVE_ADDR,
18                             TWI_SLAVE_INTLVL_LO_gc);
19
20   PMIC.CTRL |= PMIC_LOLVLEN_bm;
21   sei();
22
23   while (1) {
24     twis.sendData[0] = '>';
25     twis.sendData[1] = c++;
26     _delay_ms(1000);
27   }
28 }
29
30 ISR(TWIE_TWIS_vect)
31 {
32     TWI_SlaveInterruptHandler(&twis);
33 }
```

Code 21.19 : De master ontvangt en stuurt de data door.

```
1  #define F_CPU     2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include <util/delay.h>
6  #include <stdio.h>
7  #include "stream.h"
8  #include "twi_master_driver.h"
9
10 #define SLAVE_ADDR  0x7C
11 #define BAUD_100K   100000UL
12 TWI_Master_t twim;
13
14 int main(void)
15 {
16   uint8_t d[TWIM_READ_BUFFER_SIZE];
17
18   TWI_MasterInit(&twim, &TWIE,
19                  TWI_MASTER_INTLVL_LO_gc,
20                  TWI_BAUD(F_CPU, BAUD_100K));
21   PORTE.DIRSET   = PIN1_bm|PIN0_bm;
22   PORTE.PIN0CTRL = PORT_OPC_WIREDANDPULL_gc;
23   PORTE.PIN1CTRL = PORT_OPC_WIREDANDPULL_gc;
24
25   PMIC.CTRL |= PMIC_LOLVLEN_bm;
26   init_stream(F_CPU);
27   sei();
28
29   while(1) {
30     TWI_MasterRead(&twim, SLAVE_ADDR, 2);
31     while (twim.status != TWIM_STATUS_READY);
32     d[0] = twim.readData[0];
33     d[1] = twim.readData[1];
34
35     printf("%c %d\n", (char) d[0], d[1]);
36     _delay_ms(400);
37   }
38 }
39
40 ISR(TWIE_TWIM_vect)
41 {
42     TWI_MasterInterruptHandler(&twim);
43 }
```

De slaaf stuurt in dit voorbeeld alleen informatie en ontvangt geen gegevens. Er is geen gebruikersfunctie nodig die de ontvangen gegevens verwerkt. De derde parameter van de functie `TWI_SlaveInitializeDriver` is daarom `NULL`. De slaaf vult in de oneindige **`while`** op regel 23 de array `sendData` met informatie en wacht daarna 1000 ms. De *handler* van de interruptfunctie handelt automatisch het verzoek van de master af.

Op regel 30 ontvangt de master met de functie `TWI_MasterRead` de twee bytes van de slaaf, wacht daarna totdat de gegevens ontvangen zijn, kent het resultaat toe aan array `d` en stuurt deze vervolgens door naar de UART.

21.12 Resumé TWI

Er zijn twee TWI-bibliotheken besproken: de eenvoudige I^2C-bibliotheek uit paragraaf 21.7 en de TWI-drivers van Atmel. De eenvoudige TWI-bibliotheek is speciaal voor dit boek geschreven om te laten zien dat het I^2C-protocol met zes basisfuncties kan worden opgebouwd. In paragraaf 21.8 is deze bibliotheek gebruikt voor de communicatie met een DS3232. Omdat de foutafhandeling bij de TWI-drivers van Atmel beter is en omdat er gebruik gemaakt wordt van interrupts, is de oplossing uit paragraaf 21.10 voor de Xmega een beter alternatief.

Gebruik bij componenten met een andere voedingsspanning de methode voor levelshifting uit paragraaf 21.9.

Slimme sensoren zullen vaak als slaaf met een master moeten communiceren. De voorbeelden uit paragraaf 21.11 gebruiken een slaaf die of alleen gegevens verstuurt of alleen gegevens ontvangt. De bijbehorende master gebruikt daarbij respectievelijk `TWI_MasterRead` en `TWI_MasterWrite`. In het algemeen zal een slimme sensor zowel data ontvangen als versturen. De gebruikersfunctie zal afhankelijk van de ontvangen gegevens andere informatie terugsturen. De master kan dan met de `TWI_MasterWriteRead` de gewenste informatie opvragen.

22 Pulsbreedtemodulatie

Doelstelling	Dit hoofdstuk behandelt de pulsbreedtemodulatie. Het is een aanvulling op paragraaf 16.4 over de timer/counters. De mogelijkheden om met behulp van de timer/counters van de Xmega een pulsbreedtegemoduleerd signaal te maken, worden besproken.

Onderwerpen

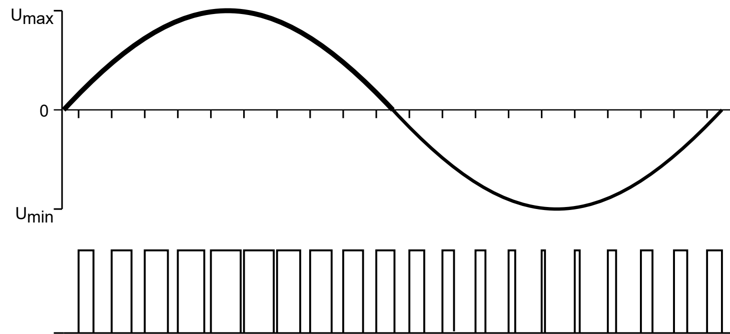
De behandelde onderwerpen zijn:

- Het principe van pulsbreedtemodulatie, PWM (*Pulse Width Modulation*).
- De opbouw van uitgangsblokken bij de timer/counter van de Xmega.
- De verschillende soorten technieken om met de timers een PWM-signaal te maken.
- De normale, de frequentie-, de single-slope- en de dual-slope-modus.
- De waveformgeneratoren van de timers met de mogelijkheden voor de uitgangssignalen.
- De aansturing van een 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 de single-slope-modus.
- Het regelen van de led-intensiteit bij een rgb-led met de single-slope-modus.
- Het aansturen van DC-motoren bij een robotwagen met de dual-slope-modus.
- Het aansturen van servomotoren bij robots met de dual-slope-modus.
- Het afspelen van muziek met behulp van de frequentiemodus.

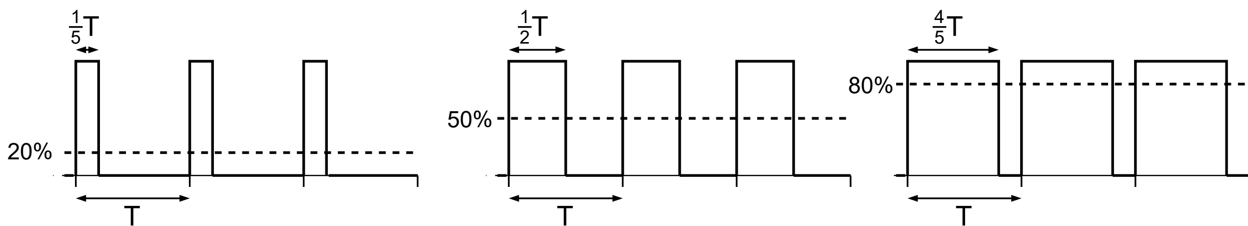
Pulsbreedtemodulatie is een modulatietechniek waarbij de informatie wordt vastgelegd in de breedte van de puls van een pulsvormig signaal. Het signaal kent een hoog en een laag signaalniveau en de frequentie ligt vast. In figuur 22.1 komt de breedte van de puls overeen met de grootte van het sinusvormige 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 3,3 V is het gemiddelde signaalniveau 0,66 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 breedte van de puls T_{pw} gedeeld door de periodetijd T:

$$\Delta = \frac{T_{pw}}{T} \tag{22.1}$$

De *duty cycle* is de relatieve pulsduur. In het Nederlands wordt dit ook duty-cycle genoemd.

In principe is de *duty cycle* Δ een getal tussen 0 en 1, maar meestal wordt dit uitgedrukt in procenten. Een signaal met een periodetijd van 100 μs en een pulsbreedte van 25 μs heeft een duty-cycle van 0,25 of 25%.
Als de uitgang een PWM-signaal is met een duty-cycle van 80%, zal een led — mits deze zo is aangesloten dat de microcontroller de stroom levert — 80% van de tijd aan zijn. Bij een pulsduur van 20% is de led slechts 20% van de tijd aan en brandt de led duidelijk minder fel.

22.1 De timer/counters van de Xmega

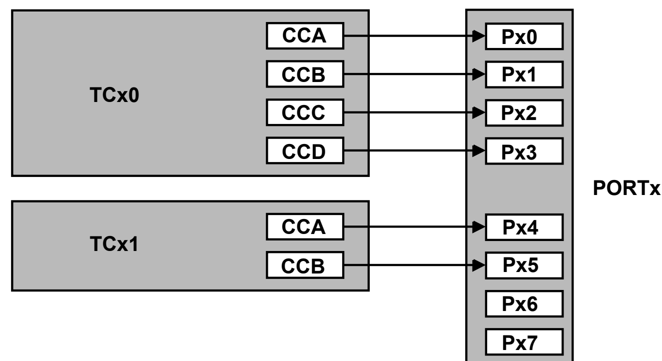
Tijd speelt bij embedded systemen een belangrijke rol. Elke microcontroller heeft daarom een of meer tellers. Tijd wordt bepaald door klokslagen te tellen. Deze tellers worden daarom ook timers genoemd. De microcontroller gebruikt de tellers om:

- gebeurtenissen te tellen;
- een tijdsduur te definiëren;
- een periodetijd of een pulsduur te meten;
- een PWM-signaal te genereren;
- een frequentiegenerator te maken.

De Xmega heeft bij iedere digitale poort één of twee 16-bits timer/counters. De Xmega256a3u heeft vier digitale poorten. De poorten C, D en E hebben een timer/counter 0 en een timer/counter 1 en poort F heeft alleen een timer/counter 0. Deze timer/counters kennen de volgende instelmogelijkheden:

- de normale modus,
- de frequentiemodus,
- *single slope* PWM,
- *dual slope* PWM.

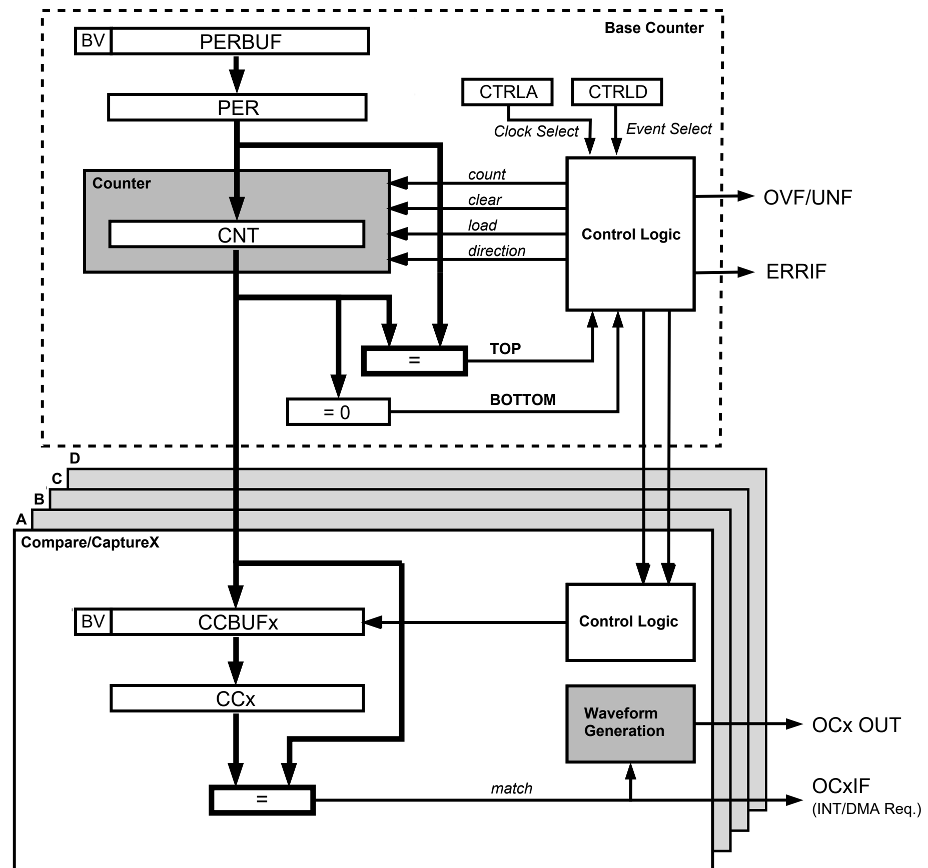
Daarnaast is er ook een *input capture*-modus voor het meten van periodetijden en pulsbreedten. De normale modus wordt gebruikt om gebeurtenissen te tellen en om tijdsduur te definiëren of te meten. De frequentiemodus is bedoeld voor het maken van een blokgolf met een specifieke frequentie en een duty-cycle van 50%. De twee PWM-modi genereren een PWM-signaal met een vaste frequentie en een specifieke pulsduur.



Figuur 22.3 : De organisatie van de timer/counters bij een poort. De PWM-uitgangen CCA, CCB, CCC en CCD van timer/counter 0 zijn verbonden met de pinnen 0 tot en met 3. De PWM-uitgangen CCA en CCB van timer/counter 1 zijn verbonden met pin 4 en 5.

De werking van de timer/counter is voor een belangrijk deel al in hoofdstuk 16.4 besproken. In figuur 16.6 staat het basisdeel met de teller. Figuur 22.4 geeft het complete blokschema, zoals dat in de datasheet getekend is. Het uitgangsdeel voor de generatie van PWM-signalen bestaat bij timer/counter 0 uit vier *compare/capture*-blokken en bij timer/counter 1 uit twee *compare/capture*-blokken. Figuur 22.3 laat zien dat de vier blokken van timer/counter 0 verbonden zijn met pin 0 tot en met 3 van de betreffende poort en dat de blokken van timer/coun-

ter 1 verbonden zijn met pin 4 en 5. Met timer/counter 0 kunnen vier afhankelijke PWM-signalen worden gemaakt en met timer/counter 1 twee afhankelijke PWM-signalen.



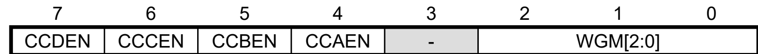
Figuur 22.4 : Het blokschema van de timer/counters van de Xmega. Het deel voor de generatie van PWM-signalen staat aan de onderkant en bestaat uit twee of vier gelijke *compare/capture*-blokken met ieder een eigen waveformgenerator.

De compare/capture-blokken van de timer/counters worden in figuur 22.3 aangeduid met CCA, CCB, CCC en CCD. Ieder blok heeft een register met dezelfde naam. In figuur 22.4 is één blok uitgewerkt en heet dit register CCx. De gebruiker stelt de waarde van CCx in. Deze waarde wordt voortdurend vergeleken met de waarde van de teller CNT. Het resultaat gebruikt de waveformgenerator om het PWM-signaal te maken.

De registers PER en CCx zijn dubbel gebufferd met de registers PERBUF en CCBUFx. Bij deze registers hoort een *buffer valid*-bit die aangeeft dat het register een nieuwe waarde bevat die gekopieerd moet worden naar het overeenkomstige PER- of CCx-register. De dubbele buffering wordt gebruikt in de inputcapturemodus en bij de generatie van PWM-signalen. Het zorgt ervoor dat duty-cycles niet kunnen veranderen middenin een PWM-periode en om PWM-signalen te synchroniseren.

22.2 Bespreking PWM-mogelijkheden

De modus van de timer/counter wordt ingesteld met de drie WGM-bits van register CTRLB, dat in figuur 22.5 is afgebeeld. Deze gekozen instelling geldt voor de timer/counter. Alle uitgangsblokken hebben dus dezelfde modus, maar kunnen wel afzonderlijk van elkaar worden aan- of uitgezet met CCENx-bits uit ditzelfde register.



Figuur 22.5 : Het register CTRLB voor het instelling van de PWM.

Voor de instelling van de modus worden drie bits gebruikt, omdat er drie verschillende versies van de dual-slope-modus zijn. Er zijn zodoende zes verschillende mogelijkheden te onderscheiden De typedefinitie uit avr/io.h bevat tien groepsconfiguraties:

```
typedef enum TC_WGMODE_enum
{
  TC_WGMODE_NORMAL_gc      = (0x00<<0),  // normal mode
  TC_WGMODE_FRQ_gc         = (0x01<<0),  // frequency mode
  TC_WGMODE_SINGLESLOPE_gc = (0x03<<0),  // single slope
  TC_WGMODE_SS_gc          = (0x03<<0),  // single slope
  TC_WGMODE_DSTOP_gc       = (0x05<<0),  // dual slope, update on top
  TC_WGMODE_DS_T_gc        = (0x05<<0),  // dual slope, update on top
  TC_WGMODE_DSBOTH_gc      = (0x06<<0),  // dual slope, update on both
  TC_WGMODE_DS_TB_gc       = (0x06<<0),  // dual slope, update on both
  TC_WGMODE_DSBOTTOM_gc    = (0x07<<0),  // dual slope, update on bottom
  TC_WGMODE_DS_B_gc        = (0x07<<0)   // dual slope, update on bottom
} TC_WGMODE_t;
```

Sommige groepsconfiguraties hebben twee namen gekregen; TC_WGMODE_SS_gc en TC_WGMODE_SINGLESLOPE_gc zijn bijvoorbeeld identiek. In de normale modus worden de waveformgeneratoren van de uitgangsblokken niet gebruikt.

De normale modus

Bij de bespreking van de interrupts is in paragraaf 16.5 de timer/counter 0 van poort E gebruikt om een uitgang hoog en laag te maken. In feite is daarbij een PWM-signaal gecreëerd met een duty-cycle van 50%. Iedere keer als de waarde van de timer gelijk is aan PER wordt de bijbehorende interruptfunctie aangeroepen en klapt de uitgang om:

```
ISR(TCE0_OVF_vect)
{
  PORTC.OUTTGL = PIN0_bm;
}
```

Met de timer/counter 0 van poort C in de PWM- of frequentiemodus kan deze uitgang ook een PWM-signaal genereren met een duty-cycle van 50%. Dat is in code 16.4 al gedaan met behulp van de frequentiemodus.

Ondanks dat bij de normale modus de waveformgeneratoren niet in gebruik zijn, kunnen met deze modus ook PWM-signalen worden gemaakt. Dit kan soms interessant zijn, bijvoorbeeld om een PWM-signaal te maken bij een aansluiting

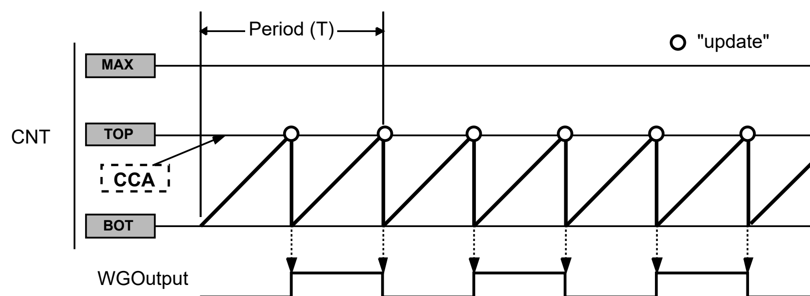
van een poort zonder timer/counter of als er meer dan vier afhankelijke PWM-signalen nodig zijn. Onderstaand voorbeeld geeft de interruptfunctie, die voor aansluiting 3 van poort A een PWM-signaal maakt met een instelbare duty-cycle:

```
volatile uint16_t period = 62500;
volatile uint8_t  duty_cycle;    // value between 0 and 100

ISR(TCE0_OVF_vect)
{
  if ( bit_is_clear(PORTA.IN, PIN3_bm) ) {
    PORTA.OUTSET = PIN3_bm;
    TCC0.PER = period/100*duty_cycle;
  } else {
    PORTA.OUTCLR = PIN3_bm;
    TCC0.PER = period/100*(100-duty_cycle);
  }
}
```

Als bij een interrupt de uitgang laag is, wordt de uitgang hoog gemaakt en krijgt PER een waarde, die overeenkomt met het aantal klokslagen dat het signaal hoog moet zijn. Als de uitgang hoog is, wordt de uitgang juist laag gemaakt en krijgt PER een waarde die overeenkomt met het aantal klokslagen dat het signaal laag moet zijn. Voor een duty-cycle van 10% — de variabele duty_cycle moet dan gelijk aan 10 zijn — zal de uitgang 6250 gedeelde klokslagen hoog en 56250 gedeelde klokslagen laag zijn.

Frequentiemodus

In de frequentiemodus genereert de uitgang een signaal met een duty-cycle van 50%. Deze modus is vooral bedoeld voor situaties waarbij de frequentie geregeld moet worden. Een voorbeeld is het genereren van een toon voor het maken van muziek.



Figuur 22.6 : De waarde van CNT en de waveform van de uitgang bij de frequentiemodus.

De frequentiemodus lijkt op de normale modus en gebruikt in plaats van het register PER het register CCA om de TOP-waarde in te stellen. Als het uitgangsblok CCA geselecteerd is en bovendien de overeenkomstige pin een uitgang is, verschijnt op de uitgang een PWM-signaal met een duty-cycle van 50%. In code 16.4 staat een voorbeeld voor aansluiting PC0 met de timer/counter 0 van poort C en in figuur 22.6 staat het verloop van de waarde van CNT en het uitgangssignaal.

Als er een andere aansluiting nodig is, dan die bij CCA hoort, is het onvoldoende om alleen de CCA in CCx te veranderen. De topwaarde van de timer/counter wordt immers ingesteld met CCA. Met de andere CCx-registers kan geen topwaarde worden ingesteld. Wel kunnen de waveformgeneratoren van de blokken B, C en D actief zijn en gelijktijdig een uitgangssignaal met dezelfde frequentie en dezelfde duty-cycle genereren.

Code 22.1 : PWM-signaal voor aansluiting PD3 met behulp van de frequentiemodus.

```
 1 #include <avr/io.h>
 2
 3 int main(void)
 4 {
 5     PORTD.DIRSET  = PIN3_bm;                           // PD3 output
 6
 7     TCD0.CTRLB    = TC0_CCDEN_bm | TC_WGMODE_FRQ_gc;    // CCD enabled
 8     TCD0.CTRLA    = TC_CLKSEL_DIV8_gc;
 9     TCD0.CCA      = 62499;            // CCA is necessary for update
10     TCD0.CCD      = 62499;            // CCD in phase with CCA
11
12     while (1) {
13         asm volatile ("nop");         // do nothing
14     }
15 }
```

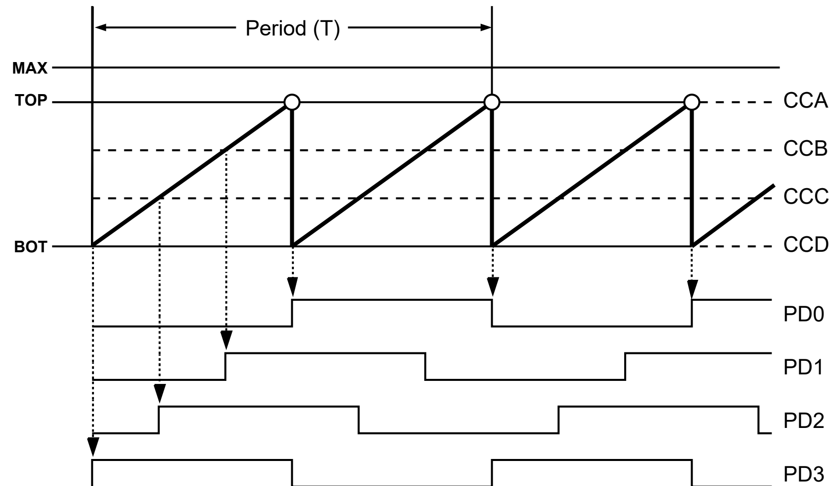
Code 22.1 gebruikt de timer/counter 0 van poort D om met de frequentiemodus een PWM-signaal te genereren bij aansluiting PD3. Op regel 5 is aansluiting PD3 uitgang gemaakt en op regel 7 is uitgangsblok CCD aangezet. Op regel 9 krijgt register CCA de gewenste topwaarde. De waarde van CCA bepaalt de frequentie van het uitgangssignaal. De waarde van CCD moet kleiner of gelijk zijn aan CCA. Als CCA en CCD gelijk zijn, zijn de waveformgeneratoren in fase. Anders is er een faseverschil. In code 22.1 is voor CCD iedere waarde van 0 tot en met 62499 geschikt.

Code 22.2 : De generatie van vier PWM-signalen met een verschillende fase.

```
 1 #include <avr/io.h>
 2
 3 int main(void)
 4 {
 5     PORTD.DIRSET  = PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
 6
 7     TCD0.CTRLB    = TC0_CCDEN_bm|TC0_CCCEN_bm|TC0_CCBEN_bm|TC0_CCAEN_bm|
 8                     TC_WGMODE_FRQ_gc;
 9     TCD0.CTRLA    = TC_CLKSEL_DIV8_gc;
10     TCD0.CCA      = 60000;
11     TCD0.CCB      = 40000;      // 16.6% ahead
12     TCD0.CCC      = 20000;      // 33.3% ahead
13     TCD0.CCD      = 0;          // 50%   ahead
14
15     while (1) {}                // do nothing
16 }
```

Code 22.2 is een variant van code 22.1 met vier PWM-uitgangen. De pinnen PD3, PD2, PD1 en PD0 zijn alle vier uitgang en de bijbehorende uitgangsblokken van

timer/counter 0 zijn alle vier actief. De registers CCx hebben alle vier een andere waarde. CCA is het grootst, CCB is 2/3 van CCA, CCC is 1/3 van CCA en CCD is nul. Het effect is te zien in figuur 22.7. Uitgang PD3 klapt direct om als de teller nog nul is. PD2 en PD1 klappen respectievelijk bij 1/6 en 2/6 van de periodetijd om. Uitgang PD0 klapt als laatste om. De andere drie signalen lopen zodoende voor op uitgang PD0.



Figuur 22.7 : De vier uitgangssignalen bij code 22.2. De vette lijn geeft de waarde van de teller CNT. Als deze gelijk is aan CCx klapt de betreffende uitgang om. Als CNT gelijk is aan TOP en dus gelijk is aan CCA, begint CNT weer bij nul.

Voor de frequentiemodus hangt de frequentie f_{frq} van het PWM-signaal af van de frequentie f_{clk} van de systeemklok, de prescaling N en de waarde van register CCA:

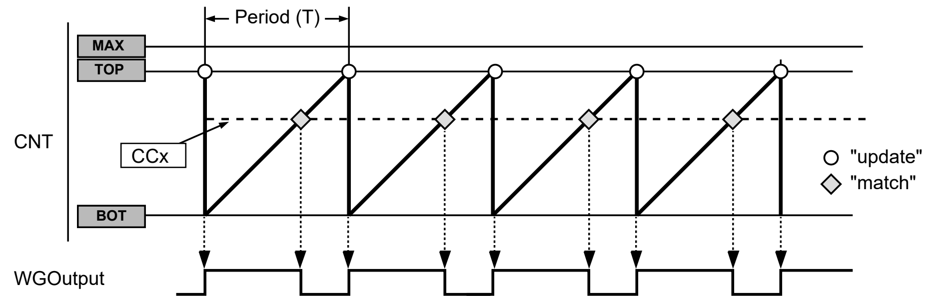
$$f_{\text{frq}} = \frac{f_{\text{clk}}}{2\,N\ (\texttt{CCA}+1)} \tag{22.2}$$

In het voorbeeld van code 22.1 is de prescaling 8 en is CCA gelijk aan 62499. Voor een systeemklok van 2 MHz is de frequentie van de uitgang dan 2 Hz.

Single Slope PWM

In de single-slope-modus genereren de uitgangen PWM-signalen met een instelbare duty-cycle. Dit is een echte PWM-modus, die bij andere microcontrollers vaak wordt aangeduid als fast-PWM en die bedoeld is voor het aansturen van allerlei systemen met behulp van PWM-signalen, zoals bijvoorbeeld de intensiteitsregeling van een rgb-led.

De single-slope-modus gebruikt evenals de normale modus het register PER om de TOP-waarde in te stellen. De waarde van CNT wordt tijdens het tellen voortdurend vergeleken met de registers CCx van de actieve uitgangsblokken. Als deze waarden gelijk zijn, gaat het uitgangssignaal van hoog naar laag. De teller telt door totdat de waarde gelijk is aan PER, dan wordt de teller op nul gezet en gaat de uitgang van laag naar hoog. In figuur 22.8 is een voorbeeld voor een willekeurig uitgangsblok getekend.



Figuur 22.8 : De waarde van CNT en de waveform van de uitgang bij de single-slope-modus.

De frequentie f_{ss} van het uitgangssignaal is bij de single-slope-modus afhankelijk van de frequentie f_{clk} van de systeemklok, de prescaling N en de waarde van het register PER:

$$f_{ss} = \frac{f_{clk}}{N\ (\texttt{PER} + 1)} \tag{22.3}$$

Deze frequentie is voor alle vier de uitgangen hetzelfde.

De duty-cycle Δ_{ss} van het uitgangssignaal hangt af van de waarde van de registers PER en CCx:

$$\Delta_{ss} = \frac{\texttt{CCx}}{\texttt{PER} + 1} \tag{22.4}$$

Uit formule 22.4 volgt, dat als CCx gelijk is aan PER, de duty-cycle niet helemaal 100% is. De duty-cycle is 100% voor iedere CCx die groter is dan PER. De duty-cycle is 0% als CCx gelijk is aan nul.

Code 22.3 : Een PWM-signaal met een duty-cycle van 80% met de single-slope-modus.

```
1  #include <avr/io.h>
2
3  int main(void)
4  {
5    PORTD.DIRSET = PIN0_bm;
6
7    TCD0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
8    TCD0.CTRLA   = TC_CLKSEL_DIV4_gc;
9    TCD0.PER     = 9999;
10   TCD0.CCA     = 8000;
11
12   while (1) {}      // do nothing
13 }
```

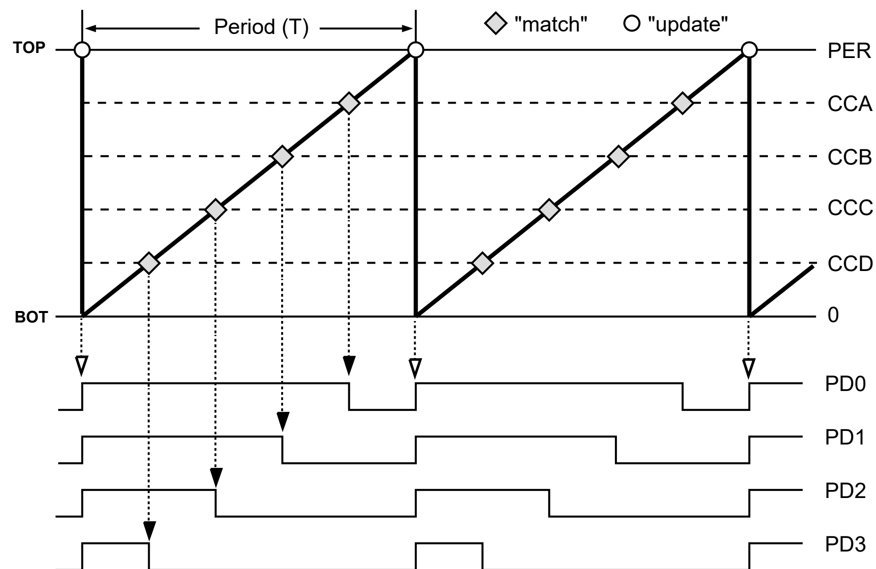
Code 22.3 gebruikt timer/counter 0 van poort D om met behulp van de single-slope-modus voor aansluiting PD0 een PWM-signaal te maken met een duty-cycle van 80%. De frequentie van het signaal volgt uit formule 22.3 en is voor een systeemklok van 2 MHz, een prescaling 4 en met PER 9999 gelijk aan 50 Hz.

Code 22.4 gebruikt alle uitgangsblokken van timer/counter 0 om vier PWM-signalen te maken met een duty-cycle van respectievelijk 80, 60, 40 en 20%.

Code 22.4 : Vier PWM-signalen met behulp van de single-slope-modus.

```
1  #include <avr/io.h>
2
3  int main(void)
4  {
5    PORTD.DIRSET = PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
6
7    TCD0.CTRLB   = TC0_CCDEN_bm|TC0_CCCEN_bm|TC0_CCBEN_bm|TC0_CCAEN_bm|
8                   TC_WGMODE_SINGLESLOPE_gc;
9    TCD0.CTRLA   = TC_CLKSEL_DIV4_gc;
10   TCD0.PER     = 9999;
11   TCD0.CCA     = 8000;
12   TCD0.CCB     = 6000;
13   TCD0.CCC     = 4000;
14   TCD0.CCD     = 2000;
15
16   while (1) {}      // do nothing
17 }
```

De vier PWM-signalen hebben alle vier dezelfde frequentie en worden alle vier tegelijkertijd hoog, zoals figuur 22.9 laat zien.



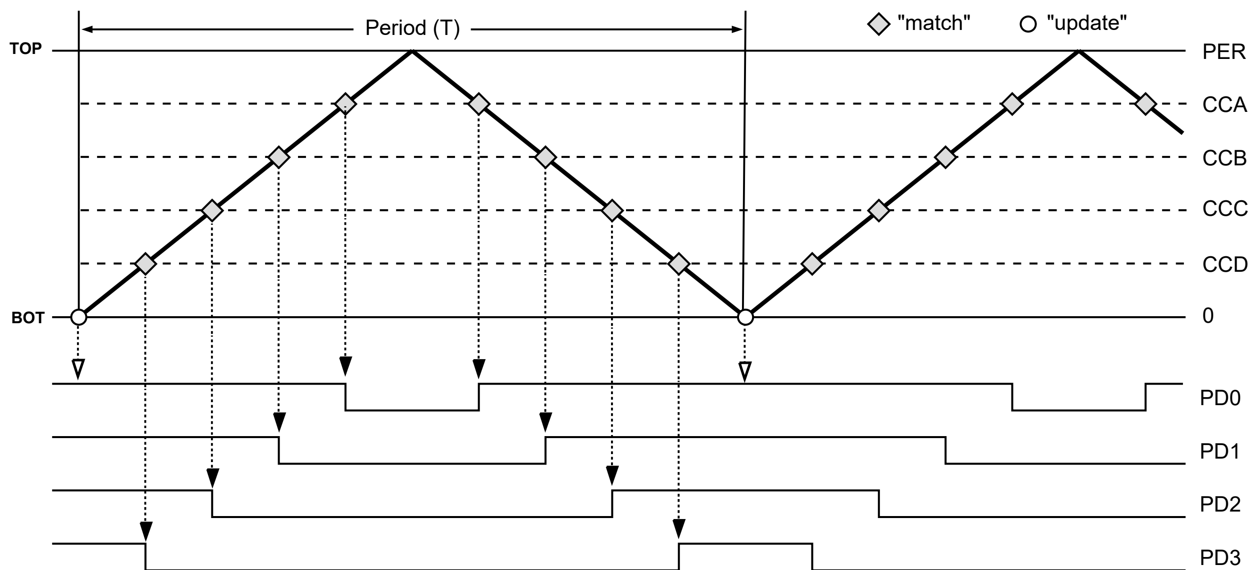
Figuur 22.9 : Vier PWM-signalen met de single-slope-modus. De signalen worden tegelijkertijd hoog.

Deze PWM-methode wordt ook wel fast-PWM genoemd omdat één cyclus van de teller overeenkomt met één periode van de PWM-signalen. Bij de dual-slope-modus zijn voor één periode van het PWM-signaal twee cycli nodig.

Dual Slope PWM

Een nadeel van de single-slope-modus is dat bij het updaten van de uitgangen alle PWM-signalen gelijktijdig hoog worden. Als deze signalen gebruikt worden om bijvoorbeeld de spoelen van een motor aan te sturen, gaan alle uitgangen tegelijkertijd aan. De aanstuurschakeling van de spoelen gaat plotseling veel stroom gebruiken. In veel situaties is het beter om de uitgangen niet gelijktijdig aan of uit te zetten. Bij de dual-slope-modus veranderen de uitgangen nooit tegelijkertijd.

In de dual-slope-modus telt de teller van de timer/counter steeds op totdat de maximale waarde bereikt is en telt dan weer terug naar de beginwaarde. In figuur 22.10 staat een voorbeeld met vier uitgangssignalen. Deze signalen worden hoog bij de *update*, laag als er een *match* is bij het omhoog tellen en weer hoog als er een *match* is bij het naar beneden tellen.



Figuur 22.10 : Vier PWM-signalen met de dual-slope-modus. De vier uitgangssignalen veranderen nooit op hetzelfde moment.

De frequentie f_{ds} van de uitgangssignalen hangt bij de dual-slope-modus af van de frequentie f_{clk} van de systeemklok, de prescaling N en de waarde van het register PER:

$$f_{ds} = \frac{f_{clk}}{2\ N\ \texttt{PER}} \tag{22.5}$$

Deze frequentie is weer voor alle vier de uitgangen hetzelfde. De duty-cycle Δ_{ds} hangt ook weer af van de waarde van de registers PER en CCx:

$$\Delta_{ds} = \frac{\texttt{CCx}}{\texttt{PER}} \tag{22.6}$$

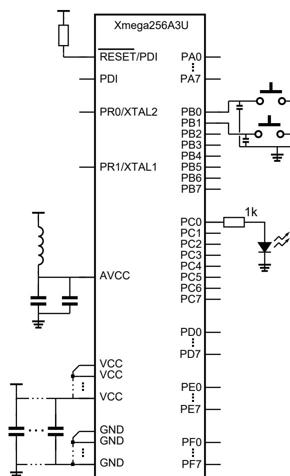
Het programma waarmee de PWM-signalen van figuur 22.10 worden gemaakt, staat in code 22.5 en verschilt op twee punten van code 22.4. Op regel 8 is de modus aangepast en op regel 10 krijgt PER de waarde 10000 in plaats van 9999. Voor de frequentie van de PWM-signalen geldt nu formule 22.5. Bij een systeemklok van 2 MHz is de frequentie 25 Hz en precies twee keer zo laag als bij de single-slope-modus. De duty-cycle is voor de gekozen waarden van CCx bij alle vier signalen weer respectievelijk 80, 60, 40 en 20%.

Code 22.5 : Vier PWM-signalen met behulp van de dual-slope-modus.

```
1  #include <avr/io.h>
2
3  int main(void)
4  {
5    PORTD.DIRSET = PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
6
7    TCD0.CTRLB   = TC0_CCDEN_bm|TC0_CCCEN_bm|TC0_CCBEN_bm|TC0_CCAEN_bm|
8                   TC_WGMODE_DSBOTH_gc;
9    TCD0.CTRLA   = TC_CLKSEL_DIV4_gc;
10   TCD0.PER     = 10000;
11   TCD0.CCA     = 8000;
12   TCD0.CCB     = 6000;
13   TCD0.CCC     = 4000;
14   TCD0.CCD     = 2000;
15
16   while (1) {}      // do nothing
17 }
```

22.3 De single-slope-modus: intensiteitsregeling voor een led

Een typisch voorbeeld voor een toepassing van de single-slope-modus 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.



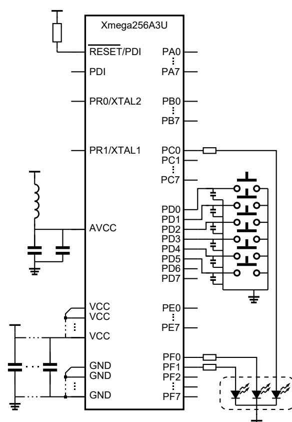
Figuur 22.11 : **Schema voor intensiteitsregeling led met twee drukknoppen.**

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 22.6 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 bepaald met `CCA`. De elf waarden voor `CCA` staan op regel 2 in een array `level`. Het oog ziet verschillen in 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 15 `CCA` ingesteld op de middelste waarde uit de reeks. Regel 14 stelt `PER` in op de maximale waarde uit de array.

In de oneindige lus wordt getest of een van de ingangen `0` en 1 van poort B laag is. Als ingang `0` laag is, wordt de index `i` opgehoogd, krijgt `CCA` een hogere waarde en zal de led feller branden. Als ingang 1 laag is, wordt de index `i` verlaagd, krijgt `CCA` een lagere waarde en zal de led minder fel gaan branden.

Code 22.6 : Regeling voor lichtintensiteit van led op basis van fast-PWM.

```
1  #include <avr/io.h>
2  uint16_t level[] = {34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181};
3
4  int main(void) {
5    uint8_t  i = sizeof(level)/sizeof(uint16_t)/2;
6
7    PORTB.DIRCLR   = PIN1_bm|PIN0_bm;
8    PORTB.PIN0CTRL = PORT_OPC_PULLUP_gc;                  // enable pull up PB0
9    PORTB.PIN1CTRL = PORT_OPC_PULLUP_gc;                  // enable pull up PB1
10
11   PORTC.DIRSET   = PIN0_bm;                             // PC0 (CCA) output
12   TCC0.CTRLB = TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
13   TCC0.CTRLA = TC_CLKSEL_DIV1_gc;
14   TCC0.PER   = level[sizeof(level)/sizeof(uint16_t) - 1];
15   TCC0.CCA   = level[i];                                // init fading level
16
17   while (1) {
18     if (bit_is_clear(PORTB.IN, 0)){                     // increase duty cycle
19       if (i < 10) i++;
20       TCC0.CCABUF = level[i];
21       loop_until_bit_is_set(PORTB.IN, 0);
22     }
23     if(bit_is_clear(PORTB.IN, 1)) {                     // decrease duty cycle
24       if (i > 0) i--;
25       TCC0.CCABUF = level[i];
26       loop_until_bit_is_set(PORTB.IN, 1);
27     }
28   }
29 }
```



Figuur 22.12 : Schema voor aansturen rgb-led. De rgb-led is aangesloten op PC0, PF0 en PF1. De drie leds hebben een gemeenschappelijke kathode en branden feller bij een grotere duty-cycle.

De frequentie van het PWM-signaal is bij een systeemklok van 2 MHz ruim 478 Hz. Deze frequentie is zo snel dat het aan- en uitgaan niet zichtbaar is. Bij hoge waarden voor `CCA` brandt de led fel en bij lage waarden gloeit de led.

22.4 De single-slope-modus: intensiteitsregeling voor een rgb-led

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 22.12, is er een gemeenschappelijke kathode. De drie PWM-signalen worden met de timer/counter `0` van poort C en de timer/counter `0` van poort F gemaakt. De intensiteit van de leds is instelbaar met de zes drukknoppen.

Het instellen van de leds kan op dezelfde manier gedaan worden als in code 22.6. Omdat er bij het dimmen van drie leds in feite zes bijna identieke handelingen nodig zijn, is er een algemene functie `fade` gemaakt. Het bestand `fade.c` met deze functie staat in code 22.8. Het bijbehorende headerbestand `fade.h` staat in code 22.7 en bevat het prototype van deze functie en de definities van de dimrichtingen `UP` en `DOWN`.

Code 22.7 : Het headerbestand `fade.h`.

```
1  #define UP              1
2  #define DOWN            0
3
4  void fade(int dir, int *pindex, PORT_t *port, int bit, uint16_t *p_ccx);
5  int  set_level_array(uint16_t *p_array, int n);
```

De functie `set_level_array` koppelt de globale pointer `p_level` aan de array met intensiteiten, die in het hoofdprogramma moet worden gedefinieerd. Pointer `p_array` wijst naar dat array en de parameter `n` is het aantal array-elementen. De functie geeft de middelste index van de array terug. Bij de aanroep in het hoofdprogramma op regel 9 van code 22.9 wordt deze waarde aan de indices `r`, `g` en `b` toegekend. Bij de initialisatie krijgen de `CCx`-registers zo de waarde van het middelste niveau en register `PER` krijgt de waarde van het hoogste niveau.

Code 22.8 : Het bestand `fade.c` met de functie `fade`.

```
1  #include <avr/io.h>
2  #include "fade.h"
3
4  uint16_t *p_level;
5  int       nlevels;
6
7  void fade(int dir, int *p_index, PORT_t *port, int bit, uint16_t *p_ccx)
8  {
9    if ( bit_is_clear(port->IN, bit) ) {
10     if ( dir == UP )  {
11       if (*p_index < nlevels-1) (*p_index)++;
12     } else {
13       if (*p_index > 0) (*p_index)--;
14     }
15     *p_ccx = p_level[*p_index];
16     loop_until_bit_is_set(port->IN, bit);
17   }
18 }
19
20 int set_level_array(uint16_t *p_array, int n)
21 {
22   p_level = p_array;
23   nlevels = n;
24
25   return nlevels/2;
26 }
```

De functie `fade` bevat in feite één van de **`if`**-statements uit code 22.6. Er is een extra test en een variabele `dir` toegevoegd, die de richting van de fading aangeeft. De pointer `port` wijst naar de poort en het bitnummer `bit` van de bijbehorende drukknop en pointer `p_ccx` wijst naar het `CCx`-register van de uitgang. Pointer `p_index` wijst naar een index voor het intensiteitsniveau van de betreffende led.

In het hoofdprogramma is voor elke kleur een niveau-index gedeclareerd. Bij de aanroep van `fade` wordt het adres van de betreffende index aan `fade` meegegeven.

Code 22.9 : Het hoofdprogramma voor de aansturing van een rgb-led. ($f_{cpu} = 2$ MHz)

```
1  #include <avr/io.h>
2  #include "fade.h"
3
4  uint16_t level[] = {34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181};
5
6  int main(void) {
7    int r, g, b;                         // indices for intensity red, green and blue
8
9    r = g = b = set_level_array(level, sizeof(level)/sizeof(uint16_t));
10   PORTD.DIRCLR    = PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
11   PORTCFG.MPCMASK = PIN5_bm|PIN4_bm|PIN3_bm|PIN2_bm|PIN1_bm|PIN0_bm;
12   PORTD.PIN0CTRL  = PORT_OPC_PULLUP_gc;
13
14   PORTF.DIRSET = PIN1_bm|PIN0_bm;
15   PORTC.DIRSET = PIN0_bm;
16   TCF0.CTRLB   = TC0_CCBEN_bm|TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
17   TCC0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
18   TCF0.PER     = level[sizeof(level)/sizeof(uint16_t) - 1];
19   TCC0.PER     = level[sizeof(level)/sizeof(uint16_t) - 1];
20   TCF0.CCB     = level[r];
21   TCF0.CCA     = level[g];
22   TCC0.CCA     = level[b];
23   TCF0.CTRLA   = TC_CLKSEL_DIV1_gc;
24   TCC0.CTRLA   = TC_CLKSEL_DIV1_gc;
25
26   while (1) {
27     fade(UP,   &r, &PORTD, PIN0_bp, (uint16_t *) &TCF0.CCBBUF);
28     fade(DOWN, &r, &PORTD, PIN1_bp, (uint16_t *) &TCF0.CCBBUF);
29     fade(UP,   &g, &PORTD, PIN2_bp, (uint16_t *) &TCF0.CCABUF);
30     fade(DOWN, &g, &PORTD, PIN3_bp, (uint16_t *) &TCF0.CCABUF);
31     fade(UP,   &b, &PORTD, PIN4_bp, (uint16_t *) &TCC0.CCABUF);
32     fade(DOWN, &b, &PORTD, PIN5_bp, (uint16_t *) &TCC0.CCABUF);
33   }
34 }
```

De niveau-index wordt door `fade` opgehoogd als de richting `UP` is en verlaagd als de richting `DOWN` is. De functie `fade` kent de bij de index behorende intensiteit toe aan het betreffende `CCxBUF`-register.

De drukknoppen zijn aangesloten met een interne pullup. Op regel 11 is de *multi-pin configuration* toegepast. Deze methode is al eerder in paragraaf 17.8 ter sprake gekomen en zet in één keer de pullups van de zes ingangen aan.

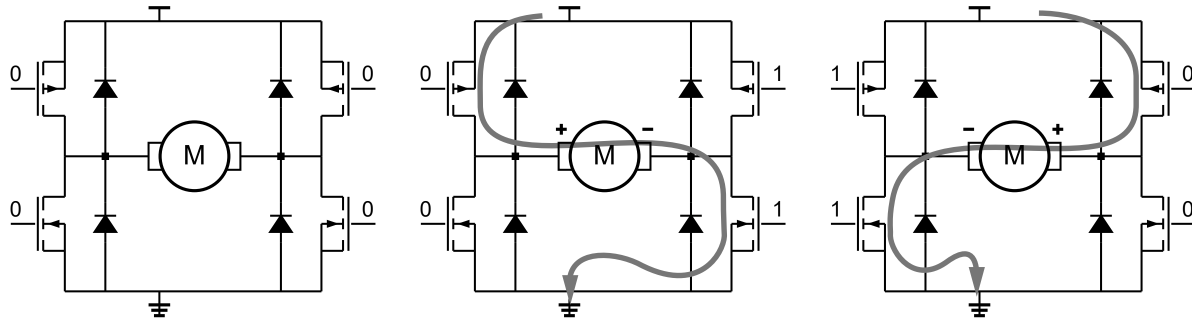
22.5 De dual-slope-modus: een robotwagen met DC-motoren

Hét voorbeeld voor het gebruik van PWM-signalen is de aansturing van motoren. Er bestaan vele soorten motoren, onder andere: wisselstroom- en gelijkstroom-motoren, 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.

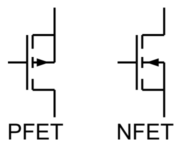
Een FET is een *Field Effect Transistor*. Dit kan een MOSFET of een JFET, *Junction* FET, zijn. Een NFET is een N-*channel* FET en een PFET is een P-*channel* FET.

Een NFET geleidt als de gate van de FET hoog is. Een PFET geleidt als de gate laag is.

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*, die 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 de referentie. Is echter de gate van de andere NFET hoog en van de andere PFET laag is, dan 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.



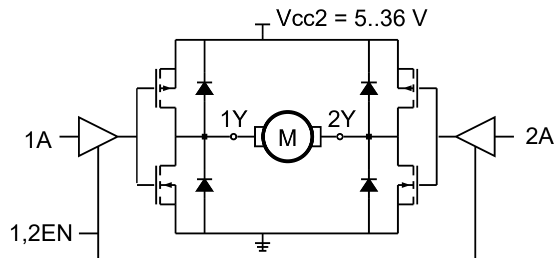
Figuur 22.13 : Drie verschillende situaties voor de H-brug bij de aansturing van een motor. Bij de linker H-brug zijn de gates van de FET laag en staat de motor stil. Bij de middelste en de rechter H-brug geleiden twee schuin tegenover elkaar liggende FET's. Er loopt dan een stroom door de motor. Afhankelijk van de richting zal de motor linksom of rechtsom draaien.



Figuur 22.14 : De symbolen van de NFET en de PFET.

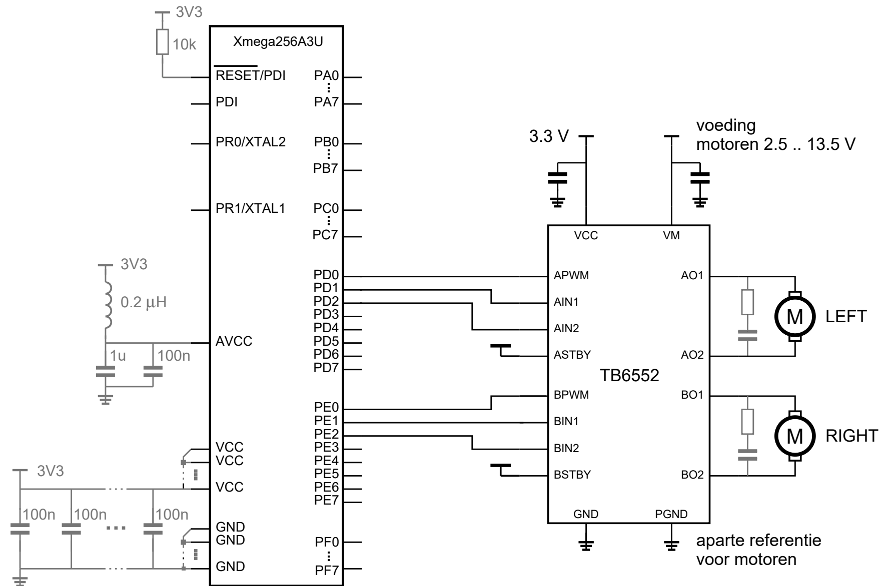
Zolang er een stroom door de motor en de H-brug loopt, spelen de vier diodes geen enkele rol. De motor bouwt door het draaien zelf een intern elektromagnetisch veld op. Als de stroom wordt uitgeschakeld, of als de stroomrichting verandert, geleiden de FET's niet meer en werkt de motor tijdelijk als dynamo. 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 in figuur 22.13 allebei aan staan, kan de stroom door de FET's te groot zijn. De ontwerper moet deze situatie vermijden.



Figuur 22.15 : Het basisconcept van een H-brug. De motor is verbonden met de aansluitingen 1Y en 2Y. De enable-ingang 1,2EN kan de ingangen 1A en 2A koppelen en ontkoppelen van de H-brug. Als de ingangen 1A en 2A van de halve bruggen een tegengestelde waarde hebben, draait de motor links- of rechtsom. De snelheid wordt geregeld door de enable-ingang met een PWM-signaal aan te sturen.

H-bruggen zijn ook als complete geïntegreerde schakeling verkrijgbaar. Populair is de L293D. Deze component bevat vier halve H-bruggen die als twee volledige H-bruggen gebruikt kunnen worden. In figuur 22.15 staat het basisconcept van één van deze volledige H-bruggen.



Figuur 22.16 : Het schema voor de aansturing van twee DC-motoren.

Een nadeel van de L293D is dat het een 5 V component is. Er bestaan echter veel alternatieven. In figuur 22.16 staat een implementatie met een Xmega256a3u, een dual H-bridge TB6552 en twee DC-motoren. Dit kunnen de motoren van een robotwagen zijn. Het uitgangsblok `CCA` van timer/counter `0` van poort D is aangesloten op de enable-ingang `APWM` van de linker motor. De draairichting wordt bepaald met de ingangen `AIN1` en `AIN2`, die aangesloten zijn op `PD1` en `PD2`. De aansturing van de rechter motor is op dezelfde wijze verbonden met poort E van de microcontroller.

Code 22.10 : De functie `motor_on` om een motor te laten draaien.

```
 7  void motor_on(uint8_t m, uint8_t dir, uint8_t duty)
 8  {
 9    uint16_t ccx;
10
11    ccx = 10000-(duty*10000)/100;
12    if ( m == LEFT ) {
13      if (dir == CLOCKWISE) {
14        PORTD.OUTCLR = PIN1_bm;
15        PORTD.OUTSET = PIN2_bm;
16      } else {
17        PORTD.OUTCLR = PIN2_bm;
18        PORTD.OUTSET = PIN1_bm;
19      }
20      TCD0.CCABUF  = ccx;
21    }
23    if ( m == RIGHT ) {
24      if (dir  == CLOCKWISE) {
25        PORTE.OUTCLR = PIN1_bm;
26        PORTE.OUTSET = PIN2_bm;
27      } else {
28        PORTE.OUTCLR = PIN2_bm;
29        PORTE.OUTSET = PIN1_bm;
30      }
31      TCE0.CCABUF  = ccx;
32    }
33  }
```

In code 22.10 staat een opzet voor de functie `motor_on`, die een motor `m` laat draaien in een draairichting `dir` en met een snelheid `duty`. De snelheid is een getal van `0` tot 100 en komt overeen met de duty-cycle. De functie bepaalt op regel 11 met formule 22.5 de waarde `ccx` die in het `CCx`-register moet komen te staan. Als `m` gelijk is aan `LEFT` wordt de linker motor aangestuurd en wordt `ccx` op regel 20 toegekend

aan CCABUF van TCD0. Als de draairichting CLOCKWISE is, is PD2 hoog en PD1 laag en anders is PD2 laag en PD1 hoog. Als m gelijk is aan RIGHT wordt hetzelfde gedaan, maar dan voor TCE0 en de uitgangen PE2 en PE1.

Code 22.11 : De initialisatiefunctie init_motor.

```
35  void init_motor(void)
36  {
37    PORTD.OUTSET = PIN2_bm|PIN1_bm|PIN0_bm;          // outputs low
38    PORTD.DIRSET = PIN2_bm|PIN1_bm|PIN0_bm;          // outputs
39    TCD0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_DSBOTH_gc; // dual slope
40    TCD0.CTRLA   = TC_CLKSEL_DIV1_gc;                // no prescaling
41    TCD0.PER     = 10000;                            // f=100 Hz @ 2 MHz
42    PORTE.OUTSET = PIN2_bm|PIN1_bm|PIN0_bm;          // outputs low
43    PORTE.DIRSET = PIN2_bm|PIN1_bm|PIN0_bm;          // outputs
44    TCE0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_DSBOTH_gc; // dual slope
45    TCE0.CTRLA   = TC_CLKSEL_DIV1_gc;                // no prescaling
46    TCE0.PER     = 10000;                            // f=100 Hz @ 2 MHz
47  }
```

De functie init_motor uit code 22.11 definieert de uitgangen en initialiseert beide timers met een dual-slope-modus, CCA als uitgangsblok en een prescaling van 1. De functie motor_off uit code 22.12 stopt een motor door de uitgangen, die de MOSFET's aansturen, laag te maken. Register CCA krijgt de waarde, die ook in PER staat. Het PWM-signaal is dan hoog, zodat de MOSFET's ook daadwerkelijk met een laag signaal worden aangestuurd.

Code 22.12 : De functie motor_off om een motor te stoppen.

```
49  void motor_off(uint8_t m)
50  {
51    if ( m == LEFT ) {
52      PORTD.OUTCLR = PIN2_bm|PIN1_bm;      // outputs low
53      TCD0.CCABUF  = 10000;                // PWM on
54    }
55    if ( m == RIGHT ) {
56      PORTE.OUTCLR = PIN2_bm|PIN1_bm;      // outputs low
57      TCE0.CCABUF  = 10000;                // PWM on
58    }
59  }
```

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 22.13 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.

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

Code 22.13 : Een aantal functies om een robotwagen te besturen.

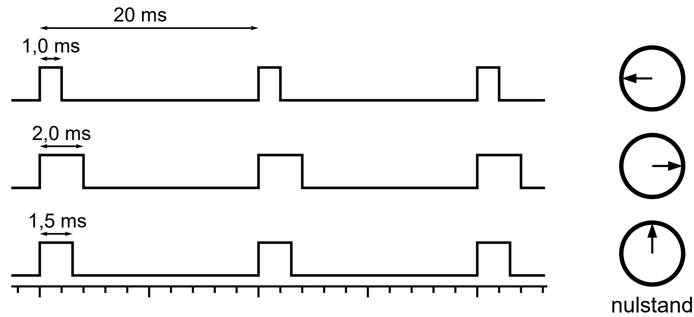
```
61  void car_forward(unsigned char speed)
62  {
63    motor_on(LEFT, CLOCKWISE, speed);
64    motor_on(RIGHT, CLOCKWISE, speed);
65  }
66
67  void car_backward(uint8_t speed)
68  {
69    motor_on(LEFT, COUNTERCLOCKWISE, speed);
70    motor_on(RIGHT, COUNTERCLOCKWISE, speed);
71  }
72
73  void car_stop()
74  {
75    motor_off(LEFT);
76    motor_off(RIGHT);
77  }
78
79  void car_left(uint8_t speed)
80  {
81    motor_on(LEFT, COUNTERCLOCKWISE, speed);
82    motor_on(RIGHT, CLOCKWISE, speed);
83  }
84
85  void car_left_curve(uint8_t speed)
86  {
87    motor_on(LEFT, CLOCKWISE, speed>>1);
88    motor_on(RIGHT, CLOCKWISE, speed);
89  }
```

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.

22.6 De dual-slope-modus: 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° en is bedoeld voor onder andere robots. Bij een pulsduur van 1,5 ms bevindt de servo zich in de zogenoemde nulstand. Figuur 22.17 toont de PWM-signalen voor de nulstand en de twee uitersten.

De servo kan worden aangestuurd met zowel de single- als de dual-slope-modus. Het is gebruikelijk om de servo aan te sturen met een PWM-signaal van ongeveer 50 Hz. Bij de dual-slope-modus volgt voor een frequentie 50 Hz en een f_{cpu} van 2 MHz uit formule 22.5 dat N `PER` gelijk is aan 20000. Voor een prescaling N van 2 moet `PER` dan gelijk zijn aan 10000.



Figuur 22.17 : De PWM-signalen voor een servomotor bij de nulstand en bij de twee uitersten.

Belangrijk is dat de nulstand goed ingesteld kan worden. Voor een frequentie van 50 Hz is de periodetijd 20 ms en is een pulsbreedte van 1,5 ms nodig. De duty-cycle is daarmee bekend en uit formule 22.6 volgt dan dat `CCx` gelijk aan 750 moet zijn. Als de waarde van `CCx` tussen 500 en 1000 ligt, ligt de pulsbreedte tussen 1,0 ms en 2,0 ms.

Code 22.14 : De besturing van een servomotor.

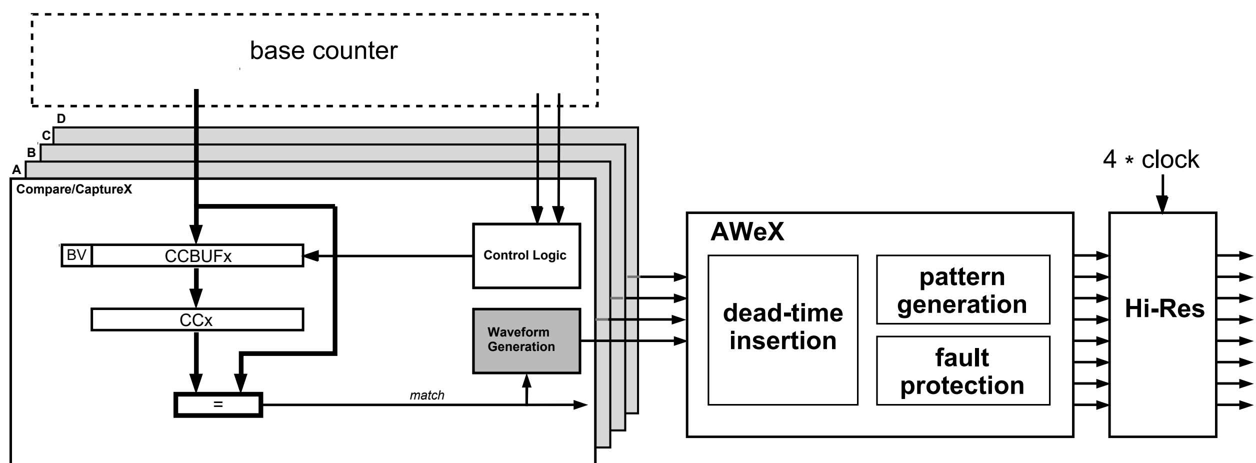
```
 1 #include <avr/io.h>
 2
 3 #define STEP   5
 4
 5 int main(void) {
 6   PORTA.OUTCLR   = PIN1_bm|PIN0_bm;
 7   PORTA.PIN0CTRL = PORT_OPC_PULLUP_gc;
 8   PORTA.PIN1CTRL = PORT_OPC_PULLUP_gc;
 9
10   PORTD.OUTSET = PIN0_bm;
11   PORTD.DIRSET = PIN0_bm;
12   TCD0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_DSBOTH_gc;
13   TCD0.CTRLA   = TC_CLKSEL_DIV2_gc;            // no prescaling
14   TCD0.PER     = 10000;                        // Tper  = 20 ms  @ 2 MHz
15   TCD0.CCA     = 750;                          // Tpuls = 1.5 ms @ 2 MHz
16
17   while (1) {
18     if(bit_is_clear(PORTA.IN, 0)){
19       if ( TCD0.CCA <= (1000 - STEP) ) TCD0.CCABUF = TCD0.CCA + STEP;
20       loop_until_bit_is_set(PORTA.IN, 0);
21     }
22     if(bit_is_clear(PORTA.IN, 1)) {
23       if ( TCD0.CCA >= (500 + STEP) )  TCD0.CCABUF = TCD0.CCA - STEP;
24       loop_until_bit_is_set(PORTA.IN, 1);
25     }
26   }
27 }
```

Code 22.14 varieert de pulsbreedte in stappen van 5. De maximale hoek is 180° en komt overeen met 100 stappen. Eén stap is dan gelijk aan een verdraaiing van 1,8°. Een stapgrootte 1 verdeelt de hele hoek zelfs in 500 stappen van 0,36°.

De servo is aangesloten op aansluiting `0` van poort D. Timer `TCD0` is ingesteld op de dual-slope-modus en de servo is aanvankelijk ingesteld op de nulstand. Pin `0` en pin 1 van poort A zijn met twee drukknoppen verbonden. Als de knop van pin `0` ingedrukt is, wordt register `CCA` van timer/counter `TCD0` met de stapgrootte `STEP` opgehoogd en de motor draait 1,8° met de klok mee. Als de knop van pin 1 ingedrukt is, wordt register `CCA` met de stapgrootte `STEP` verlaagd en de motor draait 1,8° tegen de klok in.

22.7 AWeX: advanced waveform extension

Sommige timer/counters van de Xmega hebben een uitbreiding voor een meer gespecialiseerde waveformgeneratie: de AWeX of *advanced waveform extension*. Deze uitbreiding is speciaal bedoeld is voor vermogensregelingen en motoraansturingen. De AWeX kan een dode tijd aan de uitgangssignalen toevoegen, foutprotectie toevoegen en speciale signaalpatronen genereren.



Figuur 22.18 : Het blokschema van de timer/counters van de Xmega met de AWeX-module.

Figuur 22.18 geeft de timer/counter met de AWeX- en de HiRES-uitbreiding. HiRES staat voor high-resolution en deze uitbreiding verhoogt de resolutie van de uitgangen van de waveformgenerator.

De TB6552 uit figuur 22.16 bevat intern een dode-tijdcorrectie, die ook met behulp van de AWeX geïmplementeerd had kunnen worden.

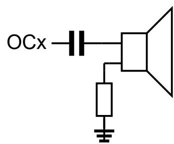
De Xmega256a3u heeft alleen bij timer/counter `0` een AWeX en wordt in dit boek verder niet behandeld. De uitbreiding AWeX is vooral interessant bij de aansturing van borstelloze DC-motoren. Atmel heeft een aantal application notes over de AWeX, zoals de AVR1607. Deze application note beschrijft de implementatie van een aansturing voor een BLDC, *brushless DC*, met gebruikmaking van drie hallsensoren.

22.8 De frequentiemodus: 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.

Een toon afspelen kan met een luidspreker of een buzzer. Er zijn magnetische en piëzo-elektrische buzzers.

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 format te verzinnen en muziekstukken om te zetten naar dat format, maar het is handiger om een bestaand format 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 Language* 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.

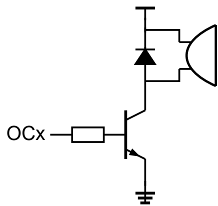


Figuur 22.19 : Aansluiting met luidspreker.
De weerstand beperkt de stroom en limiteert het volume.

Op het internet zijn vele oplossingen voor het afspelen van muziek met een microcontroller te vinden. Meestal gebruikt men een eigen format voor de muziek. Soms ontwikkelt men daarbij een aparte pc-applicatie die de RTTTL-beltonen converteert naar het eigen format. RTTTL beschrijft een muziekstuk in één grote tekststring. Een voorbeeld staat in code 22.15.

Code 22.15 : De RTTTL-string van *Für Elise*.

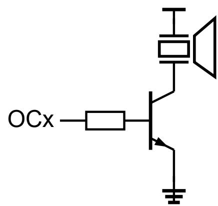
```
char Furelise[] = "FurElise:d=8,o=5,b=125:\
                   32p,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,32p,c,e,a,4b.,\
                   32p,e,g#,b,4c.6,32p,e,e6,d#6,e6,d#6,e6,b,d6,c6,4a.,\
                   32p,c,e,a,4b.,32p,d,c6,b,2a";
```



Figuur 22.20 : Aansluiting bij een magnetische buzzer. De diode voert de inductiestroom af.

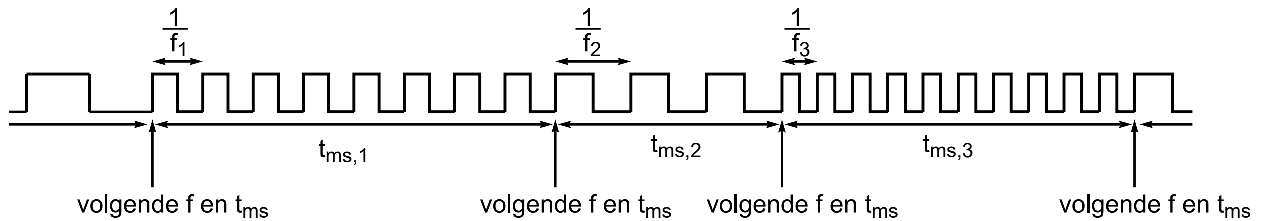
Het format is zeer compact en relatief eenvoudig en is daarom zeer geschikt als format om eenstemmige muziekstukken in op te slaan. Een groot voordeel om RTTTL te gebruiken is dat er op het internet honderden beltonen in dit format beschikbaar zijn en dat deze niet geconverteerd hoeven te worden.

De applicatie uit deze paragraaf gebruikt een RTTTL-bibliotheek met twee functies `readRTTTLdefaults` en `readRTTTLnote`. De eerste functie leest de standaardparameters `d`, `o` en `b` uit een RTTTL-string. In code 22.15 zijn dat respectievelijk 8, 5 en 125. De tweede functie leest een noot uit de RTTTL-string en geeft de frequentie en de tijdsduur van de noot via de parameterlijst terug. In code 22.15 is de eerste noot `32p` en de laatste noot `2a`.



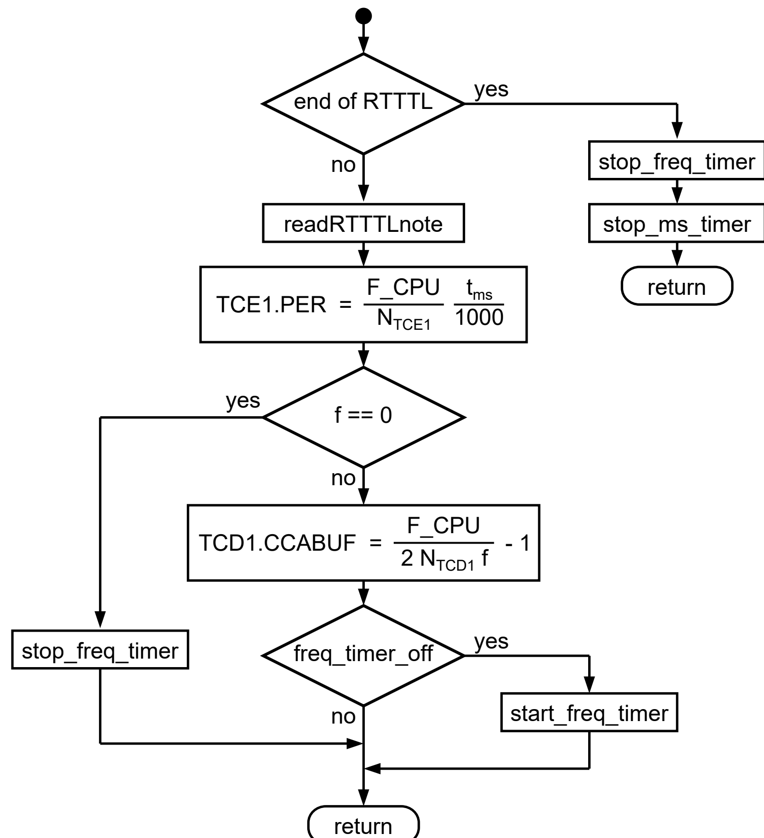
Figuur 22.21 : Aansluiting bij een piëzo-elektrische buzzer.

Het maken van een eenstemmige melodie met een microcontroller kan gedaan worden door een PWM-signaal op een luidspreker of een buzzer aan te sluiten, zie de figuren 22.19, 22.20 en 22.21.
Figuur 22.22 geeft een voorbeeld van een PWM-signaal dat een eenstemmig muziekstuk weergeeft. Voor de implementatie zijn twee timer/counters nodig: één voor het genereren van het PWM-signaal met de juiste frequentie en één voor het bepalen van de toonduur. De laatste timer start iedere keer als de toonduur verstreken is een interruptfunctie, die de frequentie en de toonduur van de volgende toon leest en de beide timers opnieuw instelt.



Figuur 22.22 : 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.

Het stroomdiagram van de interruptfunctie staat in figuur 22.23. Timer/counter `TCE1` definieert de tijdsduur en is ingesteld op de normale modus en `TCD1` definieert de frequentie en is ingesteld op de frequentiemodus. De interruptfunctie leest met `readRTTTLnote` een nieuwe noot en stelt achtereenvolgens de nieuwe tijdsduur en de nieuwe frequentie in. De tijdsduur wordt bepaald door register `PER` van timer `TCE1`. De nieuwe waarde van `PER` volgt uit formule 16.2. In het stroomdiagram is t_{ms} de tijd in ms en is N_{TCE1} de prescaling van timer/counter `TCE1`. De frequentie wordt bepaald door register `CCA` van timer `TCD1`. De nieuwe waarde van `CCA` volgt uit formule 22.2. In het stroomdiagram is N_{TCD1} de prescaling van timer `TCD1`.



Figuur 22.23 : Het stroomdiagram met het algoritme voor de `ISR`.

Als de RTTTL-string leeg is, is de beltoon helemaal afgespeeld en stoppen de beide timers. Als de frequentie van de nieuwe noot nul is, is er een rust en stopt de timer voor de frequentie. Bij de volgende *echte* toon wordt deze timer weer aangezet.

Code 22.16 : Het hoofdprogramma voor het afspelen van RTTTL-beltonen.

```
1  #define  F_CPU   2000000UL
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5  #include "rtttl.h"
6  #include "rtttl_lib.h"
7  #include "timers.h"
8
9  volatile char *rtttl;
10
11 void playRTTTL(char *p)
12 {
13   rtttl = readRTTTLdefaults(p);
14   start_ms_timer();
15 }
16
17 int main(void)
18 {
19   PORTD.OUTCLR = PIN4_bm;
20   PORTD.DIRSET = PIN4_bm;
21   TCD1.CTRLA   = TC_CLKSEL_OFF_gc;
22   TCD1.CTRLB   = TC0_CCAEN_bm |
23                  TC_WGMODE_FRQ_gc;
24   TCE1.CTRLA   = TC_CLKSEL_OFF_gc;
25   TCE1.CTRLB   = TC_WGMODE_NORMAL_gc;
26
27   PMIC.CTRL |= PMIC_LOLVLEN_bm;
28   sei();
29
30   playRTTTL(Furelise);
31   while ( playingRTTTL ) ;
32   playRTTTL(Wilhelmus);
33   while ( 1 ) ;
34 }
```

```
36 ISR(TCE1_OVF_vect)
37 {
38   uint16_t f, ms;
39
40   if ( *rtttl == '\0') {
41     stop_freq_timer();
42     stop_ms_timer();
43     return;
44   }
45
46   rtttl = readRTTTLnote((char *) rtttl, &f, &ms);
47   TCE1.PER = (((uint32_t) F_CPU/1000*ms) >> 10);
48
49   if (f == 0) {
50     stop_freq_timer();
51   } else {
52     TCD1.CCABUF = (((uint32_t) F_CPU/f) >> 1) - 1;
53     if ( freq_timer_off ) {
54       start_freq_timer();
55     }
56   }
57
58   return;
59 }
```

De prescaling N_{TCE1} is 1024 en de prescaling N_{TCD1} is 1.

De schuifoperator bij de berekening van `PER` vervangt het delen door de prescaling. Delen door 1024 is immers identiek met het 10 bits naar rechts schuiven.

Op dezelfde manier vervangt bij de berekening van `CCABUF` het één bit naar rechts schuiven de deling door 2.

Het programma uit code 22.16 speelt achter elkaar twee RTTTL-melodieën af die in `rtttl.h` staan. Het headerbestand `rtttl_lib.h` hoort bij de RTTTL-bibliotheek en bevat de prototypen van de functies `readRTTTLdefaults` en `readRTTTLnote`. Voor het starten en stoppen van de timer/counters zijn aparte functies beschikbaar, die in code 22.18 staan. Het headerbestand `timers.h`, zie code 22.17, bevat de prototypen van deze functies en de macro's `playingRTTTL` en `freq_timer_off`.

Code 22.17 : Het headerbestand `timers.h`.

```
1  #define  playingRTTTL    (TCE1.CTRLA > 0)
2  #define  freq_timer_off  (TCD1.CTRLA == 0)
3
4  void start_ms_timer(void);
5  void stop_ms_timer(void);
6  void start_freq_timer(void);
7  void stop_freq_timer(void);
```

Meer informatie over RTTTL en over de gebruikte RTTTL-bibliotheek is te vinden in paragraaf 22.6 en in bijlage E van de uitgave 'Microcontrollers en de taal C' van Wim Dolman.

In de bijlage E van dat boek worden de functies `readRTTTLdefaults` en `readRTTTLnote` uitgebreid toegelicht.

De functie `PlayRTTTL` van regel 11 heeft als ingangsparameter een pointer naar een RTTTL-string. De functie leest uit deze string de standaardwaarden en start de timer voor het meten van de toonduur. 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 timer-overflow-interrupt op en wordt de interruptfunctie gestart.

De functie `start_ms_timer` zet de timer/counter `TCE1` voor het meten van de tijdsduur aan. Het `PER`-register krijgt de willekeurige waarde. De interruptfunctie speelt de melodie af. Het algoritme begint pas na de eerste interrupt. De prescaling is 1024. Bij een systeemklok van 32 MHz is de maximale in te stellen tijdsduur 2000 ms en bij een systeemklok van 2 MHz is de fout bij het meten van een tijdsduur van 100 ms slechts 1%.
De functie `stop_ms_timer` stopt de timer door de klok uit te zetten. De overflow-interrupt van de timer is alleen actief als de timer gebruikt wordt.

Code 22.18 : Het bestand `timers.c` met de start- en stopfuncties voor de timers.

```
1  #include <avr/io.h>
2
3  void start_ms_timer(void)
4  {
5    TCE1.PER      = 10;                       // dummy value
6    TCE1.CNT      = 0;                        // reset timer/counter
7    TCE1.CTRLA    = TC_CLKSEL_DIV1024_gc;     // prescaling 1024
8    TCE1.INTCTRLA = TC_OVFINTLVL_LO_gc;       // enables overflow interrupt
9  }
10
11 void stop_ms_timer(void)
12 {
13   TCE1.CTRLA    = TC_CLKSEL_OFF_gc;         // timer/counter off
14   TCE1.INTCTRLA = TC_OVFINTLVL_OFF_gc;      // disables overflow interrupt
15 }
16
17 void start_freq_timer(void)
18 {
19   TCD1.CNT   = 0;                           // reset timer/counter
20   TCD1.CTRLA = TC_CLKSEL_DIV2_gc;           // prescaling 2
21 }
22
23 void stop_freq_timer(void)
24 {
25   TCD1.CTRLA = TC_CLKSEL_OFF_gc;            // timer/counter off
26   TCD1.CTRLC = 0;                           // output(s) low
27 }
```

De functie `start_freq_timer` start en de functie `stop_freq_timer` stopt de timer, die de frequentie van het PWM-signaal genereert. De prescaling is 1. De frequenties van RTTTL-melodieën liggen tussen 262 en 7902 Hz. Voor een systeemklok van 32 MHz is bij deze prescaling de laagste frequentie die kan worden ingesteld gelijk aan 244 Hz. Bij een systeemklok van 2 MHz is voor 7902 Hz de gemaakte fout maximaal, maar nog altijd kleiner dan 1%.
De functie `stop_ms_timer` maakt register `CTRLC` nul. Dit register bevat alleen vier `CMPx`-bits. Als de timer/counter uit staat krijgen de uitgangen van `CCx`-blokken de waarde van de betreffende `CMPx`-bits. In dit geval is bit `CMPA` laag en is de uitgang van blok `CCA` ook laag.

De macro `playingRTTTL` uit code 22.17 test of er een melodie wordt afgespeeld. Dit is het geval als het prescale-register `CTRLA` van timer `TCE1` ongelijk is aan nul. Op dezelfde manier staat de timer voor de frequentie uit als register `CTRLA` van timer `TCD1` gelijk is aan nul.

23

Nog meer Xmega

Doelstelling	Dit hoofdstuk behandelt in vogelvlucht een aantal mogelijkheden van de Xmega die in de vorige hoofdstukken nog niet aan de orde gekomen zijn.

Onderwerpen

De behandelde onderwerpen zijn:
- De digitaal-analoogconverter.
- Het gebruik van DMA.
- De analoge comparator.
- Hysterese.
- De *input capture*-modus van de timer/counter.
- Het kloksysteem van de Xmega.
- De realtime-counter.
- Het gebruik van het EEPROM.
- Het gebruik van flash.
- De slaapstanden van de Xmega.
- Het gebruik van de *watchdog*.
- Het *atomic block*.

Voorbeelden tonen:
- Toepassingen met de digitaal-analoogconverter.
- Een toepassing met DMA.
- Het genereren van een sinus met DMA en de digitaal-analoogconverter.
- Toepassingen van de analoge comparator met en zonder hysterese.
- Het gebruik van de *input capture*-modus.
- Het gebruik van de *input capture*-modus met een 32-bits timer/counter.
- Diverse voorbeeldfuncties voor het instellen van de verschillende klokken.
- Toepassingen met de realtime-counter.
- Toepassingen met het EEPROM.
- Het gebruik van strings en afdrukfuncties met flash.
- Diverse voorbeelden voor verschillende slaapstanden.
- Het gebruik van de watchdog.
- Het gebruik van de watchdog om de microcontroller in een veilige toestand te zetten.
- Een toepassing met een atomic block.

De hoofdstukken 15 tot en met 17 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.

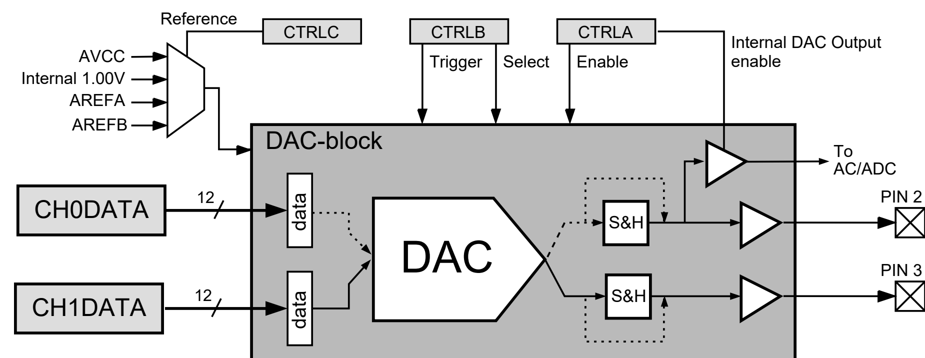
In hoofdstuk 20 tot en met 22 zijn de aansturing van een LCD, het gebruik van de ADC, de communicatie via de UART, de SPI en de I^2C-interface en het gebruik van de timers voor PWM-signalen uitgebreid besproken. Deze hoofdstukken tonen dat er vaak meerdere oplossingen zijn en laten zien dat het belangrijk is om een bepaalde eigenschap goed te bestuderen voordat deze gebruikt wordt.

Dit hoofdstuk geeft voor een aantal van de overige eigenschappen van de Xmega een beknopte uitleg. De pretentie van dit boek is niet om volledig te zijn: dat is niet nodig en niet mogelijk. Bestudeer daarvoor de datasheets, de AU-manual en de vele *application notes*.

23.1 Digitaal-analoogconverter

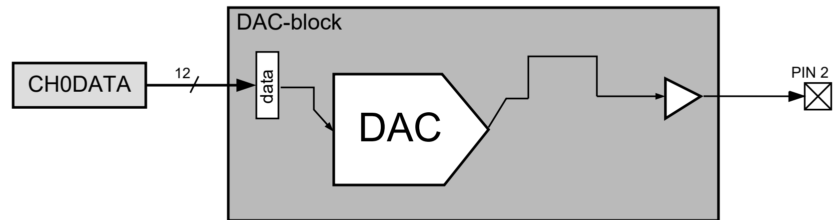
In hoofdstuk 20 is al verteld dat de buitenwereld van de microcontroller vaak analoog is en dat analoog-digitaalconversie en digitaal-analoogconversie bij de meeste systemen nodig zijn. Hoofdstuk 22 heeft laten zien dat met pulsbreedtemodulatie de gemiddelde waarde van een uitgangssignaal niet per se een 0 of een 1 hoeft te zijn. Een digitaal-analoogconverter of DAC is daarom niet altijd noodzakelijk en veel microcontrollers hebben dan ook geen digitaal-analoogconverter. Zo hebben bijvoorbeeld de Xmega's uit de B-, C- en D-serie geen DAC.

De Xmega's uit de A-, AU- en E-serie hebben één of twee 12-bits DAC's: bij poort A zit DACA en bij poort B zit DACB. De Xmega256a3u heeft alleen een DACB bij poort B. Het blokschema van de DAC uit de A-serie staat in figuur 23.1. Dit schema is anders dan dat van de DAC uit de AU-serie. De DAC uit de A-serie heeft één digitaal-analoogconverter en twee kanalen met een eigen dataregister en een eigen uitgang. Kanaal 0 is verbonden met pin 2 en kanaal 1 met pin 3 van de betreffende poort.



Figuur 23.1 : Het blokschema van de DAC uit de A-serie van de Xmega. Er zijn twee registers voor kanaal 0 en kanaal 1. In dit voorbeeld is register `CH1DATA` doorverbonden met pin 3 van de uitgang van kanaal 1.

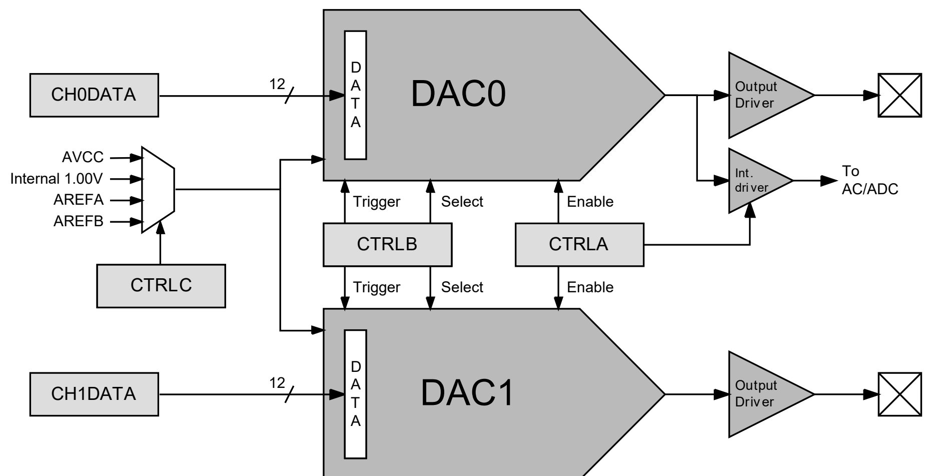
De DAC uit de A-serie kan worden gebruikt als single channel- en als dual channel-DAC. Bij het meten met twee kanalen wordt steeds aan één kanaal tegelijk gemeten. De eerdere meting met het tweede kanaal wordt vastgehouden door de sample&hold van dat kanaal. Als daarna het tweede kanaal opnieuw gemeten wordt, bewaart de sample&hold van het eerste kanaal weer zijn resultaat. Beide kanalen moeten bij deze tweekanaalsmethode regelmatig worden ververst.



Figuur 23.2 : Bij de éénkanaalsmethode is één uitgang direct verbonden met de DAC.

Bij de éénkanaalsmethode is een van de analoge uitgangen altijd via de DAC verbonden met het betreffende dataregister. De sample&hold wordt overgeslagen en het resultaat van de conversie staat altijd op de betreffende uitgang.

In figuur 23.3 staat het blokschema van de DAC uit de AU-serie. Deze DAC is geïmplementeerd in de Xmega256a3u en heeft bij beide kanalen een digitaal-analoogconverter. De sample&hold's zijn hier niet nodig. De werking van de DAC's uit de AU-serie is compatibel met die van de A-serie. Voor de tweekanaalsmethode is het bij de DAC uit de AU-serie niet nodig om de kanalen regelmatig te verversen. Alleen de interne digitaal-analoogconverter DAC0 kan intern doorverbonden worden met de AC of ADC.



Figuur 23.3 : Het blokschema van de DAC uit de AU-serie van de Xmega.

Er zijn drie methoden om de DAC te gebruiken: met alleen kanaal 0, met alleen kanaal 1 of met beide kanalen. De twee `CHSEL`-bits uit register `CTRLB` selecteren één van deze drie manieren. Er is echter nog een vierde methode. De uitgang van kanaal 0 kan ook rechtstreeks worden doorverbonden met één van de analoge comparatoren of met de ADC. Als de `IDOEN`-bit uit register `CTRLA` hoog is, gebruikt de DAC kanaal 0 en gaat het resultaat alleen naar de genoemde componenten en *niet* naar pin 2.

De externe referentiespanning moet liggen tussen 1 V en AVCC—0,6 V.

De uitgangsspanning van de DAC hangt af van de referentiespanning V_{ref}. Dit kan een interne bandgap-referentie zijn van 1,00 V, de analoge voedingsspanning AVCC of één van de externe referenties AREFA of AREFB. Voor de 12-bits DAC geldt dat de uitgangsspanning V_{DAC} gelijk is aan:

$$V_{DAC} = \frac{\text{CHnDATA}}{4095} V_{REF} \tag{23.1}$$

Hierbij is CHnDATA de digitale waarde die geconverteerd wordt.

Code 23.1 : Een programma dat met behulp van de DAC een driehoekvormig signaal genereert.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  #define STEP_VALUE  100
7  #define MAX_VALUE   2400
8  #define MIN_VALUE   0
9
10 uint16_t triangle(void)
11 {
12   static uint16_t value = MIN_VALUE;
13   static int16_t  delta = STEP_VALUE;
14
15   if ( ((delta > 0) && (value >= MAX_VALUE)) ||
16        ((delta < 0) && (value <= MIN_VALUE)) ) {
17     delta = -delta;
18   }
19   value += delta;
20
21   return value;
22 }
24 void init_dac(void)
25 {
26   DACB.CTRLC = DAC_REFSEL_AVCC_gc;
27   DACB.CTRLB = DAC_CHSEL_SINGLE_gc;
28   DACB.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;
29 }
30
31 int main(void)
32 {
33   init_dac();
34
35   while (1) {
36     DACB.CH0DATA = triangle();
37     while (!DACB.STATUS & DAC_CH0DRE_bm);
38
39     _delay_ms(200);
40   }
41 }
```

Driehoekvormig signaal met éénkanaalsmethode

Het programma uit code 23.1 genereert een driehoekvormige signaal. De functie init_dac initialiseert de DAC en stelt deze in op de éénkanaalsmethode voor kanaal 0. Pin 2 van poort B is dan automatisch de uitgang van de DAC. De functie triangle genereert het driehoekvormig signaal. Bij iedere aanroep geeft deze functie de volgende waarde van het signaal terug. Om een driehoek te krijgen moet triangle wel op regelmatige tijdstippen worden aangeroepen. In dit voorbeeld is dat iedere 200 ms.

De driehoek wordt gemaakt door iedere keer een vaste waarde bij de huidige waarde op te tellen of af te trekken. Bij de maximale waarde MAX_VALUE en de minimale waarde MIN_VALUE keert de telrichting om. In dit voorbeeld ligt de waarde, die de DAC krijgt, tussen 0 en 2400. Met formule 23.1 ligt het uitgangssignaal dan tussen 0 en 1,93 V. In de praktijk zal het bereik van het uitgangssignaal hiervan afwijken.

Deze afwijking kan in principe worden opgelost door de DAC te kalibreren. Bij de productie test Atmel de Xmega en kalibreert verschillende onderdelen. De resultaten worden in het NVM, *non volatile memory*, geplaatst. De gebruiker kan deze waarden lezen en in de juiste kalibratieregisters zetten.

Code 23.2 : De functie `readCalibrationWord` leest een 16-bits getal uit het NVM.

```
24 #include <avr/pgmspace.h>
25
26 uint8_t readCalibrationByte(uint8_t index)
27 {
28   uint8_t result;
29
30   NVM_CMD = NVM_CMD_READ_CALIB_ROW_gc;
31   result = pgm_read_byte(index);
32   NVM_CMD = NVM_CMD_NO_OPERATION_gc;
33
34   return result;
35 }
```

De fabriekskalibratie van de DAC bij de Xmega256a3u lijkt helaas niet correct.
Een nauwkeurig DAC-signaal kan alleen worden verkregen door de kalibratiewaarden zelf te meten. Paragraaf 2.6 van de application note AVR1301 geeft een kalibratieprocedure.

Code 23.2 definieert een functie `readCalibrationByte` die een kalibratiebyte uit het geheugen leest. Bij de initialisatie van de DAC kunnen met deze functie de offset en de *gain* worden gekalibreerd. In code 23.3 staat een voorbeeld. De macrodefinitie `offsetof` is gedeclareerd in `stddef.h`. Deze macro bepaalt de offset — in bytes — voor een bepaald veld uit een datastructuur.

Code 23.3 : De initialisatiefunctie `init_dac` met kalibratie.

```
37 #include <stddef.h>            // definition of offsetof
38
39 void init_dac(void)
40 {
41   DACB.CH0GAINCAL   = readCalibrationByte( offsetof(NVM_PROD_SIGNATURES_t, DACB0GAINCAL) );
42   DACB.CH0OFFSETCAL = readCalibrationByte( offsetof(NVM_PROD_SIGNATURES_t, DACB0OFFCAL) );
43   DACB.CTRLC = DAC_REFSEL_AVCC_gc;
44   DACB.CTRLB = DAC_CHSEL_SINGLE_gc;
45   DACB.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;
46 }
```

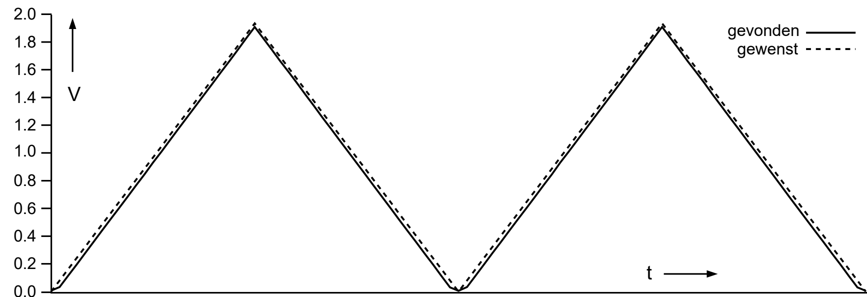
Kanaal `0` kan intern doorverbonden zijn met de ADC. Bij de AU-serie kan alleen DAC`0` intern verbonden zijn. DAC1 moet voor de kalibratie buitenom met de ADC verbonden worden.

Het resultaat van het programma van code 23.1 staat in figuur 23.4. De offset en de gain zijn handmatig gekalibreerd. Ondanks dat wijkt bij kleine waarden het analoge signaal van de DAC af van de gewenste waarde.

Bij de kalibratiemetingen is de ADC van de Xmega gebruikt. De DAC is daarbij intern doorverbonden met de ADC. De kalibratieprocedure wordt hier verder niet uitgewerkt. In de application note AVR1301 staat de procedure.

DAC intern doorverbinden met ADC of analoge comparator

In code 23.6 staat een applicatie die de DAC intern doorverbindt met de ADC en de gemeten waarde doorstuurt naar een UART. De handmatige kalibratie is bij deze beschrijving weggelaten. Zonder deze kalibratie wijken de gemeten spanning en de beoogde spanning van elkaar af.



Figuur 23.4 : Het signaal dat de applicatie van code 23.1 genereert.

In code 23.4 staat de initialisatiefunctie `init_dac`. Op regel 40 is het `IDOEN`-bit aangezet, waardoor de DAC een interne uitgang bij kanaal 0 heeft. Deze instelling vervangt die van de volgende regel. Regel 41 is daarmee overbodig.

Code 23.4 : De initialisatiefunctie `init_dac` voor een interne DAC-uitgang.

```
36  void init_dac(void)
37  {
38    DACB.CTRLC = DAC_REFSEL_AVCC_gc;
39    DACB.CTRLB = DAC_CHSEL_SINGLE_gc;
40    DACB.CTRLA = DAC_IDOEN_bm |      // internal output
41                 DAC_CH0EN_bm |      // overruled by previous
42                 DAC_ENABLE_bm;
43  }
```

In code 23.5 staat de initialisatiefunctie `init_adc`. Deze functie lijkt op de initialisatiefunctie uit code 20.3. In dit geval is op regel 48 de interne modus geselecteerd en op regel 47 is de interne uitgang van de DAC verbonden met de positieve ingang van de ADC. Als daarnaast ook de *signed* modus ingesteld is, is de negatieve ingang van de ADC automatisch intern verbonden met `GND`.

In code 23.6 staat alleen de `main` van het hoofdprogramma. De functies `init_uart`, `triangle`, `read_adc` zijn eerder al besproken. Na de initialisatie van de DAC, ADC en UART, genereert het hoofdprogramma iedere halve seconde een waarde van het driehoekvormig signaal, die de ADC vervolgens leest. De printfuncties sturen beide resultaten — met de bijbehorende waarde in mV — naar de UART0 van poort F.

Code 23.5 : De initialisatiefunctie `init_adc` met interne DAC als ingang.

```
45  void init_adc(void)
46  {
47    ADCA.CH0.MUXCTRL = ADC_CH_MUXINT_DAC_gc;          // internal DAC to + channel 0
48    ADCA.CH0.CTRL    = ADC_CH_INPUTMODE_INTERNAL_gc;  // internal input mode
49    ADCA.REFCTRL     = ADC_REFSEL_INTVCC_gc;
50    ADCA.CTRLB       = ADC_RESOLUTION_12BIT_gc |
51                       ADC_CONMODE_bm;                // signed
52    ADCA.PRESCALER   = ADC_PRESCALER_DIV16_gc;
53    ADCA.CTRLA       = ADC_ENABLE_bm;
54  }
```

Code 23.6 : De functie `main` van het programma met DAC, ADC en UART.

```
56 #include <avr/io.h>
57 #include <avr/interrupt.h>
58 #include "stream.h"
59
60 #define AVCC   3.30
61
62 int main(void)
63 {
64   uint16_t dac, adc;
65
66   init_dac();
67   init_adc();
68   init_stream(F_CPU);
69
70   PMIC.CTRL |= PMIC_LOLVLEN_bm;    // uart
71   sei();
72
73   while (1) {
74     dac   = triangle();
75     DACB.CH0DATA = dac;
76     while (!DACB.STATUS & DAC_CH0DRE_bm);
77
78     adc   = read_adc();
79
80     printf("DAC : %4d  %1.3f  ==>  ",  dac, (double) dac * AVCC / 4095);
81     printf("ADC : %4d  %1.3f\n",       adc, (double) adc/1000);
82     _delay_ms(500);
83   }
84 }
```

In paragraaf 18.8 is uitgelegd dat standaard de `printf`-functies bij AVR-gcc geen gebroken getallen kennen. Voor de applicatie uit code 23.6 zijn deze compiler- en linkeropties noodzakelijk:
`-Wl,-u,vfprintf`
`-lprintf_flt -lm`

23.2 Direct Memory Access

DMA, *Direct Memory Access* geeft de mogelijkheid om gegevens te verplaatsen zonder — of met een minimale — tussenkomst van de processor. De Xmega256a3u heeft een 4-kanaals DMA-controller. Hiermee kunnen gegevens van een geheugengebied verplaatst worden naar een ander geheugengebied, van een geheugen naar een perifeer blok en omgekeerd en tussen twee perifere blokken.

Een DMA-transactie, *DMA transaction*, is de overdracht van alle gegevens, en bestaat uit één tot 16 miljoen bytes. Een DMA-transactie kan worden opgedeeld in verschillende bloktransfers, *block transfers*. De maximale grootte van een bloktransfer is 64 KB. De bloktransfer bestaat op zijn beurt weer uit bursttransfers, *burst transfers*, ter grootte van 1, 2, 4 of 8 bytes.

De DMA van de Xmega heeft veel verschillende mogelijkheden en instellingen. Bij het schrijven van een programma is de simulator van Atmel Studio of een debugger haast onmisbaar. Deze paragraaf geeft twee voorbeelden, die een aanzet voor andere implementaties kunnen zijn, namelijk het kopiëren van een blok met gegevens naar een andere plaats in het geheugen en het genereren van een sinus met een DAC vanuit een opzoektabel.

Het kopiëren van een geheugenblok

In code 23.7 staat een programma dat een array kopieert naar adres `0x2050` in het datageheugen. De originele array staat aan het begin van het datageheugen op adres `0x2000`. Na de initialisatie met `init_dma` wordt op regel 11 de transfer gestart. Vervolgens wordt op regel 12 gewacht totdat de transfer gereed is en doet het programma in een oneindige while verder niets meer.

Code 23.7 : Het kopiëren van een geheugenblok met behulp van DMA.

```
 1  #include <avr/io.h>
 2
 3  void init_dma(void);
 4
 5  uint8_t array[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
 6
 7  int main(void)
 8  {
 9    init_dma();
10
11    DMA.CH0.CTRLA  |= DMA_CH_TRFREQ_bm;             // start the DMA transfer
12    while( !(DMA.CH0.CTRLB & DMA_CH_TRNIF_bm) ) {   // wait for DMA transfer is completed
13      asm volatile ("nop");
14    }
15
16    while(1) {
17      asm volatile ("nop");                       // do nothing
18    }
19  }
20
21  void init_dma(void)
22  {
23    DMA.CTRL           = DMA_ENABLE_bm;                  // enable DMA-controller
24    DMA.CH0.CTRLA      = DMA_CH_ENABLE_bm |              // enable channel 0
25                         DMA_CH_BURSTLEN_8BYTE_gc;       // set burst length to 8 bytes
26    DMA.CH0.ADDRCTRL   = DMA_CH_SRCRELOAD_NONE_gc |      // no reload on source
27                         DMA_CH_SRCDIR_INC_gc |          // increment source address
28                         DMA_CH_DESTRELOAD_NONE_gc |     // no reload on destination
29                         DMA_CH_DESTDIR_INC_gc;          // increment destination address
30    DMA.CH0.TRIGSRC    = DMA_CH_TRIGSRC_OFF_gc ;         // manual trigger DMA transfer
31    DMA.CH0.TRFCNT     = sizeof(array)* sizeof(uint8_t);   // transfer size
32    DMA.CH0.SRCADDR0   = (unsigned int) array;           // set source address
33    DMA.CH0.SRCADDR1   = ((unsigned int) array) >> 8;
34    DMA.CH0.SRCADDR2   = 0;
35    DMA.CH0.DESTADDR0  = 0x50;                           // set destination address to 0x2050
36    DMA.CH0.DESTADDR1  = 0x20;
37    DMA.CH0.DESTADDR2  = 0;
38  }
```

De functie `init_dma` initialiseert eerst de DMA en kanaal `0` met een *burst*-lengte van 8 bytes. De overdracht is sneller bij een grote *burst*-lengte.

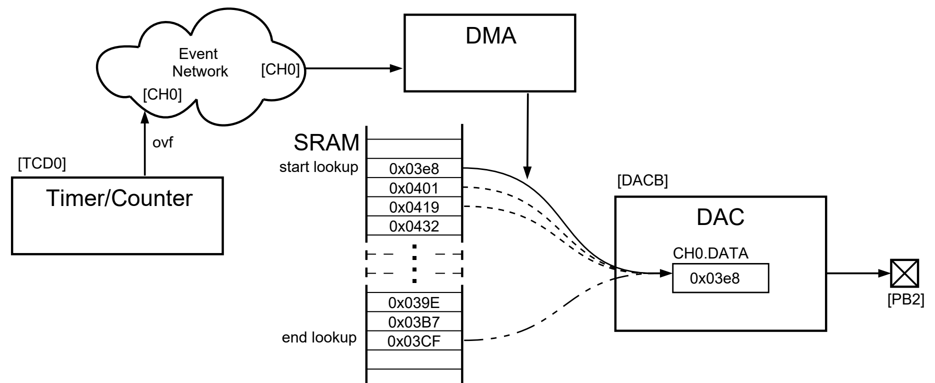
Regel 26 tot en met 29 stellen het adrescontrolregister in. De gegevens worden eenmalig geschreven, zodat er, zowel bij de bron als bij de bestemming, geen *reload* nodig is. Wel worden in beide gevallen na iedere burst de geheugenadressen aangepast.

Er is geen bron als trigger; de DMA-transfer wordt handmatig gestart. Het aantal transfers is gelijk aan het aantal te verzenden bytes en hangt dus af van de grootte van `array`. Het adres van de bron en de bestemming zijn 24-bits. Het adres van de bron, de array `array`, is 16-bits. In register `SCRADDR0` staan de minst significante bits. Het adres van de bestemming is `0x2050`.

```
0x2000  01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10  ← source
0x2010  11 12 13 14 15 16 17 18 00 00 00 00 00 00 00 00
0x2020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2050  01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10  ← destination
0x2060  11 12 13 14 15 16 17 18 00 00 00 00 00 00 00 00
0x2070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figuur 23.5 : Het datageheugen na het uitvoeren van code 23.7.

Figuur 23.5 toont het resultaat na het uitvoeren van code 23.7. De 24 bytes van de array op adres `0x2000` zijn gekopieerd naar adres `0x2050`.



Figuur 23.6 : Configuratie voor het genereren van een sinus met behulp van een opzoektabel en DMA. De timer/counter triggert via het eventnetwerk de DMA-module, die een waarde van het SRAM naar het dataregister van de DAC kopieert. De DMA start bij het begin van de opzoektabel en neemt bij de volgende trigger de volgende waarde. Bij het einde van de opzoektabel gaat de DMA terug naar het begin van de opzoektabel.

Het genereren van een sinus met een DAC en een opzoektabel

In figuur 23.6 staat de configuratie voor het genereren van een sinus met behulp van een opzoektabel en DMA. De overflow van de timer/counter triggert via het eventnetwerk de DMA. De DMA kopieert dan een waarde uit het geheugen naar het dataregister van de DAC. Als het einde van de opzoektabel bereikt wordt, gaat de DMA weer terug naar het begin. Doordat de timer/counter op vaste tijdstippen een trigger geeft, genereert de DAC een nette sinus. De frequentie van de sinus wordt bepaald door de periodetijd van de timer/counter.

De opzoektabel voor een sinus, die ligt tussen 0 en een topwaarde A, wordt gedefinieerd met formule 23.2. Het totaal aantal samples is N. Sample i uit de opzoektabel heeft een waarde die gelijk is aan sinus$[i]$.

$$\text{sinus}[i] = \frac{A}{2}\left(1 + \sin\left(\frac{2\pi\, i}{N}\right)\right) \qquad (23.2)$$

In code 23.8 staat een deel van een opzoektabel, die met formule 23.2 is gemaakt voor A gelijk aan 2000 en voor 256 samples. De waarden zijn hexadecimaal weergegeven. Deze opzoektabel is in een headerbestand `sinus.h` geplaatst.

Code 23.8 : Het headerbestand `sinus.h` met de opzoektabel.

```
1  uint16_t sinus[256] = {
2    0x03e8, 0x0401, 0x0419, 0x0432, 0x044a, 0x0462, 0x047b, 0x0493,
3    0x04ab, 0x04c3, 0x04db, 0x04f3, 0x050a, 0x0522, 0x0539, 0x0550,
4    ...
5    0x0269, 0x0280, 0x0297, 0x02ae, 0x02c6, 0x02dd, 0x02f5, 0x030d,
6    0x0325, 0x033d, 0x0355, 0x036e, 0x0386, 0x039e, 0x03b7, 0x03cf
7  };
```

Het hoofdprogramma staat in code 23.9. Het initialiseert achtereenvolgens de DAC, de DMA en de timer/counter en doet daarna niets meer. Er zijn zelfs geen interrupts. De microcontroller genereert de sinus zonder dat de processor zelf actief is.

Code 23.9 : Het genereren van een sinus met een DAC met behulp van DMA.

```
1  #include <avr/io.h>
2  #include "sinus.h"
3
4  void init_dac(void);
5  void init_dma(void);
6  void init_timer(void);
7
8  int main(void)
9  {
10   init_dac();
11   init_dma();
12   init_timer();
13
14   while(1) { }  // do nothing
15 }
```

In code 23.10 staan de initialisatiefuncties voor de DAC, de DMA en de timer. Als de klokfrequentie 2 MHz is, is de overflow van de timer ingesteld op een frequentie van ruim 10 kHz. Met 256 samples is de frequentie van de sinus ongeveer 40 Hz.
De DAC gebruikt kanaal 0 en wordt getriggerd zodra er nieuwe data in het dataregister staat. De referentiespanning is `AVCC` en deze is gelijk aan 3,3 V. Met een maximale waarde voor A van 2000 volgt met formule 23.1 dat de maximale spanning van de sinus gelijk is aan $1,61$ V.

Code 23.10 : De initialisatiefuncties voor het genereren van een sinus met DMA.

```
17 void init_dma(void)
18 {
19   DMA.CTRL          |= DMA_ENABLE_bm;
20   DMA.CH0.CTRLA      = DMA_CH_ENABLE_bm |                          // set DMA channel 0,
21                        DMA_CH_REPEAT_bm |                          // repeat mode
22                        DMA_CH_SINGLE_bm |                          // single shot
23                        DMA_CH_BURSTLEN_2BYTE_gc;                   // burst length to 2 bytes
24   DMA.CH0.ADDRCTRL   = DMA_CH_SRCRELOAD_TRANSACTION_gc |           // transaction on source
25                        DMA_CH_SRCDIR_INC_gc  |                     // increment source address
26                        DMA_CH_DESTRELOAD_BURST_gc |                // burst on destination
27                        DMA_CH_DESTDIR_INC_gc;                      // increment destination address
28   DMA.CH0.TRIGSRC    = DMA_CH_TRIGSRC_EVSYS_CH0_gc ;               // trigger from event channel 0
29   DMA.CH0.TRFCNT     = sizeof(sinus);                              // numberof bytes to transfer
30   DMA.CH0.SRCADDR0   = (unsigned int) sinus;                       // set start source address
31   DMA.CH0.SRCADDR1   = ((unsigned int) sinus) >> 8;
32   DMA.CH0.SRCADDR2   = 0;
33   DMA.CH0.DESTADDR0  = ((unsigned int) &(DACB.CH0DATA)) & 0xFF;    // set destination address
34   DMA.CH0.DESTADDR1  = ((unsigned int) &(DACB.CH0DATA)) >> 8;
35   DMA.CH0.DESTADDR2  = 0;
36 }
37
38 void init_dac(void)
39 {
40   DACB.CTRLB = DAC_CHSEL_SINGLE_gc;          // single-channel operation, trigger on data
41   DACB.CTRLC = DAC_REFSEL_AVCC_gc;           // use AVCC as DAC reference, value right-justified
42   DACB.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm; // enable DACB and channel 0
43 }
44
45 void init_timer(void)
46 {
47   TCD0.CTRLB     = TC_WGMODE_NORMAL_gc;      // normal mode
48   TCD0.CTRLA     = TC_CLKSEL_DIV1_gc;         // no prescaling
49   TCD0.PER       = 194;                       // set frequency 40 Hz with 256 samples
50   EVSYS.CH0MUX   = EVSYS_CHMUX_TCD0_OVF_gc;  // event channel 0 is overflow timer
51 }
```

De initialisatiefunctie `init_dma` van de DMA stuurt bij een trigger één waarde uit de opzoektabel naar de DAC. De DMA is daarom op regel 22 ingesteld op *single shot* en op regel 23 op een *burst*-lengte van twee bytes omdat de waarden uit de opzoektabel twee bytes groot zijn.

Het adres van de bron is het adres van de opzoektabel en het adres van de bestemming is het adres van het dataregister van kanaal 0 van de DAC. Regel 28 geeft aan dat de DMA wordt getriggerd door kanaal 0 van het eventnetwerk. Het aantal bytes dat verstuurd wordt, is het aantal bytes van de opzoektabel.

Op regel 21 is de *repeat mode* ingesteld. De DMA gaat dan als alle bytes uit de opzoektabel verstuurd zijn, weer terug naar het begin van de tabel. Het effect is dat de DAC steeds opnieuw een sinus genereert en dat de uitgang van de DAC een periodieke sinus produceert.

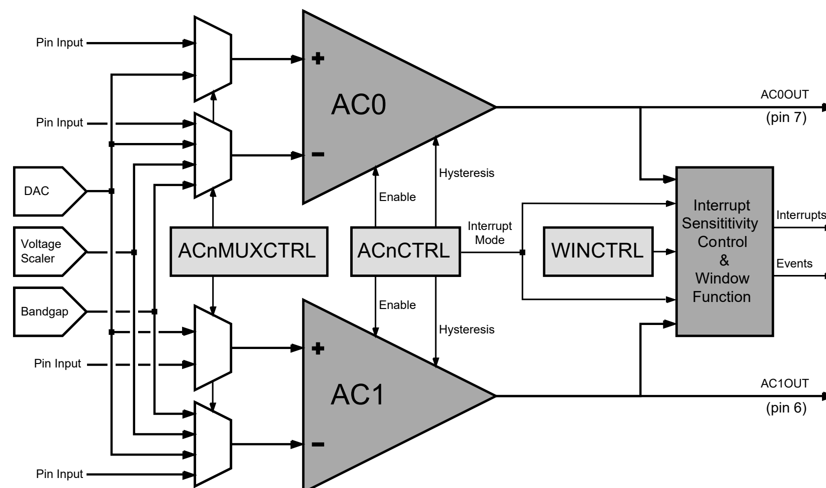
23.3 Analoge comparator

De Xmega beschikt naast een ADC en een DAC ook over meerdere analoge comparatoren. 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 op de negatieve ingang, is de uitgang hoog.

Figuur 23.7 : 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 23.8 : Blokschema met twee analoge comparatoren bij de Xmega. De vier multiplexers verbinden de ingangen van de comparatoren met externe aansluitingen of met een van de interne signalen. De uitgangen van beide comparatoren zijn vaste aansluitingen Comparator `0` is aangesloten op pin 7 en comparator 1 op pin 6. Beide uitgangen zijn aangesloten op het blok dat de interrupts genereert.

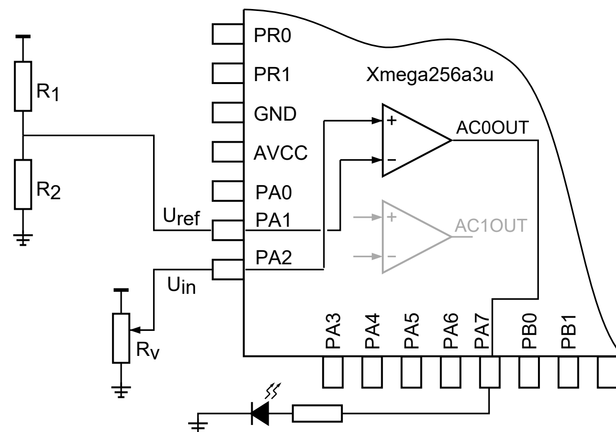
In figuur 23.8 staat het blokschema van het analoge comparatorblok. Ieder blok heeft twee analoge comparatoren. De Xmega256a3u heeft twee van deze comparatorblokken, namelijk één blok genaamd `ACA` bij poort A en één blok genaamd `ACB` bij poort B. Deze Xmega heeft dus vier comparatoren.

De registers `AC0MUXCTRL` en `AC1MUXCTRL` van het comparatorblok bepalen met welke aansluitingen de positieve en de negatieve ingang van de comparatoren verbonden zijn. Voor de positieve ingang van de comparator kan dat pin `0` tot en met 6 zijn of de interne uitgang van de DAC. Voor de negatieve ingang van de comparator is dat pin `0`, 1, 3, 5 of 7, de interne uitgang van de DAC, de bandgap-referentie van 1,1 V of een gedeelde spanning. Tabel 23.1 geeft een overzicht.

Tabel 23.1 : De aansluitingen bij de analoge comparatoren bij de Xmega.

bits	positieve ingang comparator		negatieve ingang comparator	
	groepsconfiguratie	ingangssignaal	groepsconfiguratie	ingangssignaal
000	AC_MUXPOS_PIN0_gc	Pin 0	AC_MUXNEG_PIN0_gc	Pin 0
001	AC_MUXPOS_PIN1_gc	Pin 1	AC_MUXNEG_PIN1_gc	Pin 1
010	AC_MUXPOS_PIN2_gc	Pin 2	AC_MUXNEG_PIN3_gc	Pin 3
011	AC_MUXPOS_PIN3_gc	Pin 3	AC_MUXNEG_PIN5_gc	Pin 5
100	AC_MUXPOS_PIN4_gc	Pin 4	AC_MUXNEG_PIN7_gc	Pin 7
101	AC_MUXPOS_PIN5_gc	Pin 5	AC_MUXNEG_DAC_gc	DAC output
110	AC_MUXPOS_PIN6_gc	Pin 6	AC_MUXNEG_BANDGAP_gc	Band gap
111	AC_MUXPOS_DAC_gc	DAC output	AC_MUXNEG_SCALER_gc	Voltage Scaler

De uitgang van de comparator 0 is verbonden met pin 7 en de uitgang van de comparator 1 is verbonden met pin 6. In veel gevallen wordt er geen uitgang gebruikt. In plaats hiervan reageert het programma of een van de andere perifere blokken op een interrupt of op een event, die de comparator genereert.



Figuur 23.9 : Schema voor demonstratie analoge comparator. De referentiespanning U_{ref} is de spanningsdeling van de weerstanden R_1 en R_2. Het ingangssignaal U_{in} kan worden ingesteld met de potentiometer R_v. De led is aan als het ingangssignaal groter is dan de referentiespanning, anders is de led uit.

Voorbeeld met analoge comparator

Figuur 23.9 toont een spanningsdeler met twee weerstanden R_1 en R_2 voor de referentiespanning U_{ref}. De ingang U_{in} is aangesloten op een potentiometer. Het voorbeeldprogramma staat in code 23.11. De led is aangesloten op de uitgang van de comparator. Als U_{in} groter is dan de referentiespanning is de led aan, anders is de led uit.

De initialisatiefunctie `ac_init` voor de analoge comparator uit code 23.11 zet comparator 0 aan, sluit pin 2 aan op de positieve ingang van comparator 0 en pin 1 op de negatieve ingang en activeert de uitgang van de comparator.

Code 23.11 : **Minimaal voorbeeld met analoge comparator.**

```
 1  #include <avr/io.h>
 2
 3  void init_ac(void)
 4  {
 5    PORTA.DIRCLR   = PIN2_bm | PIN1_bm;    // PA2 and PA1 are inputs
 6    ACA.AC0CTRL    = AC_ENABLE_bm;         // enable comparator
 7    ACA.AC0MUXCTRL = AC_MUXPOS_PIN2_gc |   // pin2 is + input
 8                     AC_MUXNEG_PIN1_gc;    // pin1 is - input
 9    ACA.CTRLA      = AC_AC0OUT_bm;         // output AC0 enabled
10  }
11
12  int main(void)
13  {
14    init_ac();
15
16    while (1) ;                  // do nothing
17  }
```

Voorbeeld analoge comparator met interruptfunctie

Tabel 23.2 : Instellingen voor de interrupt van de analoge comparator.

INTMODE	Betekenis
00	BOTHEDGES
01	-
10	FALLING
11	RISING

In code 23.13 staat een variant op code 23.11 met een interruptfunctie en een gedeelde spanning als referentie. De initialisatiefunctie staat in code 23.12. De gedeelde spanning hangt af van de schaalfactor `SCALER` en de voedingsspanning `VCC`. De referentiespanning V_{REF} is dan:

$$V_{\mathrm{REF}} = \frac{\texttt{SCALER + 1}}{64}\ \texttt{VCC} \tag{23.3}$$

De schaalfactor is een 6-bits getal dat in register `CTRLB` staat. In dit voorbeeld is de schaalfactor 38. Voor een `VCC` van 3,3 V is de referentiespanning dan 2,01 V.

Code 23.12 : **Initialisatie comparator voor een interrupt en een gedeelde spanning als referentie.**

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3
 4  void init_ac(void)
 5  {
 6    PORTA.DIRCLR   =  PIN2_bm;                   // PA2 is input
 7    ACA.AC0CTRL    =  AC_INTMODE_BOTHEDGES_gc |  // interrupt at both edges
 8                      AC_INTLVL_LO_gc         |  // interrupt level low
 9                      AC_ENABLE_bm;              // enable comparator
10    ACA.AC0MUXCTRL =  AC_MUXPOS_PIN2_gc       |  // pin2 is + input
11                      AC_MUXNEG_SCALER_gc;       // voltage scaler is - input
12    ACA.CTRLB      =  38;                        // VREF = (SCALER + 1)/64 * VCC = 2.01 V
13  }
```

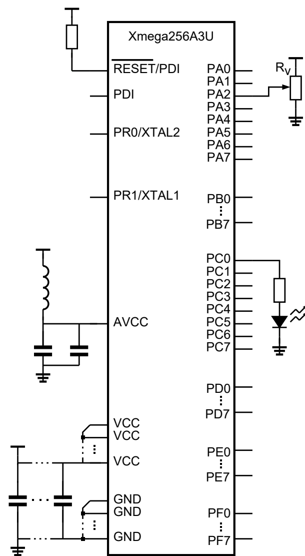
De initialisatiefunctie stelt op regel 7 de interruptmodus in op beide flanken. Iedere keer als het ingangssignaal van de comparator boven of onder de referentiespanning komt, is er een interrupt. Tabel 23.2 geeft de drie instelmogelijkheden voor de interrupt.

Pin 2 is aangesloten op de positieve ingang van de comparator en de negatieve ingang is verbonden met de *voltage scaler*. De uitgang van de comparator wordt alleen gebruikt om een interrupt te genereren. Register `CTRLA` is niet ingesteld, zodat de uitgang van de comparator niet is aangesloten op pin 7.

Code 23.13 : Analoge comparator met een interrupt en een gedeelde spanning als referentie.

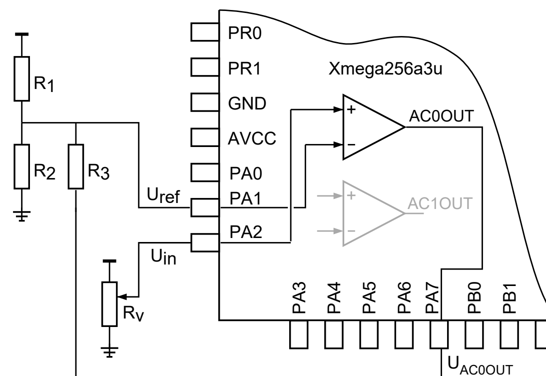
```
15  int main(void)
16  {
17    PORTC.DIRSET = PIN0_bm;
18    PORTC.OUTCLR = PIN0_bm;      // turn led off
19    init_ac();
20    PMIC.CTRL   |= PMIC_LOLVLEN_bm;
21    sei();
22
23    while (1) ;                  // do nothing
24  }
```

```
26  ISR(ACA_AC0_vect)
27  {
28    if ( ACA.STATUS & AC_AC0STATE_bm ) {
29      PORTC.OUTSET = PIN0_bm;     // led is on
30    } else {
31      PORTC.OUTCLR = PIN0_bm;     // led is off
32    }
33  }
```



Figuur 23.10 : Schema voorbeeld analoge comparator met interrupt en een gedeelde spanning als referentie.

Code 23.13 bevat de interruptfunctie en de hoofdroutine van het voorbeeld met de interrupt en de gedeelde spanning als referentie. Figuur 23.10 geeft het schema. Na de initialisaties van regel 17 tot en met 21 komt de hoofdroutine `main` in een oneindige lus, waarin niets gebeurt. Als de spanning op `PA2` hoger wordt dan referentiespanning, is er een interrupt en wordt de interruptfunctie uitgevoerd. Uitgang `PC0` wordt hoog en de led gaat aan. Zolang de spanning hoog blijft gebeurt er niets. Als de spanning op `PA2` lager wordt dan de referentiespanning, is er weer een interrupt. De uitgang `PC0` wordt dan laag en de led gaat uit.

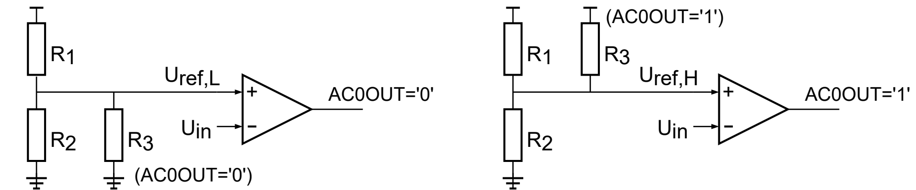


Figuur 23.11 : Deel van schema met analoge comparator met positieve terugkoppeling. De referentiespanning U_{ref} hangt af van R_1, R_2 en R_3 en van U_{AC0OUT}.

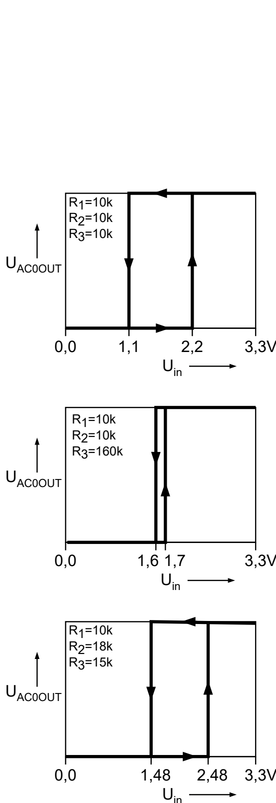
Hysterese

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 F.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 een hogere spanning ligt en voor een afnemende waarde bij een lagere spanning.

In figuur 23.11 is de uitgang van de comparator softwarematig verbonden met pin 7 van poort A en via een weerstand R_3 verbonden met de aansluiting voor de referentiespanning. De referentiespanning U_{ref} hangt af van de waarde van de weerstanden R_1, R_2 en R_3 en de waarde van de uitgangsspanning U_{AC0OUT}. Deze spanning is gelijk aan U_{cc} als `AC0OUT` hoog is en 0 V als `AC0OUT` laag is. Hierdoor zijn er twee verschillende configuraties met twee verschillende referentiespanningen. Figuur 23.12 toont de twee configuraties.



Figuur 23.12 : De twee configuraties met de terugkoppelweerstand R_3. Links staat de configuratie als `AC0OUT` laagg is en rechts die als `AC0OUT` hoog is.



Figuur 23.13 : Drie voorbeelden met hysterese. R_1, R_2 en R_3 bepalen de hysteresespanningen $U_{ref,H}$ en $U_{ref,L}$.

Als U_{in} laag is, is U_{AC0OUT} laag en heeft de referentiespanning de lage waarde $U_{ref,L}$ en als U_{in} hoog is, heeft de referentiespanning de hoge waarde $U_{ref,H}$. Voor deze referentiespanningen geldt:

$$U_{ref,L} = \frac{R_2R_3}{R_1R_2 + R_2R_3 + R_1R_3} U_{cc} \tag{23.4}$$

$$U_{ref,H} = \frac{R_1R_2 + R_2R_3}{R_1R_2 + R_2R_3 + R_1R_3} U_{cc} \tag{23.5}$$

Als U_{cc} gelijk is aan 3,3 V en R_1, R_2 en R_3 alle drie 10 kΩ zijn, dan is $U_{ref,L}$ 1,1 V en $U_{ref,H}$ 2,2 V. Voor grotere waarden van R_3 wordt de hystereselus smaller. Een weerstand van 160 kΩ geeft een $U_{ref,L}$ van 1,6 V en een $U_{ref,H}$ van 1,7 V. In figuur 23.13 staan de hystereselussen van deze voorbeelden.

Hysterese bij de Xmega

De analoge comparator van de Xmega kent standaard de mogelijkheid om hysterese toe te voegen en heeft een *high speed mode*. Wanneer het `HSMODE`-bit hoog is, is de propagatietijd kleiner, maar is het verbruikte vermogen groter. Met de `HYSMODE`-bits kan een kleine en een grote hysterese worden toegevoegd. In tabel 23.3 staan de bits en de waarden van de hysterese voor Xmega256a3u. De grootte van de hysterese is kleiner voor de high-speed-modus.

Tabel 23.3 : De hysterese modus bij de analoge comparator van de Xmega256a3u.

bits	groepsconfiguratie	hysterese	
		low speed	high speed
00	AC_HYSMODE_NO_gc	–	–
01	AC_HYSMODE_SMALL_gc	30 mV	13 mV
10	AC_HYSMODE_LARGE_gc	60 mV	30 mV
11	–	–	–

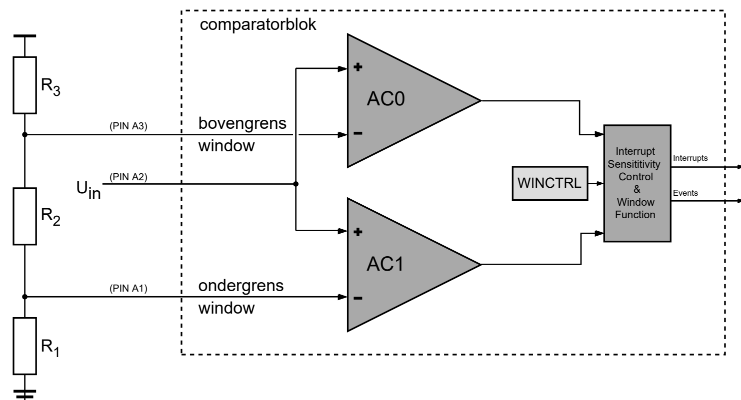
Code 23.14 : Initialisatie van de analoge comparator met een grote hysterese en een gedeelde spanning als referentie.

```
 4  void init_ac(void)
 5  {
 6    PORTA.DIRCLR    =  PIN2_bm;
 7    ACA.AC0CTRL     =  AC_INTMODE_BOTHEDGES_gc |
 8                       AC_INTLVL_LO_gc         |
 9                       AC_HYSMODE_LARGE_gc     |       // hysteresis large
10                       AC_ENABLE_bm;
11    ACA.AC0MUXCTRL  =  AC_MUXPOS_PIN2_gc       |
12                       AC_MUXNEG_SCALER_gc;
13    ACA.CTRLB       =  38;        // VREF = (SCALER + 1)/64 * VCC = 2.01 V
14  }
```

In code 23.14 is aan de initialisatiefunctie van code 23.12 de grote hysterese toegevoegd. Met deze initialisatiefunctie verandert de uitgang bij de opgaande flank bij ongeveer 2,04 V en bij de neergaande flank bij ongeveer 1,97 V.

De windowmodus

In één comparatorblok zitten bij de Xmega altijd twee analoge comparatoren. Deze twee comparatoren kunnen samen een zogenoemd *window* of venster definiëren. Het ingangssignaal is verbonden met beide comparatoren en iedere comparator heeft een eigen referentie. Het signaal kan dan boven het hoogste, onder het laagste of tussen beide referentieniveaus liggen. Figuur 23.14 geeft een voorbeeld met drie externe weerstanden, die de boven- en ondergrens bepalen.



Figuur 23.14 : Het comparatorblok voor de windowmodus. De spanningsdeler met de drie weerstanden definieert de referentiespanningen voor de bovengrens en de ondergrens van het window.

Als de voedingsspanning 3,3V en de weerstanden R_1, R_2 en R_3 respectievelijk 4k7, 4k7 en 10k zijn, is de ondergrens 0,8 en de bovengrens 1,6 V. Het ingangssignaal is verbonden met pin A2, de referentie van comparator 1 met pin A1 en de referentie van comparator 3 met pin A3. In code 23.15 staat de initialisatiefunctie voor deze configuratie.

Code 23.15 : De initialisatie van het comparatorblok voor de windowmodus.

```
 4  void init_ac(void)
 5  {
 6    ACA.WINCTRL    =  AC_WEN_bm             |            // window enable
 7                      AC_WINTMODE_INSIDE_gc |            // interrupt on inside
 8                      AC_WINTLVL_LO_gc;                  // interrupt level low
 9    ACA.AC0CTRL    =  AC_ENABLE_bm;
10    ACA.AC1CTRL    =  AC_ENABLE_bm;
11    ACA.AC0MUXCTRL =  AC_MUXPOS_PIN2_gc | AC_MUXNEG_PIN3_gc;
12    ACA.AC1MUXCTRL =  AC_MUXPOS_PIN2_gc | AC_MUXNEG_PIN1_gc;
13  }
```

Tabel 23.4 : De interrupts voor de windowmodus.

WINTMODE	Betekenis
00	ABOVE
01	INSIDE
10	BELOW
11	OUTSIDE

Regel 6 zet het windowcontrolblok aan en stelt de interrupt voor de windowmodus in op `INSIDE`, zie ook tabel 23.4. Als het ingangssignaal tussen de twee referentieniveaus komt, is er een interrupt en wordt de interruptiefunctie uitgevoerd.

De opstelling uit figuur 23.10 en de functie `main` uit code 23.13 kunnen nu gebruikt worden om de led aan te zetten als het ingangssignaal binnen het venster komt en uit te zetten als het weer buiten het venster komt. Wel moet de interruptfunctie van comparator `0` dan vervangen worden door de interruptfunctie voor de windowmodus van code 23.16.

Code 23.16 : De interruptfunctie voor de windowmodus.

```
26  ISR(ACA_ACW_vect)
27  {
28    if ( (ACA.STATUS & AC_WSTATE_gm) == AC_WSTATE_INSIDE_gc ) {               // test if it is inside
29      ACA.WINCTRL = (ACA.WINCTRL & ~AC_WINTMODE_gm) | AC_WINTMODE_OUTSIDE_gc; //   change to outside
30      PORTC.OUTSET  = PIN0_bm;                                                //   turn led on
31    } else {                                                                  // it must be outside
32      ACA.WINCTRL = (ACA.WINCTRL & ~AC_WINTMODE_gm) | AC_WINTMODE_INSIDE_gc;  //   change to inside
33      PORTC.OUTCLR  = PIN0_bm;                                                //   turn led off
34    }
35  }
```

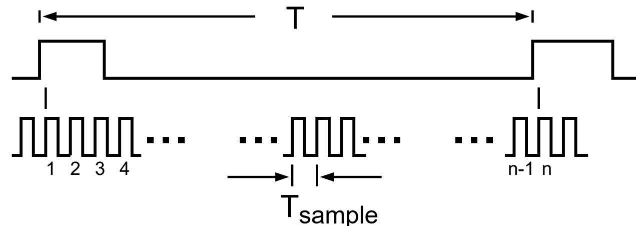
Tabel 23.5 : De statusbits voor de windowmodus.

WSTATE	Betekenis
00	ABOVE
01	INSIDE
10	BELOW
11	OUTSIDE

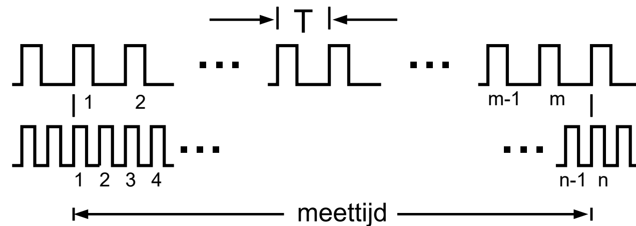
Met de instelling van de `init_ac` uit code 23.15 wordt de interruptfunctie alleen uitgevoerd als het ingangssignaal binnen de grenzen van het venster komt. Er is dan geen interrupt als het ingangssignaal weer buiten de grenzen komt. Om de led weer uit te kunnen zetten is de constructie uit code 23.16 nodig. De interruptfunctie test of de functie is aangeroepen omdat het signaal het venster binnenkomt of omdat het signaal het venster verlaat. Regel 28 vergelijkt de statusbits van de windowmodus, zie ook tabel 23.5, met de groepsconfiguratie van `AC_WSTATE_INSIDE_gc`. Als het signaal het venster binnenkomt, gaat de led aan en wijzigt de windowmodus in `OUTSIDE`. Als het signaal het venster verlaat, verandert de windowmodus wordt `INSIDE` en de led gaat uit.

23.4 Input capture

De timer/counters van de Xmega zijn ook geschikt om aan signalen te meten. In de inputcapture-modus kan de timer/counter reageren op gebeurtenissen en het tijdstip van deze gebeurtenissen vastleggen. Met de inputcapture-modus is het mogelijk de periodetijd van een periodiek signaal te bepalen of de pulsbreedte van een puls te meten.



Figuur 23.15 : De periodetijdmeting. De periodetijd T is gelijk aan nT_{sample} als n het aantal samples is dat in T past.



Figuur 23.16 : De frequentiemeting. De meettijd is gelijk aan nT_{sample} en gelijk aan mT. De periodetijd T van het signaal is hier m/T_{meettijd}.

Het meten van de periodetijd of een frequentie van een signaal kan op twee manieren: met de periodetijdmeting en met de frequentiemeting. De periodetijdmeting uit figuur 23.15 telt het aantal *samples* gedurende één periodetijd van het signaal. Voor snelle signalen is deze methode onnauwkeurig. De frequentiemeting uit figuur 23.16 telt hoeveel perioden van het signaal in een vooraf gekozen meettijd passen. Voor laagfrequente signalen is deze methode traag.

De periodetijdmeting wordt zo genoemd omdat de periode T evenredig is met het aantal samples n. Als T_{sample} de sampletijd is, is de periodetijd:

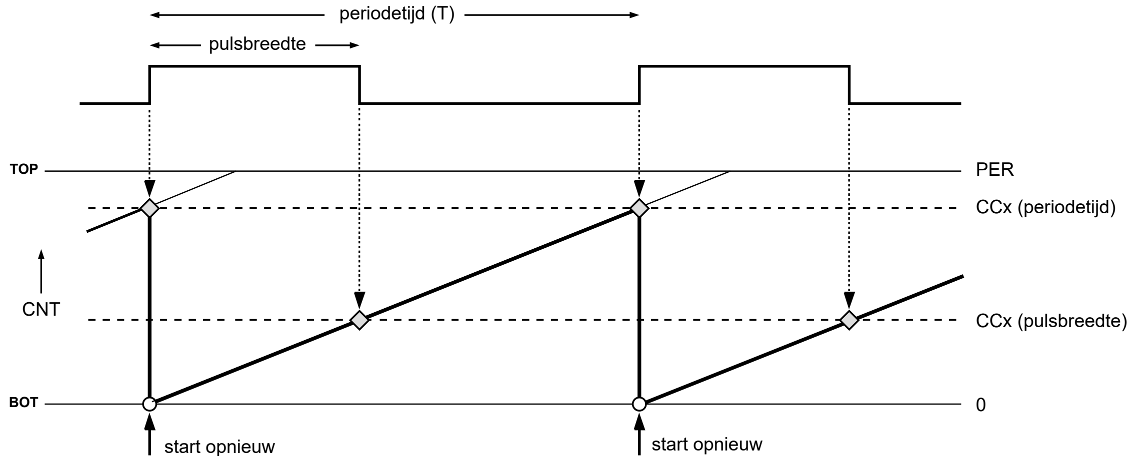
$$T = n\,T_{\text{sample}}$$

De frequentiemeting wordt zo genoemd omdat de frequentie f evenredig is met de frequentie van de meettijd. Als f_{meettijd} de reciproke van de meettijd is en m het aantal getelde pulsen, geldt voor de frequentie:

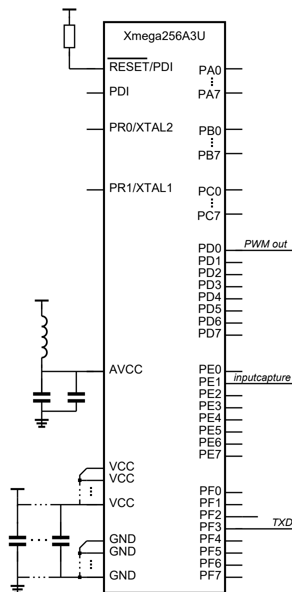
$$f = m\,f_{\text{meettijd}}$$

Voor de frequentiemeting zijn twee tellers nodig, één voor de definitie van de meettijd en één voor het tellen van het aantal perioden. Bij periodetijdmeting is slechts één teller nodig.

De inputcapture-modus van de timer/counter `0` en 1 van de Xmega gebruikt de periodetijdmeting voor het bepalen van de periodetijd en voor het bepalen van een pulsbreedte. Uit de periodetijd volgt ook de frequentie van het signaal.



Figuur 23.17 : Het meten van de periodetijd en de pulsbreedte met inputcapture-modus. Als functie van de tijd staat bovenaan het ingangssignaal en onderaan waarde `CNT` van de teller. Bij iedere flank van de ingang treedt een interrupt op. Bij de opgaande flank bevat `CCx` een waarde die overeenkomt met de periodetijd en bij de neergaande flank is een waarde die overeenkomt met de pulsbreedte van het ingangssignaal. Bij de opgaande flank herstart de gebruiker de meting.



Figuur 23.18 : Schema voor inputcapture-modus. De PWM-uitgang van pin D0 is aangesloten op de inputcapture van pin E1.

Figuur 23.17 toont voor de inputcapture-modus het ingangssignaal en de waarde `CNT` van de teller. Iedere keer als er een flank is, is er een interrupt. Het register `CCx` bevat bij de neergaande flank een waarde, die overeenkomt met de pulsbreedte, en bij de opgaande flank een waarde, die overeenkomt met de periodetijd. Bij de opgaande flank wordt de teller weer nul gemaakt.

In code 23.17 en code 23.18 staat een deel van een programma dat de periodetijd en de pulsbreedte van een signaal meet. In figuur 23.18 staat het schema. Pin 1 van poort E is de *capture*-ingang. Pin `0` van poort D genereert een PWM-signaal en is aangesloten op de *capture*-ingang. Dit programma genereert dus ook het testsignaal dat gemeten wordt.

De functie `init_pwm` uit code 23.17 stelt het PWM-signaal in. De periodetijd en de duty-cycle van het signaal volgen uit formule 22.3 en 22.4. In dit voorbeeld is de periodetijd 128 ms en de duty-cycle is 80%.

De functie `init_inputcapture` uit code 23.17 stelt de capture-ingang in. In dit voorbeeld is de capture gevoelig voor de flanken van het ingangssignaal. Hiervoor worden de `PORT`-instellingen gebruikt en via het eventnetwerk wordt het captureblok van de timer getriggerd. De complete configuratie voor de inputcapture, bestaat uit drie delen: de flankdetectie van het ingangssignaal, het eventnetwerk en de instelling van de timer/counter. De fysieke verbinding tussen de onderdelen is in figuur 23.19 getekend.

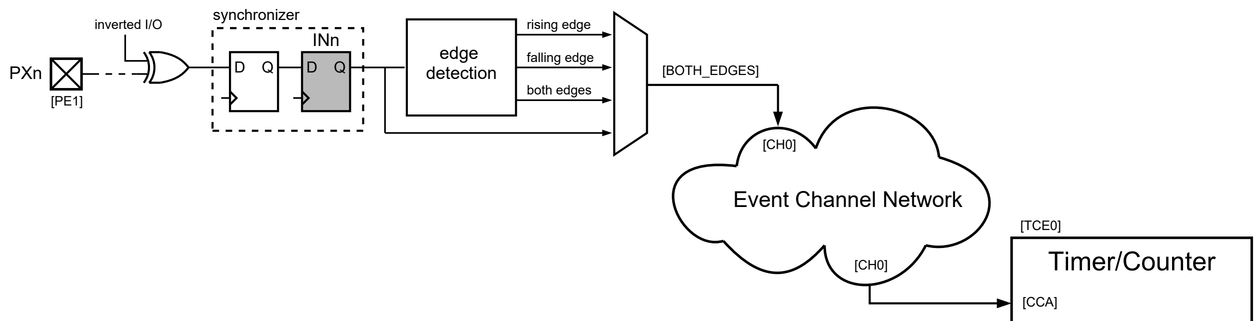
De functie `init_inputcapture` uit code 23.17 selecteert op regel `20` en `21` pin 1 van poort E als ingang en maakt deze ingang gevoelig voor beide flanken. Regel 22 verbindt de uitgang van de flankdetectie met kanaal `0` van het eventnetwerk. Op regel 23 wordt timer/counter `0` van poort E ingesteld op de inputcapture-modus en verbonden met eventkanaal `0`. De functie stelt vervolgens de timer in op de normale modus, selecteert het compare/capture-blok A, kiest een prescaling van 256, selecteert het lage interruptniveau en geeft `PER` de waarde `0x7FFF`.

Code 23.17 : De initialisatiefuncties init_pwm en init_inputcapture voor het meten van de periodetijd en de pulsbreedte.

```
 7  // De regels 1 tot en met 7 zijn identiek met die uit code 19.19
 8
 9  void init_pwm(void)
10  {
11    PORTD.DIRSET  = PIN0_bm;
12    TCD0.CTRLB    = TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
13    TCD0.CTRLA    = TC_CLKSEL_DIV256_gc;
14    TCD0.PER      = 9999;                                  // Tper = 128 ms  @ 2 MHz
15    TCD0.CCA      = 8000;                                  // duty-cycle 80%
16  }
17
18  void init_inputcapture(void)
19  {
20    PORTE.PIN1CTRL = PORT_ISC_BOTHEDGES_gc;             // both edges
21    PORTE.DIRCLR   = PIN1_bm;                           // Pin 1 is input
22    EVSYS.CH0MUX   = EVSYS_CHMUX_PORTE_PIN1_gc;         // Select PE1 as input to Event Channel 0
23    TCE0.CTRLD     = TC_EVACT_CAPT_gc | TC_EVSEL_CH0_gc; // Event capture from Event Channel 0
24    TCE0.CTRLB     = TC0_CCAEN_bm | TC_WGMODE_NORMAL_gc; // Enable Capture Channel A
25    TCE0.CTRLA     = TC_CLKSEL_DIV256_gc;               // Start timer by selecting a clock source
26    TCE0.INTCTRLB  = TC_CCAINTLVL_LO_gc;                // Set Interrupt level Capture Channel A
27    TCE0.PER       = 0x7FFF;                            // MSB is low, so MSB of CCA holds edge
28  }
```

In dit voorbeeld is compare/capture-blok A gebruikt. In principe is het ook mogelijk één van de andere of zelfs meerdere blokken te gebruiken. Omdat iedere timer/counter maar één teller heeft is het gebruik van meer ingangen meestal niet zinvol en bij één ingang wordt het eventkanaal automatisch aangesloten op compare/capture-blok A.

De meest significante bit van register `PER` wordt gebruikt voor de detectie van een opgaande of neergaande flank. Als deze bit bij de initialisatie nul is, bevat deze bit na de capture een `0` als het signaal laag en een 1 als het signaal hoog is. De bit is dus laag bij een neergaande flank en hoog bij een opgaande flank. Bij de inputcapture-modus met beide flanken is dit nuttig. Het bereik van de capture kan dan maximaal `0x7FFF` zijn. Bij andere modi zoals de frequentie-capture is dit mechanisme niet nodig en kan `PER` de waarde `0xFFFF` krijgen.



Figuur 23.19 : De aansluiting aan de capture-ingang van de timer. In het voorbeeld is aansluitpin `PE1` ingesteld op de flankdetectie `BOTH_EDGES` en is dit eventsignaal via eventkanaal `0` aangesloten op het capture/compare-blok `CCA` van timer/counter E0.

Code 23.18 : Deel van programma om de periodetijd en de pulsbreedte te meten met de inputcapture-modus.

```
30 volatile uint8_t  newCapture = 0;
31 volatile uint16_t period;
32 volatile uint16_t pulseWidth;
33
34 ISR(TCE0_CCA_vect)
35 {
36   uint16_t captureValue = TCE0.CCA;
37
38   if ( captureValue & 0x8000 ) {          // MSB is high, it is a rising edge
39     TCE0.CTRLFSET = TC_CMD_RESTART_gc;    // restart measurement
40     period = captureValue & 0x7FFF;       // assign measured value
41     newCapture = 1;                       // set newCapture flag
42   } else {                                // MSB is low, it is a falling edge
43     pulseWidth = captureValue;            // assign measured value
44   }
45 }
46
47 int main(void)
48 {
49   init_stream(F_CPU);
50   init_pwm();
51   init_inputcapture();
52   PMIC.CTRL |= PMIC_LOLVLEN_bm;           // uart, inputcapture
53   sei();
54
55   while (1) {
56     while (! newCapture) ;
57     printf("%u %u\n", period, pulseWidth);
58     newCapture = 0;                      // clear newCapture flag
59   }
60 }
```

Bij de declaratie moet `newCapture` ook **`volatile`** zijn. Het hoofdprogramma maakt deze variabele altijd nul. Zonder **`volatile`** behandelt de compiler deze variabele als een constante.

In code 23.18 staat de interruptfunctie voor de capture en het hoofdprogramma voor het meten van periodetijd `period` en de pulsbreedte `pulseWidth`. Deze globale variabelen zijn **`volatile`** omdat de interrupt deze variabelen verandert en de `main` deze alleen gebruikt. Zonder **`volatile`** zou de compiler deze variabelen als constanten behandelen.

De interruptiefunctie kent de inhoud van register `CCA` toe aan de lokale variabele `captureValue`. Op regel 38 test de functie of de meest significante bit van `captureValue` hoog is. Als dat zo is, is de opgaande flank gevonden en bevatten de andere vijftien bits de periodetijd. Bij de toekenning aan `period` wordt de meest significante bit gemaskeerd. Vervolgens wordt de meting herstart en wordt de vlag `newCapture` hoog gemaakt. Als de meest significante bit van `captureValue` laag is, is er een neergaande flank en bevat het de pulsbreedte.

Het hoofdprogramma wacht totdat de vlag `newCapture` hoog is, drukt daarna het resultaat af en maakt de vlag weer laag. Het programma verstuurt voortdurend de gemeten periodetijd 10000 en de pulsbreedte 8000 via de UART. Met een prescaling van 256 en de klokfrequentie 2 MHz volgt daaruit dat de periodetijd 128 ms is en dat de duty-cycle 80% is.

Als de prescaling van de inputcapture kleiner is, zullen de gemeten waarden groter zijn. Bij een prescaling van 64 zou het resultaat voor de periodetijd 40000 en voor de pulsbreedte 32000 moeten zijn. In werkelijkheid is dat echter 7232 en 32000. De grootste waarde, die met vijftien bits gemeten kan worden, is immers 32767. Een kleine prescaling beperkt het meetbereik. Daarentegen beperkt een te grote prescaling de nauwkeurigheid. Het meetbereik kan worden vergroot door met twee 16-bits timer/counters een 32-bits inputcapture te maken of door het aantal overflows bij de detectie te tellen en dit aantal mee te nemen bij de berekening van de periodetijd en de pulsbreedte.

Een prescaling kleiner dan 64 zorgt ervoor dat er te kleine waarden worden gemeten. Na een interrupt zijn er enige tientallen klokslagen nodig voor de interruptfunctie de meting opnieuw start.

Meetbereik van de inputcapture vergroten met overflowdetectie

In code 23.19 en code 23.20 staat een deel van het programma uit code 23.17 en 23.18 dat is uitgebreid met een overflowdetectie. De regels met commentaar zijn aangepast. In code 23.20 is de initialisatiefunctie nu ook gevoelig voor een overflow. De bijbehorende interruptfunctie staat in code 23.19 en verhoogt de variabele `numberOfOverflows` met één.

Code 23.19 : Deel van programma om de periodetijd en de pulsbreedte te meten met overflowdetectie.

```
31  volatile uint8_t  newCapture = 0;
32  volatile uint32_t period;                         // Changed to 32-bits
33  volatile uint32_t pulseWidth;                     // Changed to 32-bits
34  volatile uint8_t  numberOfOverflows = 0;          // Integer for number of overflows
35
36  ISR(TCE0_OVF_vect)                                // Interruptroutine for overflow
37  {                                                 //
38    numberOfOverflows++;                            // Increment number of overflows
39  }                                                 //
40
41  ISR(TCE0_CCA_vect)
42  {
43    uint16_t captureValue = TCE0.CCA;
44
45    if ( captureValue & 0x8000 ) {
46      TCE0.CTRLFSET = TC_CMD_RESTART_gc;
47      period = (captureValue & 0x7FFF) + (0x8000UL * numberOfOverflows);  // With correction
48      numberOfOverflows = 0;                                              // Reset overflow number
49      newCapture = 1;
50    }  else {
51      pulseWidth = captureValue + (0x8000UL * numberOfOverflows);         // With correction
52    }
53  }
```

De variabelen `period` en `pulseWidth` zijn nu 32-bits en bij de toekenning wordt de overflowcorrectie er bijgeteld. Deze correctie is het aantal overflow maal `0x8000`. De suffix `UL` zorgt ervoor dat bij de berekening van de correctie 32-bits worden gebruikt. De functie `main` blijft ongewijzigd, alleen moeten de *format specifiers* bij de `printf`-functie `%ul` zijn in plaats van `%u` om geschikt te zijn voor 32-bits.

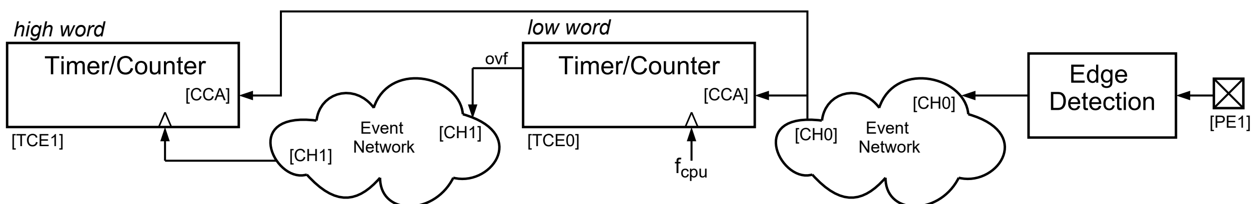
Code 23.20 : Initialisatiefunctie voor de overflowdetectie.

```
18 void init_inputcapture(void)
19 {
20   PORTE.PIN1CTRL = PORT_ISC_BOTHEDGES_gc;
21   PORTE.DIRCLR   = PIN1_bm;
22   EVSYS.CH0MUX   = EVSYS_CHMUX_PORTE_PIN1_gc;
23   TCE0.CTRLD     = TC_EVACT_CAPT_gc | TC_EVSEL_CH0_gc;
24   TCE0.CTRLB     = TC0_CCAEN_bm | TC_WGMODE_NORMAL_gc;
25   TCE0.CTRLA     = TC_CLKSEL_DIV64_gc;
26   TCE0.INTCTRLB  = TC_CCAINTLVL_LO_gc;
27   TCE0.INTCTRLA  = TC_OVFINTLVL_LO_gc;              // Set Interrupt level Overflow
28   TCE0.PER       = 0x7FFF;
29 }
```

Het programma met de overflowdetectie meet voor de periodetijd 40000 en voor de pulsbreedte 32000 gedeelde klokslagen. Als de initialisatiefunctie `init_pwm` een prescaling krijgt van 1024 in plaats van 256, is het testsignaal vier keer zo traag en zijn de gemeten waarden 160000 en 128000.

Inputcapture met een 32-bits timer/counter

Het meetbereik kan ook worden vergroot door met twee timer/counters een 32-bits inputcapture te maken. In figuur 23.20 staat de configuratie.



Figuur 23.20 : De configuratie van het eventnetwerk voor een 32-bits inputcapture. Pin `PE1` is via kanaal 0 van het netwerk aangesloten op het captureblok `CCA` van timer `TCE0`. De overflow van timer `TCE0` is via kanaal 1 van het netwerk verbonden met de klokingang van timer `TCE1`. Het `CCA`-register van timer `TCE1` bevat de meest significante bits en dat van timer `TCE0` de minst significante bits.

Het programma uit code 23.21 komt grotendeels overeen met het programma uit code 23.17 en 23.18. De functie `init_inputcapture` is uitgebreid met een tweede timer. De overflow van de eerste timer `TCE0` is via kanaal 1 aangesloten op de klokselectie van de tweede timer `TCE1`. Iedere overflow van de eerste timer verhoogt de tweede timer met één. De tweede timer bevat de meest significante bits en de eerste timer de minst significante bits. Beide timers zijn gevoelig voor de inputcapture, die via kanaal 0 verbonden is met pin 1 van poort E. Bij de tweede timer is het `EVDLY`-bit actief. Deze bit compenseert de propagatietijd van het overflowsignaal, door dit signaal één klokslag te vertragen.

De interruptfunctie heeft twee lokale 16-bits variabelen voor het hoge en het lage *word*. Als de meest significante bit van `captureValueL` hoog is, is de opgaande flank

Code 23.21 : Deel van programma om de periodetijd en de pulsbreedte te meten met een 32-bits timer/counter.

```
18 void init_inputcapture(void)
19 {
20   PORTE.PIN1CTRL = PORT_ISC_BOTHEDGES_gc;
21   PORTE.DIRCLR   = PIN1_bm;
22   EVSYS.CH0MUX   = EVSYS_CHMUX_PORTE_PIN1_gc;
23   TCE0.CTRLD     = TC_EVACT_CAPT_gc | TC_EVSEL_CH0_gc;
24   TCE0.CTRLB     = TC0_CCAEN_bm | TC_WGMODE_NORMAL_gc;
25   TCE0.CTRLA     = TC_CLKSEL_DIV64_gc;
26   TCE0.INTCTRLB  = TC_CCAINTLVL_LO_gc;
27   TCE0.PER       = 0x7FFF;
28   EVSYS.CH1MUX   = EVSYS_CHMUX_TCE0_OVF_gc;                       // Overflow TCEO input ch.0
29   TCE1.CTRLD     = TC_EVACT_CAPT_gc | TC_EVSEL_CH0_gc | TC1_EVDLY_bm; // Event capture for ch.0
30   TCE1.CTRLB     = TC1_CCAEN_bm | TC_WGMODE_NORMAL_gc;            // Enable Input Capture ch.A
31   TCE1.CTRLA     = TC_CLKSEL_EVCH1_gc;                            // Select ch.1 as clock source
32 }
33
34 volatile uint8_t  newCapture = 0;
35 volatile uint32_t period;                                             // Changed to 32-bits
36 volatile uint32_t pulseWidth;                                         // Changed to 32-bits
37
38 ISR(TCE0_CCA_vect)
39 {
40   uint16_t captureValueH = TCE1.CCA;                                  // Two capture values
41   uint16_t captureValueL = TCE0.CCA;                                  //
42
43   if ( captureValueL & 0x8000 ) {
44     TCE0.CTRLFSET = TC_CMD_RESTART_gc;                                // Restart both captures
45     TCE1.CTRLFSET = TC_CMD_RESTART_gc;                                //
46     period = ((uint32_t) captureValueH << 15) | (captureValueL  & 0x7FFF); // Calculate period
47     newCapture = 1;
48   } else {
49     pulseWidth = ((uint32_t) captureValueH << 15) | captureValueL;        // Calculate pulseWidth
50   }
51 }
```

gevonden. De capture herstart en de periode wordt gevonden door `captureValueH` 15 bits naar links te schuiven en toe te voegen aan de 15 minst significante bits van `captureValueL`. Als de minst significante bit van `captureValueL` laag is, is de neergaande flank gevonden en is de pulsbreedte ook bekend.

Frequentie-capture

Als alleen de periodetijd of de frequentie gemeten moet worden, kan de flankgevoeligheid in code 23.17 en code 23.21 op regel 20 gewijzigd worden in de stijgende of neergaande flank en kan in de interruptfunctie de test op de meest significante bit van de gemeten waarde vervallen.

Een andere methode is om in de plaats van de gewone capture de frequentie-capture te gebruiken. Bij deze capture reageert de capture alleen op de opgaande klokflank. In tabel 23.6 staat een overzicht met alle capture-instellingen. Deze paragraaf bespreekt alleen de gewone capture en de frequentie-capture.

Tabel 23.6 : Instellingen voor event actions.

EVACT	Groepsconfiguratie	Betekenis
000	OFF	geen
001	CAPT	gewone inputcapture
010	UPDOWN	op- en aftellen met extern bron
011	QDEC	kwadratuur decoding
100	RESTART	herstart waveform
101	FRQ	frequentie-capture
110	PW	pulsbreedte-capture
111	–	

Code 23.22 geeft een initialisatiefunctie die de frequentie-capture gebruikt. Regel 22 stelt de frequentie-capture in. De toewijzing aan `PIN1CTRL` is weggelaten. Standaard is de flankgevoeligheid van een pin ingesteld op beide flanken. De frequentie-capture reageert daarentegen alleen de opgaande flank. De toewijzing aan `PER` is ook weggelaten. Standaard krijgt `PER` de waarde `0xFFFF`. De meest significante bit van `CCA` is nu niet in gebruik voor de flankdetectie.

Code 23.22 : De initialisatiefunctie voor de frequentie-capture.

```
18 void init_inputcapture(void)
19 {
20   PORTE.DIRCLR   = PIN1_bm;
21   EVSYS.CH0MUX   = EVSYS_CHMUX_PORTE_PIN1_gc;
22   TCE0.CTRLD     = TC_EVACT_FRQ_gc | TC_EVSEL_CH0_gc;
23   TCE0.CTRLB     = TC0_CCAEN_bm | TC_WGMODE_NORMAL_gc;
24   TCE0.CTRLA     = TC_CLKSEL_DIV256_gc;
25   TCE0.INTCTRLB  = TC_CCAINTLVL_LO_gc;
26 }
```

Code 23.23 : De interruptfunctie voor de frequentie-capture.

```
28 volatile uint8_t  newCapture = 0;
29 volatile uint16_t period;
30
31 ISR(TCE0_CCA_vect)
32 {
33   period = TCE0.CCA;
34   TCE0.CTRLFSET = TC_CMD_RESTART_gc;
35   newCapture = 1;
36 }
```

Omdat de frequentie-capture alleen de periodetijd meet, bevat de interruptfunctie uit code 23.23 geen test meer en krijgt `period` direct de waarde van `CCA`.

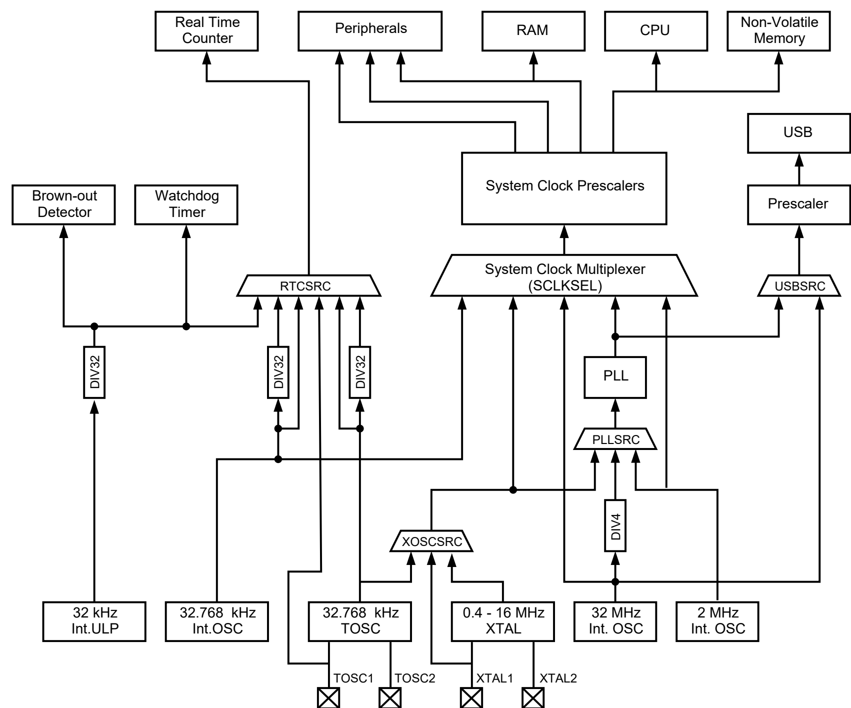
23.5 Het kloksysteem van de Xmega

De Xmega heeft een uitgebreid kloksysteem met een groot aantal instellingen. Figuur 23.21 geeft een overzicht. Er zijn meerdere klokbronnen mogelijk. De Xmega heeft drie interne oscillatoren van 2 MHz, 32 MHz en 32 kHz. Via de aansluitingen `XTALn` kan de microcontroller ook op een externe klok of een extern kristal worden aangesloten. Met een PLL, *phase locked loop* en klokdelers zijn veel verschillende klokfrequenties mogelijk. De realtime-counter kan via de pinnen `TOSCn` aangesloten worden met een extern 32,768 kHz kristal.

De realtime-counter is een teller, die speciaal bedoeld is voor een nauwkeurige tijdbepaling. Met een kristal van 32,768 kHz kan een signaal gegenereerd worden met periodetijd van één seconde.

De interne oscillator van 32 kHz is in de fabriek gekalibreerd en de frequentie ligt dichtbij de waarde van 32,768 kHz. De nauwkeurigheid en stabiliteit van de interne oscillatoren 2 MHz en 32 MHz zijn met behulp van een DFLL, *digital frequency locked loop* aanzienlijk te verbeteren.

Bij sommige Xmega's zijn de aansluitingen `XTALn` en `TOSCn` identiek. Het kristal dat is aangesloten kan dan gebruikt worden voor de microcontroller of voor de realtime-counter.



Figuur 23.21 : **Overzicht van het kloksysteem van de Xmega.** Onderaan staan de verschillende klokbronnen en bovenaan de zijn kloklijnen naar de diverse onderdelen getekend. De DFLL's voor de verbetering van de interne oscillatoren ontbreken in dit schema.

De Xmega start altijd met de interne oscillator van 2 MHz als klokbron. Na het opstarten kunnen de klokbron en de prescaling worden aangepast en kan de DFLL worden aangezet om de nauwkeurigheid te verbeteren.

Het instellen van de 32 MHz interne oscillator

De functie `Config32MHzClock` uit code 23.24 selecteert de interne oscillator van 32 MHz als klokbron. De Xmega beschermt met behulp van het *configuration change protection* mechanisme systeemkritische registerinstellingen tegen onbedoelde wijzigingen. Om een systeemkritisch register te wijzigen moet eerst een

handtekening in het CCP-register worden gezet. Daarna zijn er vier klokslagen beschikbaar om de gewenste verandering aan te brengen.

Om de 32 MHz interne oscillator te gebruiken moet deze oscillator worden ingeschakeld en als klokbron worden geselecteerd. In code 23.24 wordt op regel 3 de 32 MHz oscillator aangezet. Na het inschakelen duurt het enige tijd voordat de oscillator stabiel is. Op regel 4 wacht de functie totdat dit het geval is. Regel 6 selecteert de interne oscillator van 32 MHz als systeemklok. Omdat het register CTRL van de klok *configuration change protected* is, is op regel 5 eerst de handtekening in het CCP-register gezet.

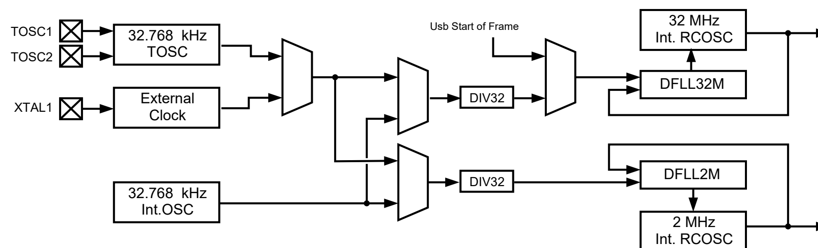
Code 23.24 : De configuratiefunctie voor het instellen van de 32 MHz interne oscillator.

```
1  void Config32MHzClock(void)
2  {
3    OSC.CTRL |= OSC_RC32MEN_bm;                // Enable internal 32 MHz oscillator
4    while(!(OSC.STATUS & OSC_RC32MRDY_bm));    // Wait for oscillator is ready
5    CCP = CCP_IOREG_gc;                        // Security signature to modify clock
6    CLK.CTRL = CLK_SCLKSEL_RC32M_gc;           // Select sysclock 32 MHz oscillator
7  }
```

Bij veel toepassingen moet de klokfrequentie bij de compilatie bekend zijn. Standaard gebruikt de compiler 1 MHz. Met de compiler-optie -DF_CPU kan dit worden gewijzigd. De voorbeelden in het boek gebruiken de standaardklok van 2 MHz en definiëren F_CPU niet met deze compiler-optie maar met de macro F_CPU. Bij 32 MHz luidt de definitie van de klokfrequentie:

```
#define F_CPU   32000000UL
```

Deze macro stelt de klok niet in. Het vertelt de compiler alleen dat de frequentie van de systeemklok 32 MHz is.



Figuur 23.22 : De regellussen met de DFLL voor een nauwkeurig kloksignaal van 2 MHz of 32 MHz. Rechts staan de regellussen met de DFLL's. Het linkerdeel van het schema selecteert de kalibratieklok waarmee de stabiliteit en de nauwkeurigheid van de oscillatoren verbeterd wordt.

Het verbeteren van de precisie van de interne 2 MHz en 32 MHz oscillator

De stabiliteit en nauwkeurigheid van kloksignalen kan aanmerkelijk verbeterd worden met een DFLL, *digital frequency locked loop*. Daarvoor is een regellus en kalibratieklok nodig met een hoge nauwkeurigheid en grote stabiliteit. De frequentie van deze kalibratieklok mag veel lager zijn dan die van het te verbeteren signaal. Figuur 23.22 laat de regellussen en de selectie van de kalibratieklok zien. Deze klok kan de interne 32 kHz oscillator, een externe 32 kHz oscillator of een extern kloksignaal zijn.

In code 23.25 staat de functie `AutoCalibration32M`, die de interne 32 MHz oscillator verbetert met behulp van een DFLL en de interne oscillator van 32 kHz. De functie zet de interne oscillator van 32 kHz aan, selecteert vervolgens de 32 kHz als kalibratieklok voor de 32 MHz oscillator en zet de DFLL van de 32 MHz oscillator aan.

Code 23.25 : De functie `AutoCalibration32M`, die de interne 32 MHz oscillator verbetert.

```
1  void AutoCalibration32M(void)
2  {
3    OSC.CTRL       |= OSC_RC32KEN_bm;             // Enable internal 32 kHz oscillator
4    while(!(OSC.STATUS & OSC_RC32KRDY_bm));       // Wait for oscillator is ready
5    OSC.DFLLCTRL   &= ~(OSC_RC32MCREF_gm);        // Select 32 kHz calibration source
6    OSC.DFLLCTRL   |=   OSC_RC32MCREF_RC32K_gc;   //    for 32 MHz oscillator
7    DFLLRC32M.CTRL |= DFLL_ENABLE_bm;             // Enable DFLL for 32 MHz oscillator
8  }
```

In code 23.26 staat de functie `AutoCalibration2M`, die op een zelfde manier de interne 2 MHz oscillator verbetert.

Code 23.26 : De functie `AutoCalibration2M`, die de interne 2 MHz oscillator verbetert.

```
1  void AutoCalibration2M(void)
2  {
3    OSC.CTRL      |= OSC_RC32KEN_bm;             // Enable internal 32 kHz oscillator
4    while(!(OSC.STATUS & OSC_RC32KRDY_bm));      // Wait for oscillator is ready
5    OSC.DFLLCTRL  &= ~(OSC_RC2MCREF_bm);         // Select 32 kHz calibration source
6    OSC.DFLLCTRL  |=   OSC_RC2MCREF_RC32K_gc;    //    for 2 MHz oscillator
7    DFLLRC2M.CTRL |= DFLL_ENABLE_bm;             // Enable DFLL for 2 MHz oscillator
8  }
```

Het verbeteren van de interne oscillatoren met een extern kristal

In plaats van de interne 32 kHz oscillator kan er ook een extern kristal van 32,768 kHz gebruikt worden. Een extern kristal geeft een nauwkeuriger en stabieler signaal dan de interne oscillator. Dit kristal moet worden aangesloten op de `TOSCn`-ingangen op de manier, die bij de introductie van de Xmega al beschreven is in figuur 14.7.

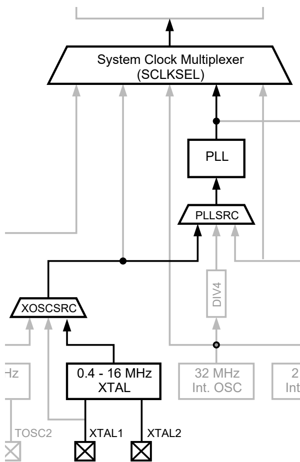
Code 23.27 : De functie `AutoCalibrationTosc32M`, die de interne 32 MHz oscillator verbetert met een extern kristal.

```
1  void AutoCalibrationTosc32M(void)
2  {
3    OSC.XOSCCTRL   |= OSC_XOSCSEL_32KHz_gc;       // Select extern 32 kHz TOSC
4    OSC.CTRL       |= OSC_XOSCEN_bm;              // Enable extern 32 kHz TOSC
5    while(!(OSC.STATUS & OSC_XOSCRDY_bm));        // Wait for oscillator is ready
6    OSC.DFLLCTRL   &= ~(OSC_RC32MCREF_gm);        // Select 32 kHz TOSC calibration source
7    OSC.DFLLCTRL   |=   OSC_RC32MCREF_XOSC32K_gc; //    for 32 MHz oscillator
8    DFLLRC32M.CTRL |= DFLL_ENABLE_bm;             // Enable DFLL for 32 MHz oscillator
9  }
```

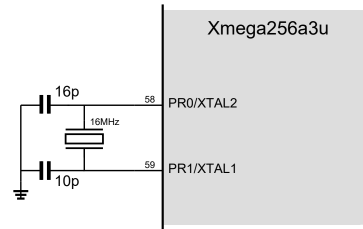
Code 23.27 lijkt heel sterk op code 23.25. Op regel 3 is de selectie van het externe 32 kHz kristal toegevoegd en op regel 4, 5 en 7 zijn de namen aangepast.

Het instellen van de klok op basis van een extern kristal

Het Xmega-bord uit bijlage J heeft een extern kristal van 16 MHz. Met dit kristal kan een zeer nauwkeurige systeemklok worden gemaakt.
In figuur 23.23 staat een deel van de klokdistributie uit figuur 23.21 dat de systeemklok instelt op een frequentie van 32 MHz. Het kristal is aangesloten op de pinnen `XTAL1` en `XTAL2` van de Xmega met twee condensatoren van 16 pF en 10 pF, zie ook figuur 23.24.



Figuur 23.23 : Een deel van het kloksysteem. Met zwart zijn de onderdelen aangegeven, die code 23.28 instelt.



Figuur 23.24 : De aansluiting van het externe kristal op de Xmega256a3u.

De instelling van het kloksysteem bestaat uit drie onderdelen: de oscillator, de PLL en de klok. In figuur 23.23 zijn dat de multiplexers `XOSCSRC`, `PLLSRC` en `SCLKSEL`, die geconfigureerd worden met de controlregisters `OSC.XOSCCTRL`, `OSC.PLLCTRL` en `CLK.CTRL`.

Code 23.28 : De functie `Config32MHzClock_Ext16M`, die de interne 32 MHz oscillator maakt met een 16 MHz extern kristal.

```
1  void Config32MHzClock_Ext16M(void)
2  {
3    OSC.XOSCCTRL = OSC_FRQRANGE_12TO16_gc |                 // Select frequency range
4                   OSC_XOSCSEL_XTAL_16KCLK_gc;              // Select start-up time
5    OSC.CTRL |= OSC_XOSCEN_bm;                              // Enable oscillator
6    while ( ! (OSC.STATUS & OSC_XOSCRDY_bm) );              // Wait for oscillator is ready
7
8    OSC.PLLCTRL = OSC_PLLSRC_XOSC_gc | (OSC_PLLFAC_gm & 2);  // Select PLL source and multipl. factor
9    OSC.CTRL |= OSC_PLLEN_bm;                               // Enable PLL
10   while ( ! (OSC.STATUS & OSC_PLLRDY_bm) );               // Wait for PLL is ready
11
12   CCP = CCP_IOREG_gc;                                     // Security signature to modify clock
13   CLK.CTRL = CLK_SCLKSEL_PLL_gc;                          // Select system clock source
14   OSC.CTRL &= ~OSC_RC2MEN_bm;                             // Turn off 2MHz internal oscillator
15   OSC.CTRL &= ~OSC_RC32MEN_bm;                            // Turn off 32MHz internal oscillator
16 }
```

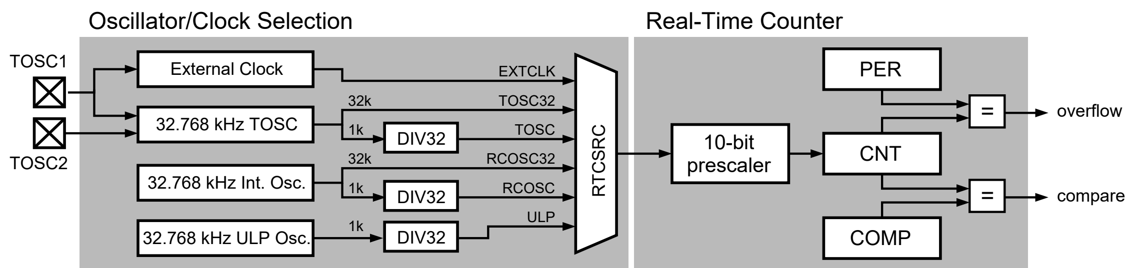
Op meerdere plaatsen in het boek wordt de functie `init_clock()` gebruikt.
In alle gevallen is dit een alias (macro) voor de functie `Config32MHzClock_Ext16M()` uit code 23.28.

Code 23.28 stelt op regel 3 de oscillator in op een extern kristal met een frequentiebereik van 12 tot 16 MHz en een maximale startuptijd van 16000 klokslagen.
Regel 8 selecteert de oscillator als bron voor de PLL en maakt de vermenigvuldigingsfactor van de PLL 2.
De toewijzing op regel 13 selecteert de PLL als bron voor de systeemklok. Omdat het register `CTRL` van de klok *configuration change protected* is, is eerst de handtekening in het `CCP`-register gezet. De regels 14 en 15 zetten de beide interne oscillatoren uit. Dit geeft minder kans op storingen en het beperkt de dissipatie.

23.6 De realtime-counter

De Xmega heeft een 16-bits realtime-counter om nauwkeurige tijd te definiëren. Met de teller kan de microcontroller op gezette tijden een bepaalde functie laten uitvoeren of wakker maken uit een slaapmodus. De realtime-counter heeft een eigen klok. Meestal is de klokbron een extern 32,768 kHz kristal, maar dit mag ook een externe klok, de interne 32,768 kHz oscillator of de ULP zijn. De ULP, *ultra low power*, is een minder precieze interne oscillator van 32 kHz met een zeer lage vermogensdissipatie. De uitgang van de ULP wordt gedeeld door 32. De frequentie van de uitgang is dus 1,024 kHz. De uitgangen van de andere twee oscillatoren kunnen ook door 32 gedeeld worden. De reden om de klok door 32 te delen is dat dit de vermogensdissipatie beperkt en de periodetijd groter is. De bronnen van 32 kHz hebben daarentegen een kleinere tijdsresolutie.

Een aantal componenten uit de Xmega-familie hebben geen 16-bits RTC, maar een 32-bits RTC. De Xmega256a3u heeft een 16-bits RTC.



Figuur 23.25 : De realtime-counter met de klokselectie. Links staan de oscillator- en klokselectie voor de realtime-counter. De multiplexer `RTCSRC` selecteert één van de zes klokbronnen. Bij de kloklijnen staan de namen van de groepsconfiguraties.

In figuur 23.25 zijn de klokselectie en de realtime-counter getekend. De realtime-counter heeft net als de gewone timer/counter een register `CNT` voor het tellen en een register `PER` voor het instellen van de maximale periodetijd van de teller. Als `CNT` gelijk is aan `PER` is er een overflow-interrupt en begint `CNT` weer bij nul te tellen. Het register `COMP` kan worden gebruikt om een alarmtijd in te stellen. Als `CNT` gelijk is aan `COMP` is er een compare-overflow.

Code 23.29 : De functie `Config32kHzRTC` definieert de 32 kHz interne klok als bron.

```
1  void Config32kHzRTC(void)
2  {
3    OSC.CTRL   |= OSC_RC32KEN_bm;             // Enable internal 32 kHz Osc
4    while(!(OSC.STATUS & OSC_RC32KRDY_bm));   // Wait for oscillator is ready
5    CLK.RTCCTRL = CLK_RTCSRC_RCOSC32_gc |     // Select Internal 32 kHz Osc RTC source
6                  CLK_RTCEN_bm;               // Enable RTC clock
7    RTC.CTRL    = RTC_PRESCALER_DIV1_gc;      // prescaling 32kHz/1 = 32768 ticks/s
8    while(RTC.STATUS & RTC_SYNCBUSY_bm);      // Wait for RTC SYNC status not busy
9  }
```

De functie `Config32kHzRTC` uit code 23.29 configureert de interne oscillator 32 kHz als klokbron voor de realtime-counter. De prescaler is ingesteld op 1, zodat er per seconde 32768 klokslagen zijn. In code 23.30 zijn twee klokken gedefinieerd: de systeemklok en de klok voor de RTC. De CPU en de RTC hebben dus een eigen klokdomein. De informatieoverdracht moet gesynchroniseerd worden. Na het

instellen van de prescaling wordt gewacht totdat synchronisatie klaar is. Dit is het geval als het `SYNCBUSY` in het statusregister laag is.

Code 23.30 : Een knipperende led met behulp van de realtime-counter.

```
 1 #include <avr/io.h>
 2 #include <avr/interrupt.h>
 3
 4 void init_rtc(void) {
 5   RTC.PER      = 0x7FFF;              // overflow after 32768 clock cyc.
 6   RTC.INTCTRL |= RTC_OVFINTLVL_LO_gc;  // set interrupt on RTC overflow
 7   RTC.CNT      = 0;                   // clear initial count
 8 }
 9
10 ISR(RTC_OVF_vect)
11 {
12   PORTC.OUTTGL = PIN0_bm;             // switch LED on/off
13 }
14
15 int main(void)
16 {
17   Config32MHzClock();                 // configure system clock
18   Config32kHzRTC();                   // configure RTC clock
19   init_rtc();
20   PORTC.DIRSET = PIN0_bm;
21
22   PMIC.CTRL |= PMIC_LOLVLEN_bm;
23   sei();
24
25   while(1);   // do nothing
26 }
```

Realtime-counter met intern kristal en 32 kHz klokselectie

In code 23.30 staat een programma dat de overflow van de realtime-counter gebruikt om een led te laten knipperen. De led gaat iedere twee seconde een keer aan en een keer uit. De functie `init_rtc` zet de overflowinterrupt aan en geeft register `PER` de waarde 32767. In `main` stelt `Config32MHzClock` de systeemklok in op 32 MHz en stelt `Config32kHzRTC` de klok van de realtime-counter in op 32,768 kHz. Na de initialisatie doet de microcontroller verder niets meer.

De realtime-counter geeft een overflow na `PER` + 1 oftewel 32768 klokslagen. Met een klokfrequentie van 32,768 kHz is er precies iedere seconde een overflow, die de interruptfunctie op regel 10 start en pin `0` van poort C laat omklappen.

Realtime-counter met extern kristal en 1 kHz klokselectie

In code 23.31 staat een functie `Config1kHzToscRTC`, die als oscillator een 32 kHz extern kristal selecteert en de daarvan afgeleide 1,024 kHz kloklijn als bron kiest. De prescaler is ingesteld op 1024, zodat er iedere seconde één klokslag is.
Als in code 23.30 op regel 5 de initialisatiefunctie register `PER` nul maakt, is `PER` + 1 gelijk aan 1 en is er iedere klokslag een overflow. Als bovendien op regel 18 de functie `Config1kHzToscRTC` wordt gebruikt, is er iedere seconde een klokslag en iedere seconde een overflow, waardoor pin `0` van poort C iedere seconde omklapt.

Code 23.31 : De functie `Config1kHzToscRTC` definieert de externe kristaloscillator als bron.

```
1  void Config1kHzToscRTC(void)
2  {
3    OSC.XOSCCTRL |= OSC_XOSCSEL_32KHz_gc;        // Select extern 32 kHz TOSC
4    CCP = CCP_IOREG_gc;                          // Security Signature to modify clock
5    OSC.CTRL     |= OSC_XOSCEN_bm;               // Enable internal 32 kHz Tosc
6    while(!(OSC.STATUS & OSC_XOSCRDY_bm));       // Wait for oscillator ready
7    CLK.RTCCTRL   = CLK_RTCSRC_TOSC_gc |         // Select Internal 1 kHz Osc RTC source
8                    CLK_RTCEN_bm;                // Enable RTC clock
9    RTC.CTRL      = RTC_PRESCALER_DIV1024_gc;    // Select 1kHz/1024 = 1 ticks/s
10   while(RTC.STATUS & RTC_SYNCBUSY_bm);         // Wait for RTC SYNC status not busy
11 }
```

Het verschil met de vorige opzet is dat de klok van de realtime-counter nu 32768 keer zo langzaam is en de realtime-counter veel minder vermogen verbruikt. Met deze klokselectie definieert een grotere waarde van `PER` ook een langere periodetijd.

23.7 Het EEPROM

De meeste microcontrollers hebben drie soorten geheugens: flash voor het programma, RAM voor vluchtige gegevens en EEPROM voor de opslag van niet-vluchtige gegevens. EEPROM staat voor *Electrical Erasable Programmable Read-Only Memory*. Bij een niet-vluchtig geheugen blijven de gegevens bewaard als de spanning wegvalt. Het aantal keer dat er naar een EEPROM geschreven wordt, is beperkt. Het EEPROM van een Xmega is maximaal 4K groot en kan maximaal 80000 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 22,2 uur.

De benadering van het flashgeheugen is eerder besproken in paragraaf 17.6.

De gegevens worden per byte in het EEPROM gezet. De beperking van 80000 keer schrijven, is te verbeteren door het hele EEPROM te gebruiken. Door de gegevens steeds op een ander adres te zetten, kan er tot 100 keer vaker worden geschreven.

Er zijn twee methoden om het EEPROM te benaderen: IO-mapped en memory-mapped. Bij IO-mapped zijn er speciale registers nodig om de data te lezen en te schrijven. Memory-mapped betekent dat de lees- en bufferbewerkingen gemapt zijn in de *data space*. Gegevens kunnen direct uit het geheugen worden gelezen en bij het schrijven worden de gegevens eerst in de bufferruimte gezet en daarna wordt met een IO-mapped schrijfoperatie in het EEPROM geschreven.

Het EEPROM bevat bij de Xmega256a3u 4K bytes en is verdeeld in 128 *pages*. Iedere *page* is 32 bytes groot. Het *page address* bevat de zeven meest significante bits en het *byte address* de vijf minst significante bits. Samen vormen deze bits het 12-bits adres dat voor het 4K geheugen nodig is, zie figuur 23.26.

Figuur 23.26 : De adressering van het 4K EEPROM bij de Xmega256a3u.

Bij application note 1315 van Atmel met de uitleg over het EEPROM hoort een driver. Deze bestaat uit drie bestanden: een c-bestand `eeprom_driver.c`, een headerbestand `eeprom_driver.h` en een algemeen hulpbestand `avr_compiler.h`. De driver kent twee functies `EEPROM_WriteByte` en `EEPROM_ReadByte`, die met behulp van de IO-mapped methode respectievelijk een byte naar het EEPROM schrijft en een byte uit het EEPROM leest.

Code 23.32 : Het schrijven naar en lezen uit het EEPROM met IO-mapped functies.

```
 1 #define F_CPU    2000000UL
 2
 3 #include <avr/io.h>
 4 #include <util/delay.h>
 5 #include "eeprom_driver.h"                // EEPROM Driver from AN 1315
 6
 7 int main(void)
 8 {
 9   PORTA.DIRSET = 0xFF;                    // Port A is output
10
11   for (int i=0; i<32; i++) {
12     EEPROM_WriteByte(0, i, i+1);          // Write i+1 to byteaddress i of page 0
13   }
14
15   while (1) {
16     for (int i=0; i<32; i++) {
17       PORTA.OUT = EEPROM_ReadByte(0, i);  // Read byte from byteaddress i of page 0
18       _delay_ms(500);
19     }
20   }
21 }
```

Voorbeeld met een IO-mapped geheugen

In code 23.32 staat een programma dat bij de initialisatie 32 bytes in het EEPROM zet en daarna voortdurend deze 32 bytes uit het EEPROM leest en deze afbeeldt op poort A.

De functies `EEPROM_WriteByte` en `EEPROM_ReadByte` hebben parameters voor twee adressen: voor het *page address* en voor het *byte address*. Dit zijn de prototypen van deze functies uit het headerbestand `eeprom_driver.h`:

```
void    EEPROM_WriteByte(uint8_t pageAddr, uint8_t byteAddr, uint8_t value);
uint8_t EEPROM_ReadByte (uint8_t pageAddr, uint8_t byteAddr);
```

In het voorbeeld is bij de aanroep van beide functie het *page address* `0` en het *byte address* gelijk aan de variabele `i`. Als er meer dan 32 bytes naar het EEPROM moe-

Code 23.33 : De functie `EEPROM_WriteByte` en `EEPROM_ReadByte` met één parameter voor het adres.

```
23 void eepromWriteByte(uint16_t address, uint8_t value)
24 {
25   EEPROM_FlushBuffer(); // defined in eeprom_driver.h
26   NVM.CMD = NVM_CMD_LOAD_EEPROM_BUFFER_gc;
27
28   NVM.ADDR0 = address & 0xFF;
29   NVM.ADDR1 = (address >> 8) & 0x1F;
30   NVM.ADDR2 = 0x00;
31
32   NVM.DATA0 = value;
33
34   NVM.CMD = NVM_CMD_ERASE_WRITE_EEPROM_PAGE_gc;
35   NVM_EXEC();          // defined in eeprom_driver.h
36 }
```

```
38 uint8_t eepromReadByte(uint16_t address)
39 {
40   EEPROM_WaitForNVM();
41
42   NVM.ADDR0 = address & 0xFF;
43   NVM.ADDR1 = (address >> 8) & 0x1F;
44   NVM.ADDR2 = 0x00;
45
46   NVM.CMD = NVM_CMD_READ_EEPROM_gc;
47   NVM_EXEC();
48
49   return NVM.DATA0;
50 }
```

ten worden geschreven, dan moeten deze verdeeld worden over meerdere pagina's en is er naast `i` een tweede index nodig voor het *page address*.

In code 23.33 staat een alternatieve schrijf- en leesfunctie met slechts één parameter voor het adres. Bij de IO-mapped methode is het gebruik van de pagina's niet noodzakelijk. Het adres is nu een getal tussen `0` en `4095`.
Code 23.32 gebruikt `EEPROM_WriteByte` en `EEPROM_ReadByte` van de driver. Deze functies gebruiken twee indices: het *page address* en het *byte address*. De functies `eepromWriteByte` en `eepromReadByte` uit code 23.34 gebruiken één index *address*. In code 23.34 schrijft de eerste **`for`**-lus in één keer 256 bytes naar het EEPROM en leest de tweede **`for`**-lus steeds 256 bytes in één keer uit het EEPROM.

Code 23.34 : Het schrijven naar en lezen uit het EEPROM met de functies uit code 23.33.

```
11    for (int i=0; i<256; i++) {
12      eepromWriteByte(i, i+1);
13    }
14
15    while (1) {
16      for (int i=0; i<256; i++) {
17        PORTA.OUT = eepromReadByte(i);
18        _delay_ms(500);
19      }
20    }
```

In het EEPROM staan dan op de geheugenadres `0` tot en met 254 de getallen 1 tot en met 255. Op adres 255 staat de waarde `0`. Het EEPROM is immers acht bits breed. De grootste waarde op een geheugenplek is 255.

Memory-mapping

De paginagrootte is wel essentieel bij de memory-mapped methode. Het voordeel van deze methode is dat er direct uit het EEPROM gelezen wordt. Voor het schrijven moet een paginabuffer worden gevuld en daarna naar het EEPROM worden geschreven.

Om de gegevens direct te benaderen kent de driver deze macrodefinitie:

```
#define EEPROM(_pageAddr, _byteAddr) \
     ((uint8_t *) MAPPED_EEPROM_START)[_pageAddr*EEPROM_PAGESIZE + _byteAddr]
```

De gegevensruimte voor de memory-mapped io begint vanaf adres `0x1000`. De macro benadert deze gegevens als een groot array. De index van een specifieke byte uit de array hangt af van het paginanummer `_pageAddr`, de grootte van de pagina en van het byte-adres `_byteAddr`.

Code 23.35 zet op regel 13 de memory-mapping aan. De macro `EEPROM` leest op regel 28 een byte van pagina-adres `0` en byte-adres `i` en op regel 15 de eerste byte van pagina `0`. Regel 19 schrijft `i+1` naar de locatie van pagina `0` en byte `i` in de buffer. Nadat alle 32 bytes geschreven zijn, schrijft regel 21 de inhoud van de buffer naar het EEPROM.

Het **`if`**-statement op regel 15 test of de eerste byte gelijk is aan `0xFF`. De bits van ongeprogrammeerde bytes zijn allemaal hoog. Als dit het geval is, worden de bytes van pagina `0` gevuld met de waarden 1 tot en met 32. Als de eerst byte ongelijk is aan `0xFF` zijn de waarden blijkbaar al eerder in het EEPROM gezet en worden er

Code 23.35 : Het schrijven naar en lezen uit het EEPROM met memory-mapping.

```
1  #define F_CPU    2000000UL
2
3  #include <avr/io.h>
4  #include "eeprom_driver.h"
5  #include <util/delay.h>
6
7  int main(void)
8  {
9    PORTA.DIRSET = 0xFF;
10   PORTC.DIRSET = PIN0_bm;
11
12   EEPROM_WaitForNVM();
13   EEPROM_EnableMapping();           // Enable memory mapping
14
15   if (EEPROM(0, 0) == 0xFF ) {      // If first byte is 'unprogrammed'
16     PORTC.OUTSET = PIN0_bm;
17     EEPROM_WaitForNVM();
18     for (int i=0; i<32; i++) {
19       EEPROM(0, i) = i+1;           //   Write to buffer
20     }
21     EEPROM_AtomicWritePage(0);      //   Write buffer to EEPROM
22   } else {
23      PORTC.OUTCLR = PIN0_bm;
24   }
25
26   while (1) {
27     for (int i=0; i<32; i++) {
28       PORTA.OUT =  EEPROM(0, i);    // Read byte from byteaddress i of page 0
29       _delay_ms(500);
30      }
31   }
32 }
```

geen nieuwe bytes in het EEPROM geschreven. Het EEPROM wordt zo alleen de eerste keer gevuld. Bij een herstart van de microcontroller blijft het EEPROM ongewijzigd. Dit kan de levensduur van het EEPROM aanmerkelijk verlengen.

Code 23.36 : De functie `eepromReadBuffer` leest `n` bytes uit het EEPROM.

```
1  void eepromReadBuffer(uint16_t address, void *pbuf, uint16_t n)
2  {
3    EEPROM_WaitForNVM();
4    EEPROM_EnableMapping();
5    memcpy(pbuf, (void*)(address+MAPPED_EEPROM_START), n);
6    EEPROM_DisableMapping();
7  }
```

Bij het lezen is de buffering niet nodig. De leesbewerkingen zijn bij memory-mapping feitelijk identiek met de gewone RAM-bewerkingen. In code 23.36 staat een functie `eepromReadBuffer`, die `n` bytes vanaf adres `address` uit het EEPROM kopieert naar de geheugenplaats in het RAM waar `pbuf` naar wijst.

Intern gebruikt `eepromReadBuffer` functies van de driver om de memory-mapping aan en uit te zetten. De onderstaande aanroep kopieert 500 bytes van geheugenplaats 256 naar de array `dataArray`:

```
uint8_t dataArray[500];
eepromReadBuffer(256, dataArray, 500);
```

In code 23.35 heeft de macrodefinitie `EEPROM` zowel het paginanummer als het byte-adres nodig om een byte vanaf een specifieke locatie te lezen. De functie `eepromReadBuffer` gebruikt alleen het geheugenadres van het EEPROM. In code 23.35 had dit ook direct gekund met:

```
PORTA.OUT =  *((int8_t *) (MAPPED_EEPROM_START + i));
```

23.8 Flash

Globale variabelen staan in het programmageheugen en worden bij de start van het programma naar het datageheugen gekopieerd. De globale variabelen staan zowel in flash als in SRAM. De ruimte in SRAM is beperkt. Bij de Xmega256a3u is dat 16 KB.

In `main` en in andere functies worden bij de lokale variabelen het attribuut `__progmem__` en de *qualifier* `__flash` genegeerd.

In paragraaf 17.6 is een voorbeeld behandeld met een opzoektabel, die alleen in het programmageheugen staat. Dit voorbeeld gebruikt hiervoor een methode met de *qualifier* `__flash`. Deze methode is beschikbaar vanaf *avr-gcc* versie 4.7. Tot aan deze versie werd alleen een methode met het attribuut `__progmem__` gebruikt.

Code 23.37 : **Een programma met een variable, die alleen in flash en niet in SRAM staat.** Links staat de versie met het attribuut `__progmem__` en rechts de moderne versie met de qualifier `__flash`.

```
1  #include <avr/io.h>
2  #include <avr/pgmspace.h>
3
4  const uint8_t value PROGMEM = 0x55;
5
6  int main(void)
7  {
8    PORTD.DIR = 0xFF;
9    PORTD.OUT = pgm_read_byte(&value);
10
11   while(1) {}
12 }
```

```
1  #include <avr/io.h>
2
3
4  const __flash uint8_t value = 0x55;
5
6  int main(void)
7  {
8    PORTD.DIR  = 0xFF;
9    PORTD.OUT = value;
10
11   while(1) {}
12 }
```

Code 23.37 geeft twee functioneel identieke voorbeelden: één met het attribuut `__progmem__` en één met de *qualifier* `__flash`. De methode met het attribuut gebruikt de macro `PROGMEM` die gedefinieerd is in het bestand `pgm_space.h`:

```
#define PROGMEM   __attribute__((__progmem__))
```

De twee codes wijken op drie plaatsen van elkaar af. Op regel 2 sluit de methode met het attribuut het includebestand `pgm_space.h` in, terwijl bij de methode met de *qualifier* geen speciaal headerbestand nodig is.

Bij de declaratie van de variabele `value` op regel 4 staat het attribuut `PROGMEM` achter de variabelenaam en *qualifier* `__flash` voor het type van de variabele.

De toewijzing op regel 9 gebruikt bij de methode met het attribuut de functie `pgm_read_byte` om een byte uit het programmageheugen te lezen. Bij de methode met de *qualifier* wordt de waarde van `value` direct aan `PORTD.OUT` toegekend.
Het includebestand `pgm_space.h` bevat naast de functie `pgm_read_byte` een groot aantal andere leesfuncties, zoals `pgm_read_word`, `pgm_read_dword` en `pgm_read_float`, die respectievelijk een 16-bits en 32-bits getal of een gebroken getal uit het programmageheugen lezen.

Naast opzoektabellen zijn afbeeldingen voor grafische displays en tekststrings voor niet-grafische displays voorbeelden van grote datablokken. In code 23.38 staat voor beide methoden een functioneel identiek voorbeeld met een string.

Code 23.38 : Een programma dat een string uit het programmageheugen afdrukt. Links staat de versie met het attribuut `__progmem__` en rechts de moderne versie met de qualifier `__flash`.

```
1  #include <avr/interrupt.h>
2  #include <avr/pgmspace.h>
3  #include "stream.h"
4  const char hello[] PROGMEM = "Hello world!\n";
5
6  int main(void) {
7    init_stream(2000000UL);
8    sei();
9    printf_P(hello);
10   while(1) {}
11 }
```

```
1  #include <avr/interrupt.h>
2
3  #include "stream.h"
4  const __flash char hello[] = "Hello world!\n";
5
6  int main(void)
7  {
8    init_stream(2000000UL);
9    sei();
10   printf_P(hello);
11   while(1) {}
12 }
```

De functie `printf_P` maakt onderdeel uit van de `stdio`-bibliotheek en leest automatisch uit het programmageheugen. Deze bibliotheek bevat ook een functie `puts_P` en andere `printf`-functies die gegevens uit het programmageheugen lezen.

Code 23.39 geeft een alternatieve versie voor beide methoden. De methode met het attribuut kan ook zonder expliciete declaratie van de string met de variabele `hello` worden geschreven. Met de macro `PSTR` kan de string direct in de functieaanroep worden geplaatst. De compiler zet deze string in het programmageheugen en `printf_P` leest bij het uitvoeren deze direct uit dit geheugen.

Code 23.39 : Een programma dat een tekst uit het programmageheugen afdrukt. Links staat de versie met het attribuut `__progmem__` en rechts de moderne versie met de qualifier `__flash`.

```
1  #include <avr/interrupt.h>
2  #include <avr/pgmspace.h>
3  #include "stream.h"
4
5  int main(void)
6  {
7    init_stream(2000000UL);
8    sei();
9    printf_P(PSTR("Hello world!\n"));
10   while(1) {}
11 }
```

```
1  #include <avr/interrupt.h>
2
3  #include "stream.h"
4  const __flash char hello[] = "Hello world!\n";
5
6  int main(void)
7  {
8    init_stream(2000000UL);
9    sei();
10   printf("%S", hello);
11   while(1) {}
12 }
```

Bij de methode met `__flash` moeten de strings expliciet worden gedeclareerd. Wel kan in dat geval de normale `printf` gebruikt worden. De *format specifier* `%S` geeft aan dat hello in het programmageheugen staat. Wel geeft de compiler een waarschuwing dat deze het een type `wchar *` verwacht, maar een **`const`** `__flash` **`char`** `*` meekrijgt.

De format specifier `%S` wordt soms ook als `%ls` geschreven.

Code 23.40 geeft voor de methode met `__flash` een voorbeeld met een array van strings. Op regel 4 staat een macro `FSTR`, die een typecasting aan een variabele toevoegt. In de array `towns` op regel 8 worden de strings hiermee getypecast. De strings zijn allemaal arrays van het type char, die nu alleen in het flashgeheugen staan. De tweede **`const`** `__flash` in de declaratie van `towns` zorgt ervoor dat de tabel zelf ook alleen in flash staat.

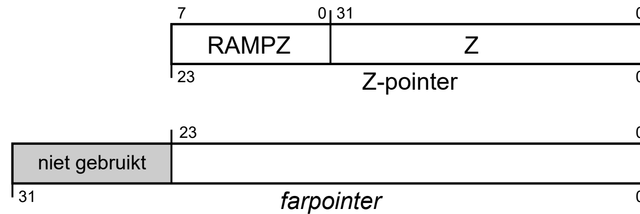
Code 23.40 : Het manipuleren van strings en een array van strings met `__flash`.

```
1  #include <avr/interrupt.h>
2  #include "stream.h"
3
4  #define FSTR(X)    ( ( const __flash char [] ) { X } )
5
6  const __flash char format[] = "%S %S\n";
7  const __flash char *text    = FSTR("town: ");
8  const __flash char * const __flash towns[] = {
9     FSTR("Amsterdam"), FSTR("Breda"), FSTR("Culemborg"), FSTR("Delft") };
10
11 int main(void)
12 {
13   init_stream(2000000UL);
14   sei();
15
16   for (uint8_t i=0; i<4; i++)  printf_P(format, text, towns[i]);
17
18   while(1) {}
19 }
```

Regel 6 definieert een formatstring, die alleen in flash staat. Regel 7 definieert een string, die ook alleen in flash staat. De notatie is anders: `FSTR` geeft expliciet aan dat de string `"town: "` alleen in flash staat en `text` pointer is, die naar deze string wijst.
De functie `printf_P` van regel 16 gebruikt de formatstring `format` die in flash staat, en drukt de tekst `"town: "` met één van de arrayelementen af.

Adressen buiten het 64K bereik

De Xmega is een 8-bits microcontroller met een 16-bits adressering. Het bereik bij een 16-bits adres is 64K. Het flashgeheugen van de Xmega256a3u is ruim 256 KB groot. Er is dus een speciale adressering van een adres groter dan 64K te lezen.
Het programmageheugen is daarom ingedeeld in pagina's van 64 KB. Iedere pagina heeft een nummer dat kan worden toegevoegd aan het adres. Voor het lezen uit het programmageheugen gebruikt de Xmega register `Z` en het paginanummer staat in register `RAMPZ`. Figuur 23.27 laat zien hoe deze registers samen een 24-bits adres vormen.



Figuur 23.27 : De registers `Z` en `RAMPZ` voor de adressering boven de 64K. Samen vormen deze twee registers een 24-bits adres. Speciale functies uit `pgmspace.h` gebruiken een `uint32_t` voor het adres. De acht meest significante bits worden niet gebruikt.

De `pgm_space`-bibliotheek definieert alle macrofuncties voor het lezen op twee manieren: met de beperking dat het adres lager is dan 64K en zonder deze beperking. Zo zijn er twee functies om een byte uit het programmageheugen te lezen, namelijk `pgm_read_byte_near` en `pgm_read_byte_far`. De functie `pgm_read_byte` uit code 23.37 is identiek met `pgm_read_byte_near` en gebruikt een `uint16_t` voor het adres. Deze functie gebruikt alleen register `Z`.
De `pgm_read_byte_far` gebruikt een `uint32_t` voor het adres dat ook aangeduid wordt als een *far pointer*. Deze functie vult register `RAMPZ` met de adresbits 23 : 16 en register `Z` met de adresbits 16 : 0.

De nieuwe methode met de *qualifier* `__flash` kent standaard nog geen simpele methode of functies om voorbij de 64K te lezen. De pagina's boven de 64K kunnen worden benaderd met `__flashN` waarbij `N` het nummer van de pagina is of met behulp van de `__memx` dat een 24-bits adres op de juiste manier op de adressering van het data- of programmageheugen afbeeldt.

Meer informatie over `__flash` en de adressering staat op de site van de GNU-compiler: https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html

In het algemeen lijkt bij het gebruik van het programmageheugen de nieuwe methode met de *qualifier* `__flash` een goede keuze. Bij adressen groter dan 64K lijkt de oude methode voorlopig nog eenvoudiger.

23.9 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.

De meeste microcontrollers en microprocessoren kennen een of meer slaapstanden. Bij een slaapstand worden bepaalde delen van de chip uitgezet. Dit wordt gedaan door de klok van het betreffende deel uit te zetten. Figuur 23.21 liet al eerder zien dat uit de systeemklok verschillende andere kloksignalen zijn afgeleid. De Xmega kent vijf slaapstanden: *idle*, *power-down*, *power-save*, *standby* en *extended standby*. Tabel 23.7 toont deze slaapstanden, de klokken die daarbij actief zijn en de interrupts die de microcontroller uit de slaapstand haalt.

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. *Power-down* is de diepste slaapstand. Alle klokken staan uit en de microcontroller is alleen met een asynchroon signaal de Xmega uit deze slaapstand halen.

Tabel 23.7 : De slaapstanden van de Xmega. De linker kolom geeft de modus. De volgende vijf kolommen geven de klokken en oscillatoren die uit of aan staan. In de laatste vijf kolommen staan de interrupts die de microcontroller weer wakker maken.

Modus	CPU-/NVM-klok	Perifere-klok	RTC-klok	Systeem-oscillator	RTC-oscillator	USB resume	Asyn. port interrupt	TWI match interrupt	RTC interrupt	Overige interrupts
Idle	uit	aan	aan	aan	aan	X	X	X	X	X
Power-down	uit	uit	uit	uit	uit	X	X	X		
Power-save	uit	uit	aan	uit	aan	X	X	X	X	
Standby	uit	uit	uit	aan	uit	X	X	X		
Extended standby	uit	uit	aan	aan	aan	X	X	X	X	

Voorbeeld met de lichtste slaapstand: *idle*

De *avr-libc*-bibliotheek bevat een headerbestand `sleep.h` met een aantal voorgedefinieerde functies om de microcontroller in een slaapstand te zetten. In code 23.41 brengt op regel 20 de functie `sleep_mode` de microcontroller in de slaapstand. Het hoofdprogramma stopt direct. Nadat de microcontroller uit de slaapstand is gehaald, wordt de rest van de code uitgevoerd. De led op uitgang 0 van poort C knippert vijf keer en de microcontroller komt daarna op nieuw in de slaapstand.

Code 23.41 : De lichtste slaapstand idle timer/counter TCC0 als wekker.

```
1  #define F_CPU  2000000UL
2
3  #include <avr/io.h>
4  #include <avr/sleep.h>
5  #include <avr/interrupt.h>
6  #include <util/delay.h>
7
8  void init_sleep_alarm(void);
9
10 int main(void)
11 {
12   PORTC.DIRSET = PIN0_bm;                   // output for led
13   init_sleep_alarm();                       // initialize sleep mode
14                                             //    and alarm function
15   PMIC.CTRL |= PMIC_LOLVLEN_bm;
16   sei();
17
18   while(1)
19   {
20     sleep_mode();                           // go to sleep
21
22     for(int i=0; i<10; i++) {               // blink led five times
23       PORTC.OUTTGL = PIN0_bm;
24       _delay_ms(500);
25     }
26   }
27 }
```

Op regel 13 initialiseert `init_sleep_alarm` de slaapmodus en timer/counter 0 van poort C, die als wekker functioneert. De functie `init_sleep_alarm` staat in code 23.42 en stelt op regel 31 met de functie `set_sleep_mode` de slaapstand in op *idle*.

Daarna wordt timer/counter `TCC0` ingesteld op de normale modus met een overflowinterrupt na iedere 8 seconde.

Code 23.42 : De slaapstand *idle* met timer/counter `TCC0` als wekker.

```
29  void init_sleep_alarm(void)
30  {
31    set_sleep_mode(SLEEP_MODE_IDLE);        // set sleep mode
32
33    TCC0.CTRLB    = TC_WGMODE_NORMAL_gc;
34    TCC0.CTRLA    = TC_CLKSEL_DIV1024_gc;
35    TCC0.PER      = 15624;                  // t = N(PER+1)/f = 8s
36    TCC0.INTCTRLA = TC_OVFINTLVL_LO_gc;
37  }
38
39  ISR(TCC0_OVF_vect)
40  {
41    // do nothing
42  }
```

De overflowinterrupt wekt de microcontroller uit de slaapstand en voert eerst de interruptfunctie uit. Deze functie doet niets. Als deze `ISR` niet bestaat, vindt de microcontroller geen `ISR` en reset zichzelf automatisch. Een lege `ISR` voorkomt dat. Nadat de interruptfunctie is uitgevoerd, gaat de microcontroller verder met het hoofdprogramma. De led knippert vijf keer en gaat weer in de slaapstand.

Voorbeeld met de diepste slaapstand: *power down*

In de diepste slaapstand, de *power down*-modus, zijn alle klokken en oscillatoren uit. De Xmega kan dan alleen wakker worden door een asynchrone externe interrupt. Dat kan alleen door een *resume*-interrupt van de USB, een *address match*-interrupt van een TWI en door een asynchrone externe interrupt. Bij iedere poort heeft pin 2 een asynchrone interrupt.

Code 23.43 : De diepste slaapstand power down met een asynchrone interrupt0 als wekker.

```
29  void init_sleep_alarm(void)
30  {
31    set_sleep_mode(SLEEP_MODE_PWR_DOWN);     // set mode
32
33    PORTB.INT0MASK = PIN2_bm;                // define pin B2
34    PORTB.PIN2CTRL = PORT_OPC_PULLUP_gc |    // with a pull-up
35                     PORT_ISC_FALLING_gc;    // as an asynchronous
36    PORTB.INTCTRL  = PORT_INT0LVL_LO_gc;     // external interrupt
37  }
38
39  ISR(PORTB_INT0_vect)
40  {
41    // do nothing
42  }
```

In code 23.43 staat een functie `init_sleep_alarm`, die de slaapmodus instelt op *power down* en pin 2 van poort B instelt op een externe interrupt. De bijbehorende interruptfunctie doet niets. Deze is weer toegevoegd om te voorkomen dat de

Xmega automatisch reset. Het programma uit code 23.41 valt met de functies uit code 23.43 ook weer in slaap, maar de led knippert nu vijf keer als het signaal op ingang PB2 laag wordt. Dat signaal kan bijvoorbeeld een inbraaksensor zijn.

Het gedissipeerde vermogen

Het gedissipeerde vermogen van een digitaal CMOS-IC hangt af van de totale capaciteit, de voedingsspanning V 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 dynamisch gedissipeerde vermogen P is theoretisch:

$$P = fCV^2 \tag{23.6}$$

Opmerkelijk is dat dit niet afhangt van de weerstanden in het IC. De weerstanden bepalen samen met de capaciteiten wel de schakelsnelheid van het IC. Het dynamisch vermogen kan gereduceerd worden door de voedingsspanning en de frequentie laag te houden.

In de datasheet staat een overzicht met het stroomverbruik voor de slaapstanden bij verschillende voedingsspanningen en klokfrequenties. Een voedingsspanning van 3,0V en een frequentie van 32 MHz geven voor de actieve modus 9,5 mA, voor *idle* 3,8 mA, voor *power save* $< 1{,}5\ \mu$A en voor *power down* 0,1 μA.

De zwevende ingangen en het stroomverbruik

Standaard zijn aansluitingen die niet gebruikt worden ingangen zonder pullup of pulldown. Het zijn zwevende ingangen; ze kunnen laag of hoog zijn. Het stroomverbruik is aanmerkelijk groter met zwevende ingangen.

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.

23.10 De mogelijkheden om de Xmega256a3u te herstarten

De Xmega kent zes 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_{pot} ligt, blijft de interne reset actief. Als de voedingsspanning boven V_{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 1,5μs 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 een bepaalde, minimale spanning krijgt de microcontroller eveneens een reset.
- *PDI-reset*
  Deze reset wordt gebruikt bij de PDI-interface.
- *Software reset*
  Deze reset treedt op als de `SWRST`-bit in het `CTRL`-register van `RST` hoog is. Deze bit is *configuration change protected*.

Het statusregister van `RST` bevat zes bits die aangegeven wat de oorzaak van de reset was. Na het herstarten kan van dit gegeven gebruik worden gemaakt voor een specifiek vervolg.

23.11 Watchdog

Een watchdog is een timer die op geregelde tijden nul gemaakt moet worden. Als dit niet gedaan wordt, treedt er na verloop van tijd een interrupt op die de microcontroller herstart.
De watchdog controleert of het microcontrollerprogramma correct functioneert. Tijdens de normale loop van het programma maakt het de watchdogtimer nul. In geval dat het programma vastloopt, blijft de timer doorlopen en treedt er uiteindelijk een interrupt op waardoor de microcontroller opnieuw start en weer normaal kan 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 systeem na het herstarten in een veilige toestand te brengen door alle eventueel gevaarlijke subsystemen — zoals motoren en hoogspanningscircuits — uit te zetten.

De Xmega heeft een controlregister `CTRL` met vier `PER`-bits die de time-outperiode van de watchdog instellen, een `ENABLE`-bit en een `CEN`-bit. De `ENABLE`-bit zet de watchdog aan; dat mag alleen als `CEN` hoog is. Dit laatste bit is *configuration change protected*. De Xmega heeft ook een `WINCTRL`-register voor de zogenoemde windowmodus, die hier buiten beschouwing blijft. De klok van de watchdog is een 1 kHz klok, die is afgeleid van de 32 kHz *ultra low power*-oscillator, zoals figuur 23.21 laat zien. Het statusregister bevat alleen een `SYNCBUSY`-bit. Deze bit wordt hoog als `CTRL` of `WINCTRL` wijzigt en laag als de watchdog synchroon is met de rest van de microcontroller. Tabel 23.8 geeft de groepsconfiguratie van de `PER`-bits.

Bij het maken van code voor de watchdog zijn er drie mogelijkheden: zelf handmatig de registers de gewenste instelling geven, het headerbestand `wdt.h` uit de *avr-libc*-bibliotheek gebruiken of de driver, die bij application note 1310 hoort, gebruiken. Deze paragraaf gebruikt de functies uit `wdt.h` om de watchdog in de normale modus in te stellen en daarmee toe te passen. Dit headerbestand ondersteunt het gebruik van de window-modus niet. In dat geval is de driver een beter alternatief.

Tabel 23.8 : De PER-bits met de time-out-tijd van de watchdog.

PER[3:0]	Groepsconfiguratie	time-out bij 3,3V
0000	WDT_PER_8CLK_gc	8 ms
0001	WDT_PER_16CLK_gc	16 ms
0010	WDT_PER_32CLK_gc	32 ms
0011	WDT_PER_64CLK_gc	64 ms
0100	WDT_PER_128CLK_gc	128 ms
0101	WDT_PER_256CLK_gc	256 ms
0110	WDT_PER_512CLK_gc	512 ms
0111	WDT_PER_1KCLK_gc	1 s
1000	WDT_PER_2KCLK_gc	2 s
1001	WDT_PER_4KCLK_gc	4 s
1010	WDT_PER_8KCLK_gc	8 s

Het bestand `wdt.h` definieert drie macrodefinities voor de watchdog: `wdt_enable`, `wdt_disable` en `wdt_reset`. Deze functies zetten respectievelijk de watchdog aan, de watchdog uit en de timer op nul.

Code 23.44 : Een basaal voorbeeld met de watchdog.

```
1  #define F_CPU 2000000UL
2
3  #include <avr/io.h>
4  #include <avr/wdt.h>
5  #include <util/delay.h>
6
7  int main(void)
8  {
9    PORTC.DIRSET  = PIN0_bm;
10   PORTC.OUTSET  = PIN0_bm;
11   _delay_ms(10);
12
13   wdt_enable(WDT_PER_256CLK_gc);    // enable watchdog
14
15   while(1)
16   {
17     wdt_reset();                    // reset timer watchdog
18     PORTC.OUTTGL = PIN0_bm;         // normal operation
19     _delay_ms(200);
20   }
21 }
```

In code 23.44 staat het basisconcept voor het gebruik van de watchdogtimer. Op regel 13 wordt de watchdog aangezet en ingesteld op een wachttijd van 256 ms. In de oneindige wachtlus wordt op regel 17 de watchdog steeds op nul gezet. In dit voorbeeld wordt bij normaal functioneren de wachtlus binnen 256 ms doorlopen. De led die met uitgang `0` van poort C verbonden is, zal voortdurend knipperen.

Wanneer 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; er zal na ruwweg 250 ms een watchdog-interrupt zijn die de microcontroller herstart. Als het programma vastloopt bij de delay-functie, zal het na de herstart waarschijnlijk weer vastlopen.

Code 23.45 : Een praktisch voorbeeld met de watchdog.

```
 1 #define F_CPU 2000000UL
 2
 3 #include <avr/io.h>
 4 #include <avr/wdt.h>
 5 #include <util/delay.h>
 6
 7 int main(void)
 8 {
 9   uint8_t i=0;
10   PORTF.DIRSET = PIN1_bm | PIN0_bm;
11
12   if ( RST.STATUS & RST_WDRF_bm ) {
13     RST.STATUS &= RST_WDRF_bm;       // clear watchdog reset flag
14     while (1) {                      // safety loop
15       PORTF.OUTTGL = PIN1_bm;        // indicate wrong operation
16       _delay_ms(125);
17     }
18   }
19   wdt_enable(WDT_PER_1KCLK_gc);      // enable watchdog for 1 s
20
21   while(1)
22   {
23     wdt_reset();                     // reset timer watchdog
24     PORTF.OUTTGL = PIN0_bm;          // indicate normal operation
25     if (i < 10) {
26       _delay_ms(500);                // working properly
27     } else {
28       i = 0;
29       _delay_ms(2000);               // waiting too long
30     }
31     i++;
32   }
33 }
```

In code 23.45 staat een programma waarbij de microcontroller na het herstarten bij een watchdog-interrupt niet meer de normale functie uitvoert, maar een speciale veiligheidslus doorloopt. Na de herstart is de `WDRF`-bit van het statusregister hoog. Regel 12 test of deze bit hoog is. Na een power-on-reset is dit laag en gaat het programma verder bij regel 19 en voert de normale acties uit. De led van `PF0` knippert met een frequentie van 1 Hz. Als `i` gelijk is aan 10, wordt er een taak uitgevoerd, die te lang duurt en volgt er een herstart door de watchdog. Na de herstart is de `WDRF`-bit hoog en voert het programma de *safety loop* op regel 14 uit. In dit voorbeeld knippert de led van `PF1` dan met een frequentie van 4 Hz.

Code 23.45 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.

23.12 Het atomic block

De Xmega is een 8-bits microcontroller. Alle bewerkingen worden in eenheden van 8-bits uitgevoerd. Voor een 8-bits toekenning is één klokslag nodig. Deze toekenning is niet onderbreekbaar. Het is een *atomic* of atomische bewerking. Voor een 16-bits toewijzing zijn meer klokslagen nodig. Daarom kan een interrupt een 16-bits toekenning onderbreken. Dit noemt men een *non atomic* of niet-atomische bewerking.

Bij een niet-atomische toekenning kan er een probleem ontstaan als bij de onderbreking de waarde van het rechterlid verandert. In code 23.46 staat een voorbeeld. Regel 38 kent aan de lokale 16-bits variabele `local` de 16-bits globale variabele `res` toe. De interruptfunctie van de timer verandert iedere 128 ms de waarde van `res` van `0x00FF` naar `0xFF00` of andersom.

Dit voorbeeld is ontleend aan de blog: http://jhaskellsblog.blogspot.nl/2011/11/demonstration-atomic-access-and.html

Code 23.46 : Een demonstratievoorbeeld dat niet-atomische fouten zichtbaar maakt.

```
 7  // regel 1 t/m 7 identiek aan regel 1 t/m 7 van code 19.19
 8
 9  #include <util/atomic.h>
10
11  void init_timer(void)
12  {
13    TCE0.CTRLA     = TC_CLKSEL_DIV64_gc;
14    TCE0.CTRLB     = TC_WGMODE_NORMAL_gc;
15    TCE0.INTCTRLA = TC_OVFINTLVL_LO_gc;
16    TCE0.PER       = 3999;   // Tper = 128 ms @ 2 MHz
17  }
18
19  volatile uint16_t res = 0xFF00;
20
21  ISR(TCE0_OVF_vect)
22  {
23    if (res==0xFF00) res = 0x00FF;
24    else             res = 0xFF00;
25  }
26
27  int main(void)
28  {
29    uint16_t local;
30
31    init_timer();
32    init_stream(F_CPU);
33
34    PMIC.CTRL |= PMIC_LOLVLEN_bm;
35    sei();
36
37    while (1) {
38      local = res;
39
40      if (! (local == 0xFF00 || local == 0x00FF) ) {
41        printf("0x%04x\n", local);
42      }
43    }
44  }
```

De toekenning aan `local` kan door de interrupt worden onderbroken. De waarde van `res` verandert dan tussentijds. Stel dat voor de interrupt `res` `0xFF00` is en dat de lage byte `0x00` al is toegekend aan `local`. Na de interrupt is `res` gewijzigd in `0x00FF`. De hoge byte van `local` wordt dan eveneens `0x00`, zodat `local` de waarde `0x0000` krijgt. Op dezelfde manier kan, als `res` aanvankelijk `0x00FF` was, `local` ook `0xFFFF` worden.

Bij de test op regel 40 kan `local` dus ook `0x0000` of `0xFFFF` zijn. In dat geval is de test waar en drukt het programma `0x0000` of `0xFFFF` af. Dit gebeurt in dit voorbeeld een paar keer per seconde.

Het voorkómen van niet atomische problemen

Een remedie voor dit probleem is om bij de toekenning aan `local` het interruptmechanisme met `cli` tijdelijk uit te zetten:

```
cli();          // clear enable global interrupt
local = res;
sei();          // set enable global interrupt
```

Een andere, meer geschikte oplossing is om regel 38 uit code 23.46 te vervangen door een *atomic block* of atomisch blok.

Code 23.47 : Een demonstratievoorbeeld met een *atomic block*.

```
38    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
39    {
40      local = res;
41    }
```

In code 23.47 staat binnen de accolades van de macro `ATOMIC_BLOCK` de globale interrupt uit en wordt de toekenning altijd zonder onderbreking uitgevoerd. De parameter `ATOMIC_RESTORESTATE` herstelt de interruptstatus, die het programma had voor de start van het atomisch blok. Dit betekent dat als het interruptmechanisme uitgeschakeld was, het na het atomisch blok ook uit staat en als het ingeschakeld was, het na het atomisch blok weer ingeschakeld is.

Met het atomisch blok maakt code 23.47 de variabele `local` altijd `0x00FF` of `0xFF00` en verstuurt het programma nooit iets via de UART.

De macro's `ATOMIC_BLOCK` en `ATOMIC_RESTORESTATE` zijn gedefinieerd in `atomic.h`. Dit bestand bevat ook een macro `ATOMIC_FORCEON`, die na het atomisch blok het interruptmechanisme altijd actief maakt:

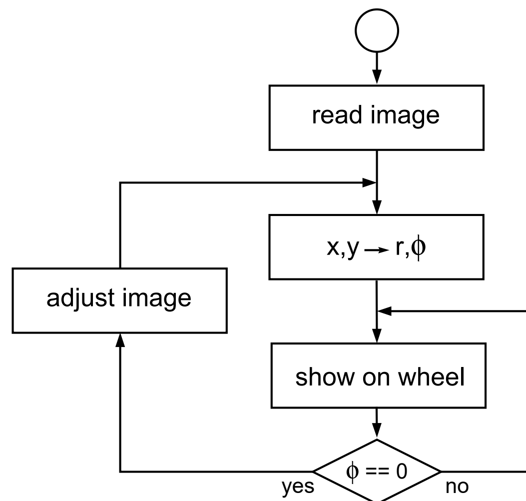
```
ATOMIC_BLOCK(ATOMIC_FORCEON)
{
  local = res;
}
```

Daarnaast is er nog een macrodefinitie `NONATOMIC_BLOCK` waarmee binnen een atomisch blok een niet-atomisch blok kan worden gedefinieerd. Dit blok kent weer twee opties `NONATOMIC_RESTORESTATE` en `NONATOMIC_FORCEOFF`. In het algemeen zullen er weinig toepassingen zijn waarbij `NONATOMIC_BLOCK` nodig is.

A

Stroomdiagrammen

Een stroomdiagram of *flow chart* is een grafische weergave van een programma, deel van een programma, functie, proces of algoritme. Het bestaat uit een aantal geometrische vormen die verbonden zijn met pijlen. De richting van de pijl bepaalt de volgorde waarin het diagram wordt doorlopen. De vormen representeren de acties of deelprocessen die informatie verwerken.



Figuur A.1 : Een voorbeeld van een stroomdiagram.

In figuur A.1 staat een voorbeeld van een stroomdiagram. Het diagram begint bij de cirkel en voert daarna drie acties uit. Daarna wordt als ϕ nul is de actie *adjust image* uitgevoerd, anders wordt er opnieuw naar de actie *show on wheel* gegaan. Naast vormen en pijlen zijn teksten ook een wezenlijk onderdeel van een stroomdiagram. Bij de symbolen staan de acties en de voorwaarden vermeld.
Het stroomdiagram hoort typisch bij een microcontrollerprogramma. Er is geen eindsymbool. Het programma voert voortdurend de ene lus of de andere lus uit.

Een stroomdiagram of *flow chart* legt een algoritme of een proces vastleggen. Alle stappen die voor het algoritme nodig zijn, worden stap voor stap doorlopen. Een stroomdiagram wordt ook gebruikt om een bepaald proces of programma beter te begrijpen. De acties en de volgorde waarin deze acties worden uitgevoerd, worden met een stroomdiagram verduidelijkt.

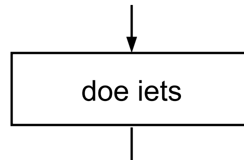
Er bestaat geen eenduidige definitie voor stroomdiagrammen of flowcharts. In 1964 is door de ECMA, European Computer Manufacturers Association, een document ECMA-4 met als titel "Standard for Flow Charts" uitgegeven. In 1966 is de tweede druk van deze standaard uitgebracht. Deze standaard onderscheidt twee soorten flowcharts: de *program flow chart* en de *data flow chart*. Voor beide soorten geeft de ECMA-4 verschillende symbolen

Naast de flowchart bestaan er veel andere diagrammen. Een veelgebruikte vorm is het DFD, *data flow diagram*. Dit diagram lijkt op een flowchart, maar overeenkomstige symbolen hebben een heel andere betekenis. DFD's worden vooral gebruikt bij objectgeoriënteerd talen.

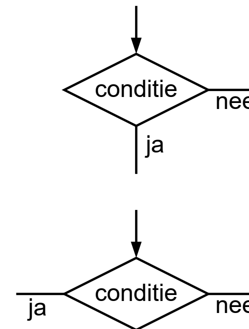
De norm ISO5807:1985 bevat documentatie over symbolen en afspraken voor onder andere flowcharts.

A.1 Symbolen voor een stroomdiagram

Deze paragraaf geeft voor de stroomdiagrammen de meest voorkomende symbolen met de meest voor de hand liggende betekenis.



Figuur A.2 : Het symbool voor een proces, functie of actie.

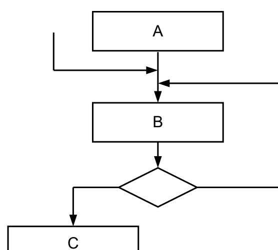


Figuur A.3 : Het symbool voor een keuze of beslissing.

In figuur A.2 staat het symbool dat een proces, functie of actie vertegenwoordigt. De tekst in de rechthoek vertelt de actie en mag ook een eenvoudige, enkele toewijzing zijn.

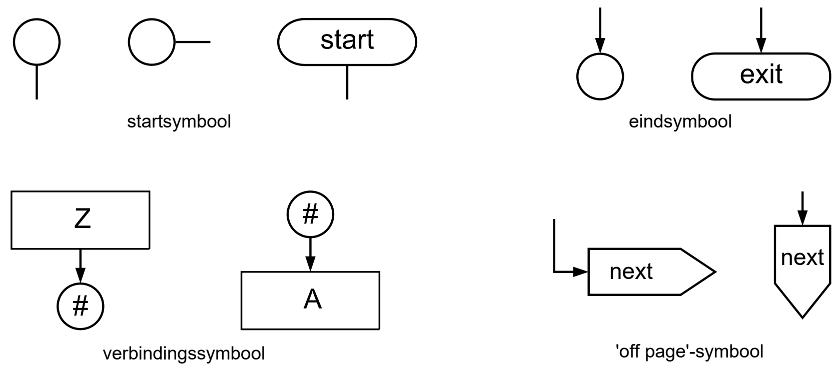
De ruit of het wybertje geeft een beslissing of keuze weer. De oriëntatie is meestal zo gekozen dat de datastroom aan de bovenkant binnenkomt. Aan de zijkanten of aan de onderkant wordt de ruit weer verlaten. Figuur A.3 laat zien dat er dus meerdere configuraties mogelijk zijn.

De pijl geeft de richting aan waarin het diagram moet worden doorlopen. Dat kan de richting zijn van de gegevensstroom of de volgorde waarin de acties uitgevoerd moeten worden.
Een verbinding die een symbool ingaat, krijgt bij het symbool een pijlpunt. Als er meer pijlen het symbool ingaan, sluiten de andere pijlen op de pijl aan die het symbool ingaat. Figuur A.4 geeft een voorbeeld.
Een verbinding kan alleen worden gesplitst met een beslissingssymbool. In figuur A.4 komen bij symbool B drie trajecten bij elkaar. Na symbool B is een beslissingssymbool nodig om het traject te splitsen in een traject naar symbool C en een traject terug naar symbool B.



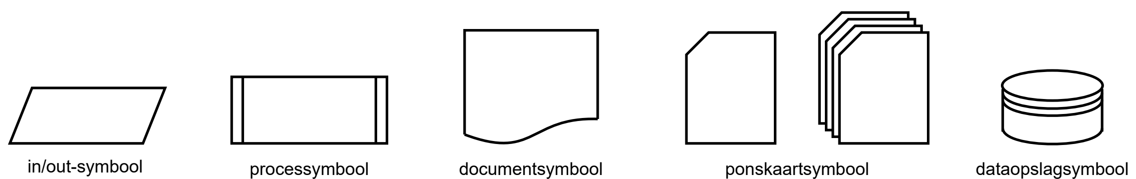
Figuur A.4 : Pijlen geven de richting aan.

Hoewel er in de standaard ECMA-4 aanwijzingen staan voor kruisende en snijdende verbindingslijnen, is het raadzaam de verbindingen in een flowchart nooit te laten kruisen. In de praktijk is het altijd mogelijk het diagram zo op te zetten dat de verbindingen niet kruisen. Eventueel kunnen de verbindingssymbolen uit figuur A.5 gebruikt worden. Vaak is het dan beter om het diagram te vereenvoudigen of te splitsen.

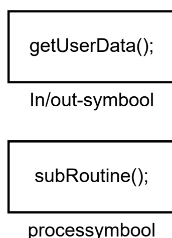


Figuur A.5 : De connectiesymbolen voor het stroomdiagram.

De pijl, de ruit en de rechthoek zijn drie belangrijke symbolen voor het samenstellen van een stroomdiagram. Andere symbolen zijn bijvoorbeeld de aansluitsymbolen uit figuur A.5. Dit boek gebruikt een cirkel om het begin en het einde van het stroomdiagram te markeren. Met tekst is dit symbool pilvormig.
De cirkel wordt ook gebruikt om een verbinding te maken in plaats van een pijl. De cirkels met hetzelfde nummer zijn dan met elkaar verbonden.
Het 'off page'-symbool geeft aan dat de rest van het diagram op een volgende pagina staat.



Figuur A.6 : Diverse symbolen voor stroomdiagrammen.



Figuur A.7 : Alternatieven voor het in/out-symbool en het processymbool.

In figuur A.6 staan een aantal van de overige symbolen, die gebruikt worden bij flowcharts. Het in/out-symbool is een parallellogram. Het krijgt gegevens van of toont informatie aan de gebruiker. Het processymbool representeert een subroutine.

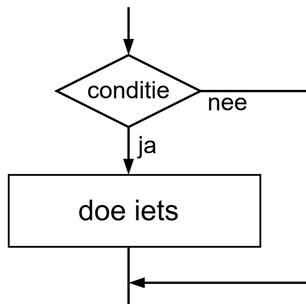
De ECMA-4 standaard en flowcharttekenprogramma's kennen een groot aantal andere symbolen. Met name voor allerlei vormen van gegevensopslag. Figuur A.6 geeft het symbool voor een document, een ponskaart en magnetische dataopslag. Het symbool voor magnetische dataopslag is in figuur 2.4 gebruikt als symbool voor bibliotheken die op een vaste plaats op de harde schijf zijn opgeslagen. In dezelfde figuur is het ponskaartsymbool gebruikt om bestanden weer te geven.

A.2 Programmacode representeren met flowcharts

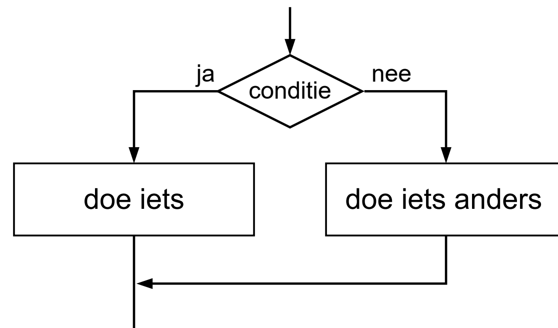
C is een procedurele taal en kan daarom volledig met stroomdiagrammen worden beschreven. Er zijn ontwikkeltools — bijvoorbeeld *Flowcode* — waarmee de

gebruiker stroomdiagrammen tekent, die automatisch omgezet worden naar C. Zolang de diagrammen niet te groot worden en het programma niet te complex is, kan dit overzichtelijk zijn. Bij complexe functies wordt het snel onoverzichtelijk om de hele code op detailniveau met flowcharts te representeren. Niettemin kan iedere C-constructie in een stroomdiagram worden vastgelegd.

Met het beslissingssymbool kunnen de `if`-statements gerepresenteerd worden. In figuur A.8 staat het equivalente stroomdiagram van een `if`-statement en in figuur A.9 die van de `if-else`.

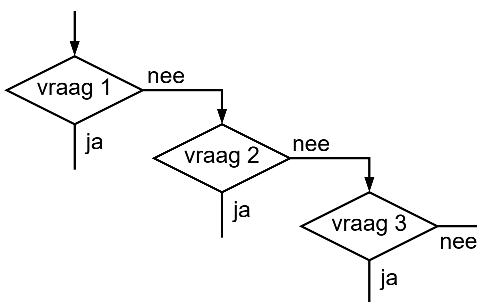


Figuur A.8 : Het equivalent van een `if`-statement.

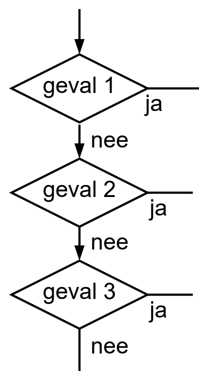


Figuur A.9 : Het equivalent van een `if-else`-statement.

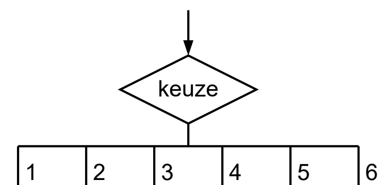
In figuur A.10 staat het equivalente stroomdiagram van een `if-else-if` en in figuur A.11 dat van een `switch`-statement. Er is weinig verschil tussen deze twee diagrammen, alleen de plaatsing van de symbolen is verschillend.
Zeker bij een `switch`-statement met veel keuzes kan het diagram onoverzichtelijk worden. Figuur A.12 geeft een alternatief.



Figuur A.10 : Het equivalent van een `if-lese-if`-statement.

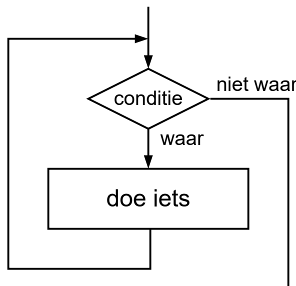


Figuur A.11 : Het equivalent van een `switch`-statement.

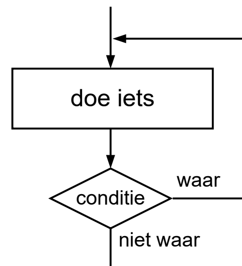


Figuur A.12 : Een alternatief voor het `switch`-statement.

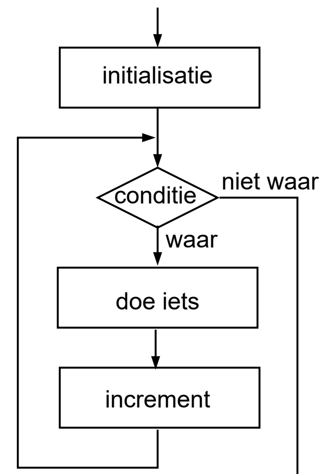
De herhalingsopdrachten kunnen eveneens met een stroomdiagram worden weergegeven. In figuur A.13 staat de flowchart van de `while` en in figuur A.14 staat die van de `do-while`. Bij `while` wordt altijd eerst de test uitgevoerd en bij de `do-while` worden de statements minimaal één keer uitgevoerd.



Figuur A.13 : Het stroomdiagram van een `while`-statement.



Figuur A.14 : Het stroomdiagram van een `do while`-statement.

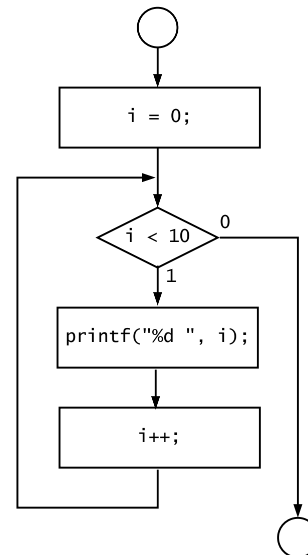


Figuur A.15 : Het stroomdiagram van een `for`-statement.

De stroomdiagram voor het `for`-statement staat in figuur A.15. Dit diagram is bijna identiek met die van de `while`. Voor het beslissingssymbool staat nu een processymbool met de initialisatie en in de lus staat als laatste statement de incrementie.

Code A.1 : Een programma met een `for`-lus.

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



Figuur A.16 : Het stroomdiagram voor het programma met het `for`-statement.

In de diagrammen van figuur A.13 en figuur A.14 zijn de `while` en de `do-while` direct te herkennen. Bij de flowchart uit figuur A.15 is het `for`-statement minder duidelijk. Het voorbeeld uit figuur A.16, dat bij code A.1 hoort, maakt het concreter.

De regels 7, 8 en 9 van code A.1 komen overeen met het diagram van figuur A.16. Voor iemand die C kent zijn deze drie regels veel eenvoudiger te lezen, dan de relatief omvangrijke figuur. Dit demonstreert dat het ontwerpen vanuit gedetail-

leerde stroomdiagrammen weinig zinvol is. Een stroomdiagram is juist een krachtig hulpmiddel om de globale *flow* van een programma vast te leggen.

B

RS232

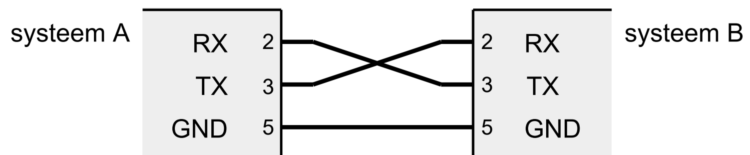
De RS232 (*Recommended Standard 232*) is een standaard voor asynchrone seriële communicatie. Het is een van de oudste communicatie protocollen voor het versturen van gegevens.

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 B.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 B.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

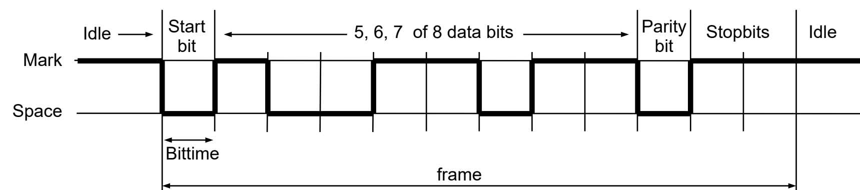
De verzender verstuurt de volgende informatie:

- 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 de stopbit mag eventueel een pariteitsbit worden meegezonden.

In figuur B.2 staat een voorbeeld met acht databits, een pariteitsbit en twee stopbits. 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 B.1 : RS232 snelheden.

bits per seconde
300
1200
2400
4800
9600
19200
38400
57600
115200



Figuur B.2 : Een voorbeeld van een RS232-protocol. Naast de startbit heeft deze variant acht databits, een even pariteitsbit en twee stopbits.

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 B.1 geeft de standaard snelheden voor RS232-communicatie.

Het schrijven via de RS232-poort

Code B.2 bevat de functies die in code B.1 gebruikt worden om alle 256 bytes van `0` tot `0xFF` via de RS232-poort te versturen. Code B.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 B.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.

De RS232-applicaties in deze bijlage zijn voor de pc. De applicaties voor de Xmega staan in hoofdstuk 19. bij de bespreking van de USART.

Code B.1 : Het versturen van gegevens via de COM-poort (main.c).

```
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 B.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 B.2 : Het versturen van gegevens via de COM-poort (comm.c).

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  char   comm[7];
5  HANDLE h;
6
7  void OpenComm(char *c)
8  {
9    DCB dcb;
10
11   strcpy(comm, c);
12   if ( (h = CreateFile(comm, GENERIC_READ|GENERIC_WRITE,
13                             0, NULL, OPEN_EXISTING,
14                             FILE_ATTRIBUTE_NORMAL, NULL )) == INVALID_HANDLE_VALUE) {
15     printf("Can't open comm port: %s\n", comm);
16     exit(1);
17   }
18
19   if ( SetupComm(h, 4096, 4096) == 0 ) {
20     printf("Can't setup comm port: %s\n", comm);
21     exit(1);
22   }
23
24   if ( GetCommState(h, &dcb) == 0 ) {
25     printf("Can't get state comm port: %s\n", comm);
26     exit(1);
27   }
28
29   dcb.BaudRate = 115200;
30   dcb.ByteSize = 8;
31   dcb.Parity = NOPARITY;
32   dcb.StopBits = 0;
33
34   if ( SetCommState(h, &dcb) == 0 ) {
35     printf("Can't set state comm port: %s\n", comm);
36     exit(1);
37   }
38 }
39
40 void CloseComm()
41 {
42   CloseHandle(h);
43 }
44
45 void WriteCommByte(unsigned char b)
46 {
47   unsigned long d;
48
49   if ( WriteFile(h, &b, 1, &d, NULL) == 0 ) {
50     printf("Can't write to comm port: %s\n", comm);
51     exit(1);
52   }
53 }
```

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 B.2 kan de functie `ReadCommByte` uit code B.3 worden toegevoegd, die met behulp van de functie `ReadFile` een byte leest van de RS232-poort.

Code B.3 : Het ontvangen van gegevens via de COM-poort (aanvulling comm.c).

```
55  void ReadCommByte(char *buf)
56  {
57    unsigned long n;
58
59    ReadFile(h, buf, 1, &n, NULL);
60  }
```

Code B.4 past deze functie toe om een karakter van de COM-poort te lezen en de hexadecimale waarde van de byte af te drukken op het scherm.

Code B.4 : 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  }
```

Een RS232-bibliotheek

In code B.5 staat een communicatieprogramma dat gebruik maakt van de RS232-bibliotheek van Teunis van Beelen. Het programma leest via de standaardinvoer een karakter, stuurt dit karakter door naar een Xmega, die een string terugstuurt en die tenslotte deze string op het scherm afdrukt.

Code B.5 : Communicatie via een COM-poort met behulp van een RS232-bibliotheek.

De RS232-bibliotheek van Teunis van Beelen staat op http://www.teuniz.net/RS-232/.

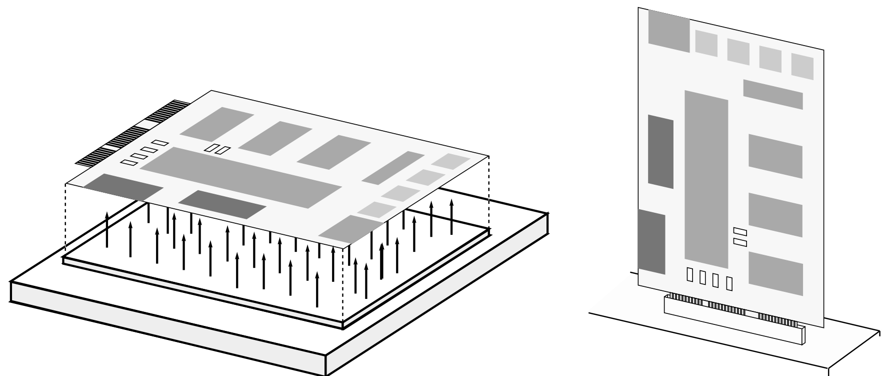
Deze RS232-bibliotheek bestaat uit c- en een h-bestand: `rs232.c` en `rs232.h`.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "rs232.h"
4  #include <Windows.h>
5  #define _delay_ms(__ms)  Sleep((__ms))
6
7  int main(int argc, char *argv[])
8  {
9    int  commNumber = 0;      //  /dev/ttyS0 (COM1 on windows)
10   int  commBaud   = 9600;   //  9600 baud
11   char commMode[] = "8N1";
12   char commBuffer[4096];
13
14   char line[256];
15   int  n;
16   unsigned char c = 0;
17
18   // no buffer for stdout
19   setbuf(stdout, NULL);
20
21   // read number com-port and baud rate
22   if (argc < 3) {
23     printf("usage: %s COM<n> <baudrate>\n", argv[0]);
24     return 0;
25   }
26   sscanf(argv[1], "COM%d", &commNumber);
27   commNumber--;
28   sscanf(argv[2], "%d", &commBaud);
29
30   // open com-port
31   if ( RS232_OpenComport(commNumber, commBaud, commMode) ) {
32     printf("Can not open comport\n");
33     return(0);
34   }
35
36   printf("Demo RS232 Program\n");
37   while (1)  {
38     // read char from stdin
39     printf("> ");
40     if ( fgets(line, 255, stdin) != NULL ) {
41       c = line[0];
42     }
43
44     // send char read
45     RS232_SendByte(commNumber, c);
46
47     // wait for response and print response
48     _delay_ms(20);
49     n = RS232_PollComport(commNumber, (unsigned char*) commBuffer, 4096);
50     commBuffer[n] = '\0';
51     printf("%s", commBuffer);
52   }
53
54   return 0;
55 }
```

C

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 C.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.

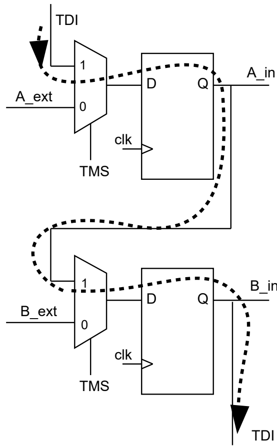


Figuur C.1 : **Het testen van elektronica.** In de linker figuur staat een *bed of needles*. Hiermee worden de verbindingen tussen de componenten op een PCB getest. In deze opstelling wordt de fysieke structuur getest, of de verbindingen wel of niet aanwezig zijn. In de rechter figuur zijn alle normale in- en uitgangen van de PCB aangesloten op een testapparaat. Hiermee kan de elektronica functioneel worden getest.

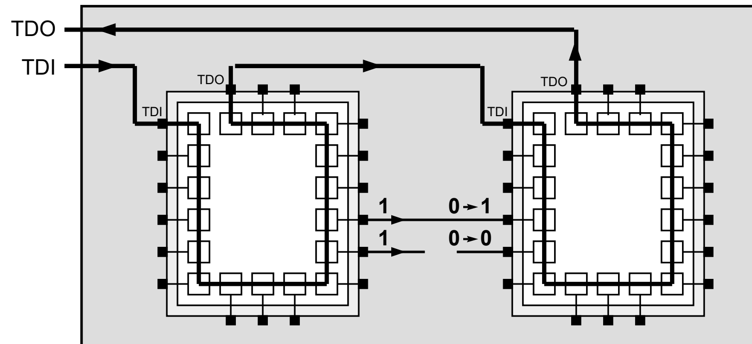
Philips is altijd zeer actief geweest met JTAG. Het bedrijf JTAG Technologies in Eindhoven is een spin-off van de vroegere activiteiten van Philips en is prominent op het gebied van *boundary scan*.

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 modi:

de testmodus en de normale modus. In de testmodus kan via de multiplexer een testsignaal worden aangeboden, zie figuur C.2. Door alle in- en uitgangsflipfloppen 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 C.3.



Figuur C.2 : Twee *boundary scan*-flipfloppen. In testmode (TMS hoog) is er een *scan path*, waar langs de testsignalen kunnen worden ingeklokt.



Figuur C.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.

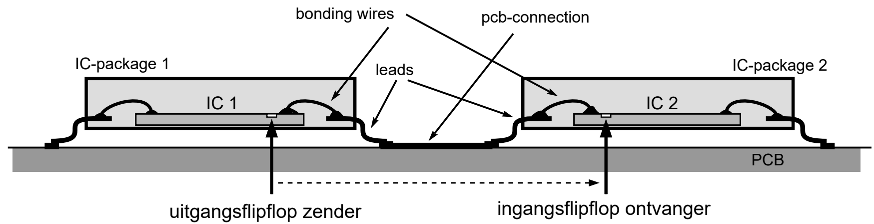
Op de PCB kan deze lange seriële verbinding — de *boundary scan* — gebruikt worden om testdata in en uit te lezen. Door een bekend testpatroon van enen en nullen het scanpad (*scan path*) in te schuiven en vervolgens de schakeling een klokslag normaal te laten functioneren, komt er via de PCB verbindingen 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 C.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:

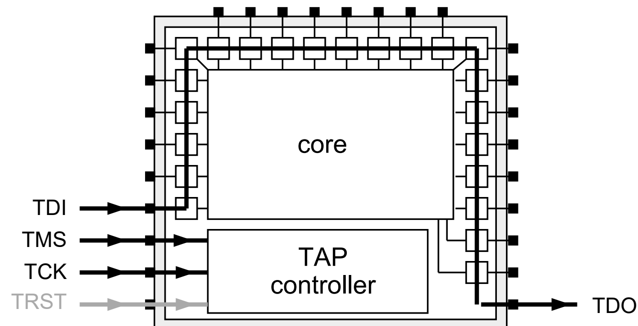
`TDI`	Test Data In
`TDO`	Test Data Out
`TMS`	Test Mode Select
`TCK`	Test ClocK
`TRST`	Test ReSeT

Het resetsignaal is niet altijd aanwezig en is ook niet per se noodzakelijk. Figuur C.5 toont een IC met de TAP-controller en de vijf aansluitingen.



Figuur C.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.

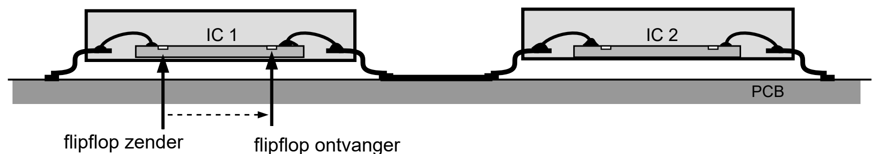
De *core* van een IC is het deel van het silicium zonder de in- en uitgangscellen. Het is de binnenkant — het hart — van het IC en bevat het functionele gedrag.



Figuur C.5 : Een IC met boundary scan en een TAP-controller.

Andere mogelijkheden met JTAG

JTAG maakt een IC serieel toegankelijk voor testvectoren. Figuur C.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 C.6. Vaak worden er logica en



Figuur C.6 : De *intest* bij een IC op een PCB.

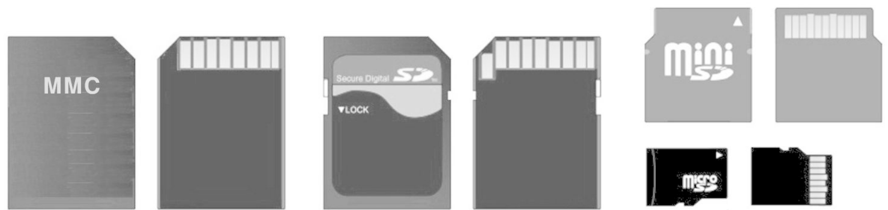
flipfloppen aan een IC toegevoegd om het volledig testbaar te maken. De registers en flipfloppen in de *core* van het IC worden aan het *scan path* toegevoegd. Als alle flipfloppen bereikbaar en leesbaar zijn is het IC volledig te testen. Het intern testen van IC's noemt men een *intest*

De toegankelijkheid via *boundary scan* kan ook gebruikt worden om een component *in system* te programmeren en te debuggen. Verschillende Xmega's van Atmel hebben 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. De Xmega256a3u heeft een JTAG-aansluiting bij het hoogste nibble van poort B. Bij een nieuwe Xmega staat JTAG standaard aan. Bij Het Xmega-bord staat de JTAG-interface uit.

D

SD-kaart

Het Xmega-bord, dat bij dit boek gebruikt is, heeft een microSD-kaarthouder. Een microSD-kaart is een geheugenkaart op basis van flash en wordt bijvoorbeeld veel toegepast bij mobiele telefoons. De microSD-kaart is een kleinere variant van de gewone SD-kaart en is verkrijgbaar in verschillende varianten. De gewone microSD-kaart heeft een geheugenlimiet van 2 GB. De microSDHC heeft een capaciteit van maximaal 32 GB en de microSDXC is bestemd voor capaciteiten groter dan 32 GB. In figuur D.1 staan voorbeelden van diverse SD-kaarten.

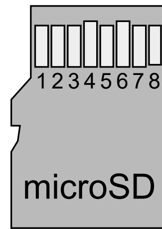


Figuur D.1 : De voor- en achterkant van verschillende geheugenkaarten. Links staan de gewone MMC- en SD-kaart en rechts de miniSD- en de microSD-kaarten.

Het formaat microSD-kaart is veel kleiner dan dat van een gewone SD-kaart. De SD-kaart is in 1999 door SanDisk, Matsushita en Toshiba ontwikkeld als alternatief voor de MMC-kaart, die in 1997 door onder andere Siemens is bedacht. SD staat voor *Secure Digital* en MMC staat voor *Multi Media Card*. Beide kaarten zijn grotendeels compatibel. Het belangrijkste verschil is dat de SD-kaart een veiligheidsschuif heeft en dat alle SD-kaarten beschikken over de mogelijkheid voor data-encryptie. Met een speciale adapter is de microSD-kaart ook als SD-kaart te gebruiken.

De microSD-kaart heeft net als de andere SD- en MMC-kaarten een SPI-interface om gegevens naar de kaart te sturen en om gegevens van de kaart te lezen. In figuur D.2 staan de SPI-aansluitingen van de microSD-kaart.

Bestandssystemen
Een bestandssysteem is een systeem voor de opslag en organisatie van computerbestanden op een medium zodanig dat deze op een eenvoudige en efficiënte manier te vinden en te benaderen zijn. Voorbeelden van media zijn: interne en externe harde schijven, floppy disks, cd's, dvd's, USB-sticks, solid-state disks en SD-kaarten.



Pin	SD	SPI
1	DAT2	
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	

Figuur D.2 : De aansluitingen bij een microSD-kaart. Links staat een tekening van de achterkant met de genummerde aansluitingen. Rechts staat een overzicht met de betekenis van de aansluitingen voor de gewone modus en voor de SPI-modus.

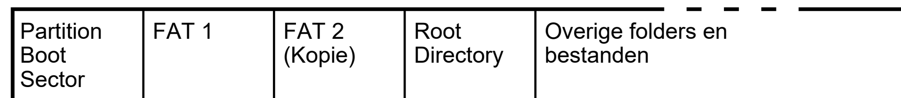
Voor de meeste applicaties met een SD-kaart is een bestandssysteem nodig. De microSD-kaart wordt standaard geleverd met een FAT32-bestandssysteem. Mits de microcontroller in staat is om met dit bestandssysteem om te gaan, kunnen bestanden gemaakt, gelezen, geschreven, gekopieerd en verwijderd worden en kan bijvoorbeeld de inhoud van folders bekeken worden.

Er bestaan veel verschillende bestandssystemen. Standaard kent Windows FAT32, NTFS en exFAT.
NTFS, *New Technology File System*, gebruikt Microsoft bij interne harde schijven. ExFAT is een verbeterde versie van FAT32.

Het FAT32-bestandssysteem is een variant op FAT16 en FAT12 met meer geheugen, grotere clusters en een grotere maximale bestandsgrootte. Naast FAT32 is NFTS een veel gebruikt bestandssysteem. Mede omdat microSD-kaarten standaard worden geleverd met FAT32, is dit bij deze kaarten het meest gebruikte bestandssysteem.

FAT staat voor *File Allocation Table* en is een opzoektabel met verwijzingen in welke clusters en sectoren van het geheugen de bestanden en folders staan, zodat de bestanden eenvoudig en snel gevonden kunnen worden.

De Partition Boot Sector noemt men ook wel: Partition Boot Record, BIOS Parameter Block, Drive Parameter Block of Reserved Sector.



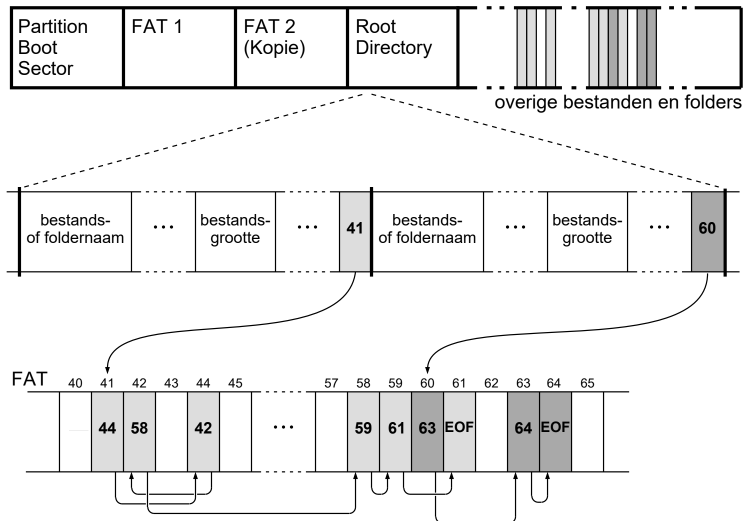
Figuur D.3 : De indeling van het geheugen bij een FAT32-bestandssysteem.

Figuur D.3 geeft de indeling van een FAT32-bestandssysteem. Het eerste deel van het geheugen bestaat uit een bootsector, een FAT, een kopie van de FAT en de hoofdfolder. In de rest van het geheugen staan de overige folders en alle bestanden. In de FAT staan de nummers van de sectoren waar de bestanden en folders staan. Grotere bestanden gebruiken meerdere sectoren. Deze sectoren hoeven niet aangesloten te zijn. In de FAT wordt bijgehouden welke sectoren gebruikt worden. Figuur D.4 geeft een voorbeeld hoe twee grote bestanden in het geheugen kunnen staan.

Bestanden kunnen dus gefragmenteerd zijn over het geheugen. Windows kent de mogelijk om het geheugen te defragmenteren, zodat de bestanden op aaneengesloten sectoren staan.

Keuze FAT-bibliotheek

Het schrijven van een systeem voor een FAT is niet eenvoudig. Het vereist veel kennis van het FAT-systeem. Gelukkig zijn er veel oplossingen te vinden. Bij



Figuur D.4 : Een voorbeeld van een toepassing met een FAT.

De code van de FatFs is met de uitgebreide documentatie te vinden via de website van Chan http://elm-chan.org/fsw/ff/00index_e.html

kleine embedded systemen is het FAT-bestandssysteem FatFS van Chan zeer populair.
Op de site van Chan staat ook een zipbestand met voorbeelden voor verschillende microcontrollers. Hoewel de zip twee voorbeelden voor AVR-microcontrollers bevat, ontbreekt die voor de Xmega.

Meer informatie over het FAT-systeem van Roland Riegel staat op http://www.roland-riegel.de/sd-reader/

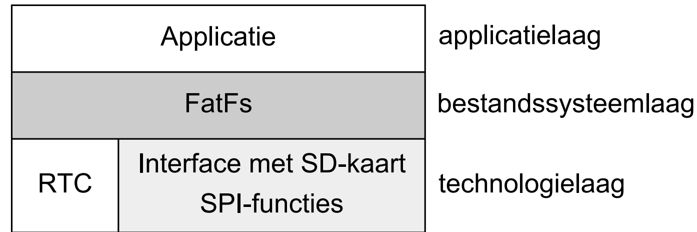
Voor de AVR-microcontrollers bestaan verschillende alternatieve FAT-systemen, onder andere één van Roland Riegel voor de ATmega en een variant voor de Xmega128a3u van Martin Junghans. De bibliotheek van Riegel is moderner van opzet dan die van Chan. Chan gebruikt bijvoorbeeld eigen definities voor integers en Riegel gebruikt de standaarddefinities uit `inttypes.h`. Geen van de implementaties is direct toepasbaar bij het Xmega-bord. In alle gevallen moet de SPI worden aangepast.

Belangrijke voordelen van de FatFS van Chan zijn dat deze bibliotheek veel gebruikt wordt, dat de bibliotheek onderhouden wordt, dat er een forum is en dat er veel documentatie is. De status van het FAT-systeem van Riegel is daarentegen minder duidelijk.

FatFs - Generic FAT File System Module

Het FatFs van Chan is gestructureerd opgezet en het bestaat uit twee lagen: het feitelijke bestandssysteem voor de bestandsmanipulaties en een laag voor de communicatie met de geheugenkaart, harde schijf of ander opslagmedium, zie ook figuur D.5. In het geval van de SD-kaart bestaat deze onderste laag uit twee delen, namelijk de functies voor de communicatie tussen het bestandssysteem en de SD-kaart en de functies voor de fysieke communicatie met het opslagsysteem.

De ontwerper is verantwoordelijk voor de applicatielaag en gedeeltelijk voor de technologielaag. Chan beschrijft de bestandssysteemlaag in `ff.c` en de technologielaag staat in principe in `diskio.c`. De voorbeelden van Chan bevatten een aantal implementaties voor verschillende media en embedded systemen.



Figuur D.5 : De verschillende abstractielagen bij FatFs. De ontwerper is verantwoordelijk voor de applicatielaag en gedeeltelijk voor de technologielaag.

Op de site van het boek staan aanpassingen van `sdmm.c` en `mmc_avr.c` die direct bij het Xmega-bord gebruikt kunnen worden.

Deze aanpassingen zijn gebaseerd op de voorbeelden van release 11a. Chan heeft bij release 12 de voorbeelden aangepast. Bestand `mmc_avr.c` is vervangen door `mmc_avr_uart.c` en `mmc_avr_bb.c`. Het eerste bestand gebruikt de UART in SPI-modus en het tweede gebruikt *bit banging*.

De voorbeelden van Chan zijn niet direct geschikt voor het Xmega-bord. De voorbeelden `generic` en `avr` zijn relatief eenvoudig aan te passen. Het bestand `diskio.c` is bij `generic` hernoemd als `sdmm.c`. Dit is een zeer basale implementatie, waarbij de benodigde tijdvertragingen gebaseerd zijn op delay-functies. Het voorbeeld `avr` bevat meerdere implementaties van `diskio.c`, ondermeer een bestand `mmc_avr.c`. Deze implementatie is veel uitgebreider dan `sdmm.c` en gebruikt een timer voor de tijdvertragingen.
De headerfile `ffconf.h` bevat een dertigtal opties, die bepalen welke functies en functionaliteiten geïmplementeerd worden. Bijvoorbeeld het gebruik van stringfuncties, zoals de afdrukfunctie `f_printf`, het gebruik van lange bestandsnamen en het wel of niet gebruiken van een RTC.

Voorbeelden van toepassingen met FatFs voor de microSD-kaart

Het hoofdprogramma uit code D.1 stelt met de functie `init_clock()` de klok in op 32 MHz, initialiseert met `init_stream()` de seriële verbinding via USART0 van poort F en roept vóór de oneindige while de functie `demo_ff()` aan. Deze demonstratie van het gebruik van de SD-kaart wordt eenmalig uitgevoerd.

Code D.1 : Hoofdprogramma voor testen FatFs.

```
 1  #define F_CPU 32000000UL
 2
 3  #include <avr/interrupt.h>
 4  #include "clock/clock.h"
 5  #include "uart/stream.h"
 6
 7  int demo_ff(void);
 8
 9  int main (void)
10  {
11    init_clock();
12    init_stream(F_CPU);
13    sei();
14
15    demo_ff();
16
17    while(1) {}  // do nothing
18  }
```

In code D.2 staat de functie `demo_ff()`, die op de microSD-kaart een bestand *demo.txt* opent, vervolgens hier tekst naar schrijft, het bestand sluit, opnieuw opent in de alleen-lezen modus, de inhoud leest en naar de UART stuurt.

Code D.2 : Functie met demonstratie van het lezen van en schrijven naar een SD-kaart.

```
1  #include <stdio.h>
2  #include "ff/ff.h"
3
4  char    buffer[256];
5  FATFS FatFs;          //! FatFs work area needed for each volume
6  FIL   Fil;            //! File object needed for each open file
7
8  int demo_ff(void)
9  {
10   UINT    bw;
11   uint8_t ret;
12
13   printf("Demo FatFs\n");
14   if ( (ret = f_mount(&FatFs, "", 1)) != FR_OK ) {
15     printf("Demo aborted: device not found\n");
16     return ret;
17   }
18
19   // create and write file demo.txt
20   if ((ret = f_open(&Fil, "demo.txt", FA_WRITE|FA_CREATE_ALWAYS)) !=
21                                                               FR_OK) {
22     printf("Demo aborted: can’t create file\n");
23     return ret;
24   }
25
26   printf("DEMO : write file\n");
27   f_write(&Fil, "Some text written!\r\n", 20, &bw);
28   f_printf(&Fil, "Printf %s\n", "works as well");
29   f_printf(&Fil, "Variabele bw is %d", bw);
30
31   f_close(&Fil);
32
33   // read file demo.txt
34   if ((ret = f_open(&Fil, "demo.txt", FA_READ)) != FR_OK) {
35     printf("Demo aborted: can’t open file\n");
36     return ret;
37   }
38
39   while ( f_gets(buffer, 255, &Fil) != NULL ) {
40     printf("\t%s", buffer);
41   }
42
43   f_close(&Fil);
44
45   return FR_OK;
46 }
```

Er is geen modusvlag om een bestand te openen en daar dan gegevens aan toe te voegen. Als alleen `FA_WRITE` aan `f_open` wordt meegegeven staat de filepointer altijd aan het begin van het bestand. Chan geeft een oplossing op: http://elm-chan.org/fsw/ff/img/app1.c De functie `open_append` gebruikt de functies `f_lseek` en `f_size` om naar het einde van het bestand te gaan.

De datastructuur `FatFs` legt de gegevens over het bestandssysteem vast. De functie `f_mount` vult deze datastructuur. De datastructuur `Fil` legt de gegevens over het bestand vast. Deze is vergelijkbaar met de datastructuur waar een *gewone* filepointer naar wijst. De functie `f_open` vult deze datastructuur.

Code D.2 gebruikt twee functies om tekst te schrijven. De functie `f_write` schrijft 20 karakters uit de string naar de SD-kaart en na afloop staat in de variabele `bw` het aantal geschreven bytes. De functie `f_printf` werkt op dezelfde wijze als de

gewone functie `printf`. De functies `f_open` en `f_gets` komen overeen met de `fopen` en `fgets` van de `stdio`-bibliotheek. Om deze functies te gebruiken moet de definitie `_USE_STRFUNC` uit `ffconf.h` ongelijk aan `0` zijn.

Code D.1 gebruikt een — voor het Xmega-bord — aangepaste versie van `sdmm.c` uit het voorbeeld `generic` van Chan. De functie `demo_ff` kan ook worden gebruikt bij het aangepaste voorbeeld `mmc_avr.c` uit het voorbeeld `avr` van Chan. Bij deze oplossing is een timer nodig voor de tijdvertragingen. In code D.3 staat een hoofdprogramma dat een timer en de aangepaste versie van `mmc_avr.c` gebruikt. Er is voor timer/counter C1 gekozen. De PWM-aansluitingen van deze timer kunnen toch niet gebruikt worden, omdat deze bezet zijn door de SD-kaart en de draadloze module.

De aangepaste versies van `sdmm.c` en `mmc_avr.c` staan op de internetsite van dit boek.

Code D.3 : Hoofdprogramma voor testen FatFs met timer voor de tijdvertragingen.

```
 1  #define F_CPU 32000000UL
 2
 3  #include <avr/io.h>
 4  #include <avr/interrupt.h>
 5  #include "ff/ff.h"
 6  #include "ff/diskio.h"
 7  #include "clock/clock.h"
 8  #include "uart/stream.h"
 9
10  void init_timer(void);
11  void demo_ff(void);
12
13  int main (void)
14  {
15    init_clock();
16    init_stream(F_CPU);
17
18    init_timer();
19    PMIC.CTRL |= PMIC_LOLVLEN_bm;
20    sei();
21
22    demo_ff();
23
24    while(1) {}  // do nothing
25  }
26
27  // Initialize 100 Hz timer
28  void init_timer(void)
29  {
30    TCC1.CTRLB     = TC_WGMODE_NORMAL_gc;
31    TCC1.CTRLA     = TC_CLKSEL_DIV8_gc;
32    TCC1.INTCTRLA  = TC_OVFINTLVL_LO_gc;
33    TCC1.PER       = 39999;
34  }
35
36  ISR(TCC1_OVF_vect)
37  {
38    disk_timerproc();  // Update timerinfo, see mmc_avr.c
39  }
```

E

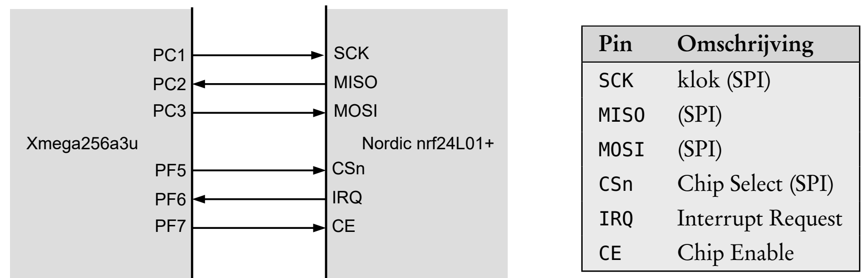
Draadloze module

De nRF24L01+ wordt soms nRF24L01p genoemd en is de opvolger van de nRF24L01.

Het Xmega-bord, dat bij dit boek gebruikt is, heeft een draadloze module, die gebaseerd is op de Nordic nRF24L01+. Deze component is een zender en ontvanger, die werkt in de ISM frequentieband tussen 2,400 en 2,4835 GHz en die gebruik maakt van een eigen *Enhanced Shockburst* protocol.

De aansluitingen van de nRF24L01+

De nRF24L01+ heeft een SPI voor de configuratie en het versturen en het ontvangen van de gegevens. Figuur E.1 toont de aansluitingen tussen de Xmega256a3u en de nRF24L01+ op het Xmega-bord. De klok, de MOSI en de MISO van de SPI zijn verbonden met `UARTC0` van de Xmega, die in de SPI-modus werkt. De andere drie aansluitingen zijn aangesloten met pin 5, 6 en 7 van poort F.



Pin	Omschrijving
`SCK`	klok (SPI)
`MISO`	(SPI)
`MOSI`	(SPI)
`CSn`	Chip Select (SPI)
`IRQ`	Interrupt Request
`CE`	Chip Enable

Figuur E.1 : **De aansluitingen op het Xmega-bord tussen de Xmega256a3u en de nRF24L01+.** Links staat een tekening met de aansluitingen en rechts een overzicht met de betekenis van de aansluitingen. De Xmega is de master en de nRF24L01+ is de slave.

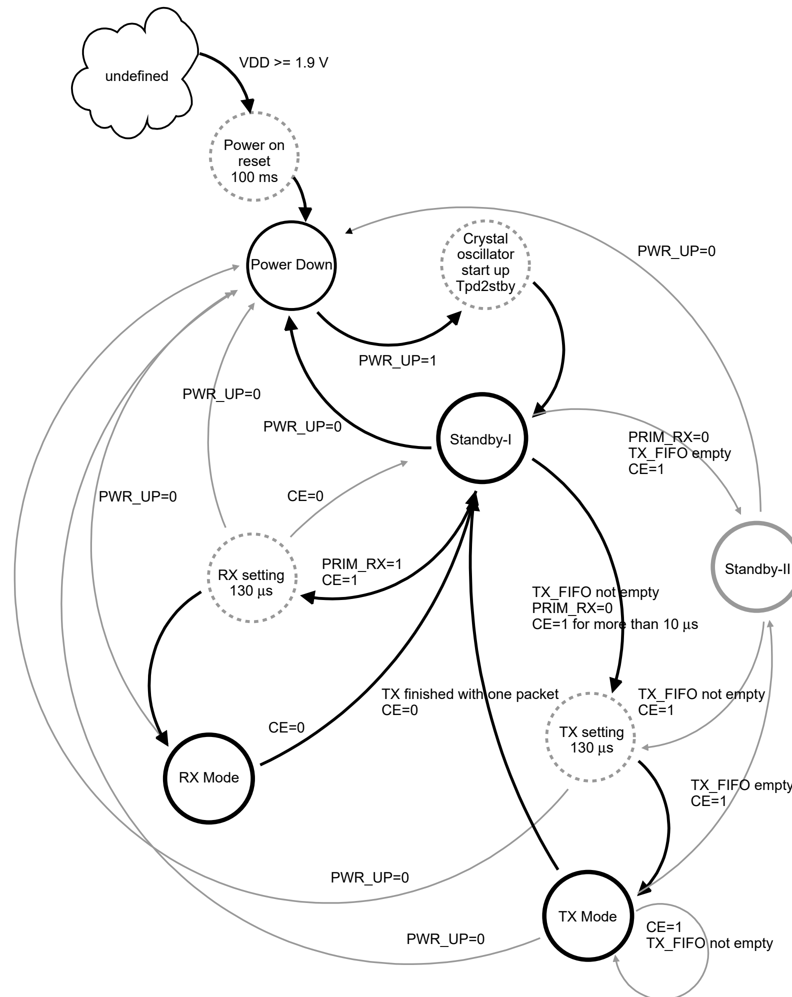
De SPI-opdrachten en de register map

De elf SPI-commando's staan in tabel 20 van de datasheet en in tabel 28 staat de *register map*.

De nRF24L01+ kent elf SPI-opdrachten. Opdracht `W_REGISTER` schrijft gegevens naar de *register map* en opdracht `R_REGISTER` leest gegevens uit de *register map*. De *register map* bestaat uit ruim dertig registers en bevat alle instellingen voor het ontvangen en zenden van gegevens.

Instellingen van de nRF24L01+ zijn onder andere: de sterkte en de datasnelheid van het radiosignaal, het kanaal van de radioverbinding, wel of geen *acknowledge* en het gebruik van dynamische *payloads*. Daarnaast regelt men via de *register map* de *power up* en de modus van de radiomodule, de instellingen van de fifo's en de status van interruptbits.

De te versturen en te ontvangen gegevens worden met fifo's gebufferd, die met behulp van een van de andere SPI-commando's worden gelezen en geschreven.



Figuur E.2 : Het toestandsdiagram van de radiomodule van de nRF24L01+. Dit diagram komt overeen met figuur 4 uit de datasheet. De vijf toestanden en de belangrijkste toestandsovergangen hebben een getrokken lijn. De toestanden met onderbroken lijnen zijn tussentoestanden waarin de toestandsmachine van de radiomodule zich kan bevinden.

De toestanden van de radiomodule

De radiomodule van de nRF24L01+ kent vijf verschillende toestanden:

Power Down Dit is de slaaptoestand van de radiomodule, waarin de module automatisch terecht komt na het opstarten. In deze toestand zijn alle registers via SPI bereikbaar en is de radiomodule uit.

Standby-I In deze toestand is de radiomodule opgestart met een minimale dissipatie.

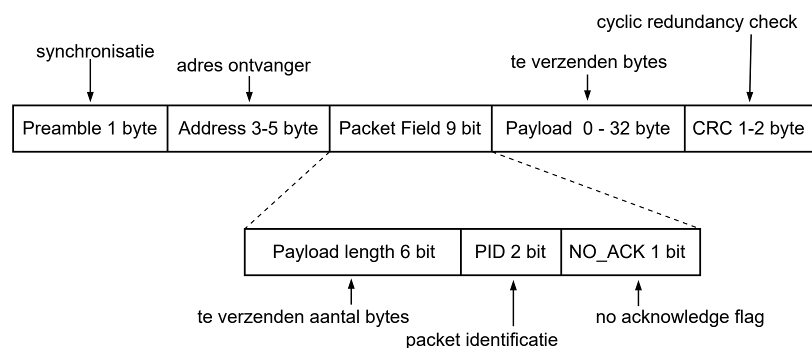
Standby-II Deze toestand is identiek aan Standby-I, maar hierbij zijn de klokbuffers actief en wordt er meer vermogen gedissipeerd.

RX-Mode In deze toestand kan de radiomodule informatie ontvangen en zet deze gegevens in één van de fifo's voor het ontvangen.

TX-Mode In deze toestand verstuurt de radiomodule de gegevens die in de fifo's voor het verzenden staan.

In de RX-mode werkt de nRF24L01+ als ontvanger. De nRF24L01+ demoduleert daarbij de signalen van het betreffende RF-kanaal en zoekt naar een geldig datapakket en zet dat in één van de fifo's voor het ontvangen. Om in deze toestand te komen moeten het `PWR_UP`- en het `PRIM_RX`-bit uit het `CONFIG`-register van de *register map* hoog zijn en moet ingang `CE` eveneens hoog zijn.

In de TX-mode verstuurt de nRF24L01+ datapakketten, die in de fifo's voor het verzenden staan. Nadat alle pakketten verzonden zijn, komt de toestandsmachine automatisch in Standby-II. Om in de TX-mode te komen, moet het `PWR_UP`-bit hoog zijn, het `PRIM_RX`-bit laag zijn, de fifo's voor het verzenden niet leeg zijn en de ingang CE minimaal 10 ms hoog zijn.



Figuur E.3 : Het datapakket bij het Enhanced ShockBurst protocol.

Enhanced Shockburst protocol

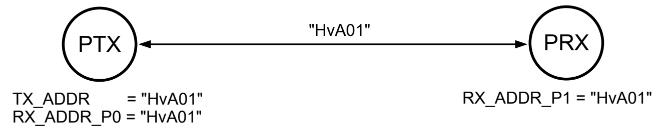
Nordic gebruikt een eigen protocol dat op een efficiënte manier gebruik maakt van de radiomodule en noemt dit protocol *Enhanced ShockBurst*. Het datapakket van dit protocol is in figuur E.3 getekend en bestaat uit een *preamble* voor de synchronisatie, het adres van de ontvanger, een 9-bits veld met onder andere het aantal te verzenden bytes, de *payload* met de te verzenden bytes en een *cycle redundancy check*.

Enhanced Shockburst kan met en zonder auto-acknowledge communiceren. Bij de meeste toepassingen zal het `NO_ACK`-bit laag zijn en werkt het protocol als volgt:

- De zender begint een transactie in de TX-mode met het versturen van een datapakket naar de ontvanger. Na het versturen schakelt de zender automatisch over naar de RX-mode voor de ontvangst van het ACK-pakket.
- De ontvanger is de RX-mode. Na de ontvangst van een datapakket gaat de ontvanger naar de TX-mode, stuurt een ACK-pakket terug naar de zender en gaat weer terug naar de RX-mode.
- Als de ontvanger niet op tijd een ACK-pakket ontvangt, stuurt het automatisch het originele datapakket opnieuw naar de ontvanger en wacht weer op een ACK-pakket. Deze procedure herhaalt zich totdat een ACK-pakket is ontvangen of het maximale aantal *retransmits* of *retries* is bereikt.

Data pipes

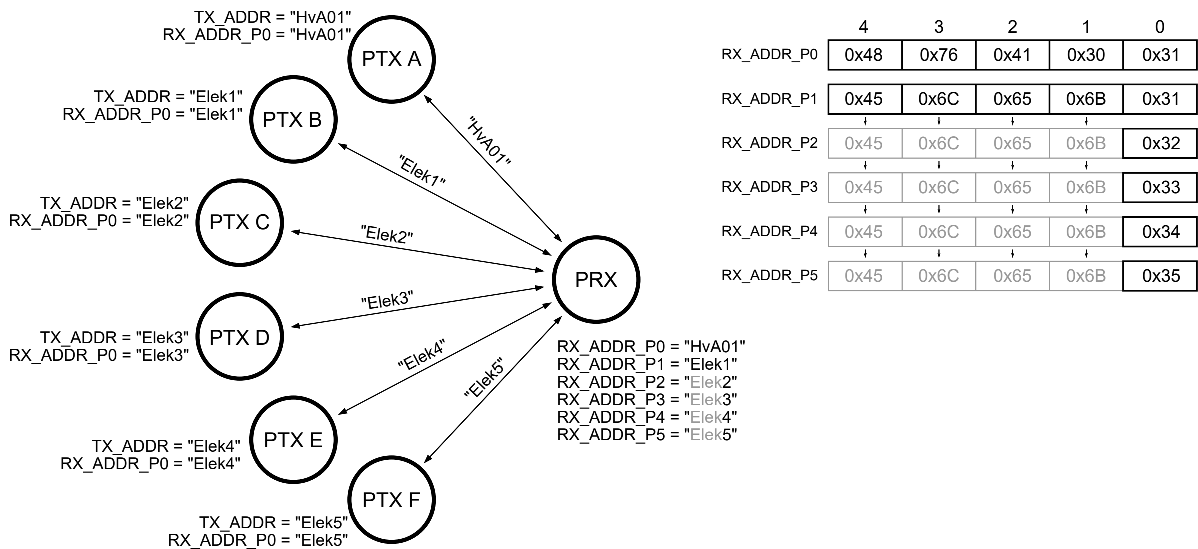
Tussen een zender en een ontvanger bevindt zich een zogenoemde *pipe*. De nRF24L01+ kan berichten van zes verschillende zenders ontvangen. In de ontvangstmodus gebruikt het daarvoor zes verschillende *pipes*. Iedere *data pipe* heeft zijn eigen fysieke adres, dat uit drie tot vijf bytes bestaat.



Figuur E.4 : Een *data pipe* tussen een zender PTX en een ontvanger PRX.

In figuur E.4 staat de *data pipe* tussen een zender en één enkele ontvanger. Het adres is in dit geval `0x4876413031` oftewel `HvA01`.

Het TX-adres van de zender (PTX) is gelijk aan het adres van de *pipe*. De ontvanger stuurt automatisch een ACK-pakket terug naar de zender nadat het een datapakket heeft ontvangen en heeft gevalideerd. Om het teruggestuurde ACK-pakket te kunnen ontvangen, moet één van de RX-adressen van de zender ook gelijk zijn aan het adres van de *pipe*. In figuur E.4 is het adres `RX_ADDR_P0` daarom ook gelijk aan `HvA01`. Bij de ontvanger (PRX) moet één van de RX-adressen overeenkomen met dat van de *pipe*. In dit voorbeeld is dat `RX_ADDR_P1`



	4	3	2	1	0
RX_ADDR_P0	0x48	0x76	0x41	0x30	0x31
RX_ADDR_P1	0x45	0x6C	0x65	0x6B	0x31
RX_ADDR_P2	0x45	0x6C	0x65	0x6B	0x32
RX_ADDR_P3	0x45	0x6C	0x65	0x6B	0x33
RX_ADDR_P4	0x45	0x6C	0x65	0x6B	0x34
RX_ADDR_P5	0x45	0x6C	0x65	0x6B	0x35

Figuur E.5 : Een ontvanger die een dataverbinding heeft met zes verschillende zenders. Iedere verbinding heeft zijn eigen *pipe address*. Bij zenders staat dit adres in `TX_ADDR` en vanwege de *acknowledge* ook in `RX_ADDR_P0`. Bij de ontvanger worden alle zes RX-adressen gebruikt. De eerste vier bytes van de adressen `RX_ADDR_P2` tot en met `RX_ADDR_P5` zijn identiek met die van `RX_ADDR_P1`.

De nRF24L01+ kan — zoals eerder is gezegd — van maximaal zes verschillende zenders informatie ontvangen. In figuur E.5 is dit gevisualiseerd. De ontvanger PRX gebruikt zes *pipes* voor de dataverbindingen met de zes zenders, waarbij het betreffende RX-adres van de ontvanger dan overeenkomt met het TX-adres van de bijbehorende zender.

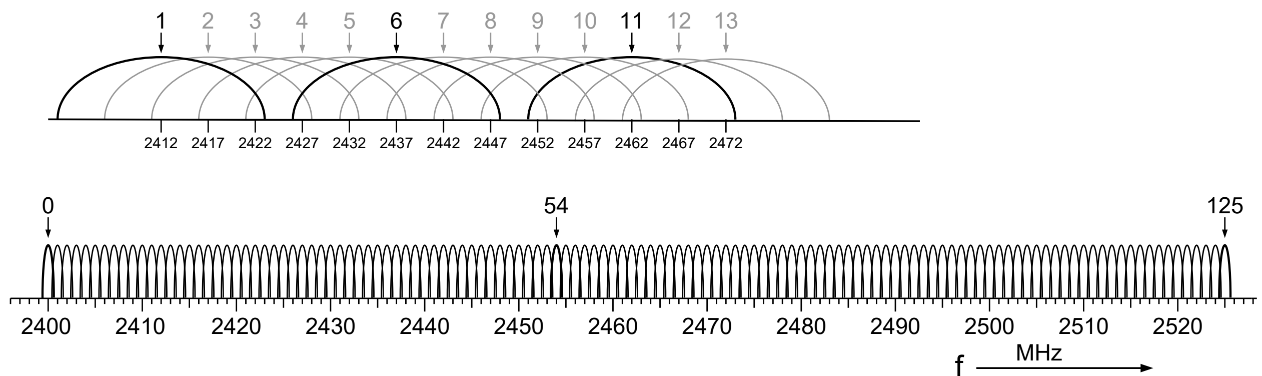
In figuur E.5 gebruikt de ontvanger alle zes RX-adressen. De registers van de adressen RX_ADDR_P0 en RX_ADDR_P1 bestaan uit vijf bytes. Die van RX_ADDR_P2 tot en met RX_ADDR_P5 zijn maar één byte groot. Alleen de minst significante byte kan worden ingesteld. De andere bytes zijn identiek met die van RX_ADDR_P1. Daarom zijn de eerste vier bytes van de adressen bij de zenders B tot en met F hetzelfde.

De kanalen bij de nRF24L01+

De nRF24L01+ is ontworpen voor communicatie in de ISM-frequentieband tussen 2,4000 en 2,4835 GHz. Deze frequentieband is vrij te gebruiken en wordt ook bij Bluetooth en bij wifi gebruikt. Het frequentiegebied van de nRFL2401+ is breder en loopt van 2,400 GHz tot en met 2,525 GHz, en bestaat uit 126 kanalen. Voor de frequentie f van een kanaal geldt:

$$f = 2400 + n \qquad [\text{MHz}] \tag{E.1}$$

waarbij n het kanaalnummer is dat varieert van 0 tot en met 125. De bandbreedte hangt af van de datasnelheid en is kleiner dan 1 MHz bij 250 Kbps en bij 1 Mbps en kleiner dan 2 MHz bij een datasnelheid van 2 Mbps.



Figuur E.6 : De kanalen van de nRF24L01+ vergeleken met de wifi-kanalen. De nRF24L01+ heeft 126 niet overlappende kanalen van 2,400 GHz tot en met 2,525 GHz. Een deel van deze kanalen vallen samen met de 13 wifi-kanalen. wifi heeft maximaal drie niet overlappende kanalen, namelijk bij kanaal 1, 6 en 11.

Figuur E.6 toont de 126 kanalen van de nRF24L01+ en de 13 wifi-kanalen. Wifi gebruikt DSSS, *Direct-Sequence Spread Spectrum*, waardoor de frequentiebanden veel breder zijn dan bij de nRF24L01+, die GFSK als modulatietechniek gebruikt. Bij de nRF24L01+ zijn de kanalen 0 tot en met 83 vrij te gebruiken. De overige kanalen mogen niet zonder meer in commerciële producten toegepast worden.

GFSK staat voor Gaussian Frequency Shift Keying.

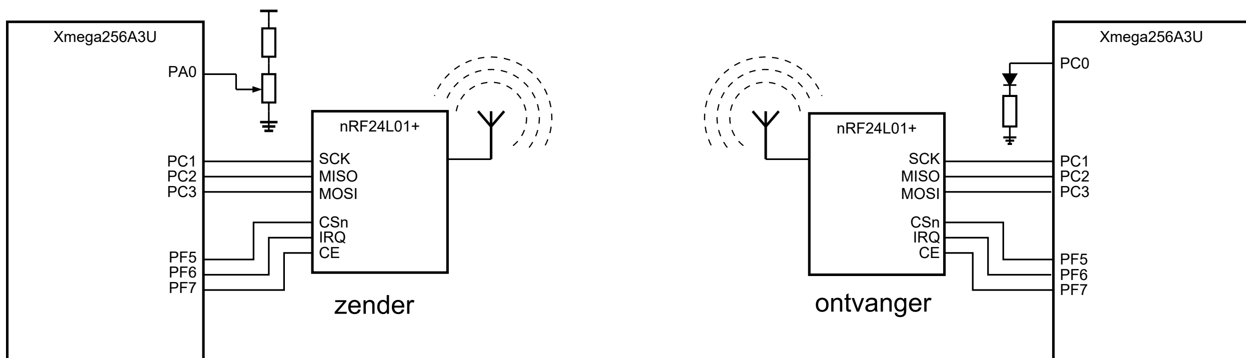
Een bibliotheek voor de nRF24L01+

Het gebruik van de nRF24L01+ is niet triviaal. De radiomodule heeft veel instellingen en het *Enhanced ShockBurst*-protocol kent ook diverse opties. Het is verstandig om gebruik te maken van een bibliotheek.
Op de internetsite, die bij 'De taal C en de Xmega' hoort, staat een bibliotheek voor de nRF24L01+, die geschikt is voor het Xmega-bord. Deze bibliotheek heeft twee c- en twee h-bestanden. De bestanden `nrf24spiXM2.c` en `nrf24spiXM2.h` bevatten de drivers voor het Xmega-bord en de bestanden `nrf24L01.c` en `nrf24L01.h` bevatten de functies en macro's om de radiomodule van de nRF24L01+ te kunnen

gebruiken. Deze bibliotheek is onder meer gebaseerd op de C++-drivers van J. Coliz en de C++-bibliotheek van S. Brennen Bal.
Dit boek bevat geen uitgebreide beschrijving van de bibliotheek. Meer informatie over deze bibliotheek staat op de internetsite, die bij 'De taal C en de Xmega' hoort.

E.1 Voorbeeld met de nRF24L01+ met een zender en één ontvanger

Figuur E.7 geeft de implementatie van de communicatie uit figuur E.4 tussen een zender en één ontvanger met behulp van twee Xmega-borden.
De zender meet iedere 20 ms met de ADC de analoge spanning van de aansluiting `PA0` en stuurt de gemeten waarde via de nRF24L01+ naar de ontvanger. De ontvanger leest deze analoge waarde en past hiermee de duty-cycle van het PWM-signaal aan, die de led bij aansluiting `PC0` aanstuurt.



Figuur E.7 : Een Xmega256a3u met een nRF24L01+ als zender en een Xmega256a3u met een nRF24L01+ als ontvanger. De zender verstuurt de analoge waarde van ingang `PA0` en de ontvanger gebruikt deze waarde om de intensiteit van de led te regelen.

In code E.1 staat een initialisatiefunctie voor de nRF24L01+ van de zender. Na de functies `nrfspiInit()` en `nrfBegin()` die de SPI en de radiomodule initialiseren volgen zes functies, die de radiomodule een specifieke instelling geven. Deze functies overschrijven deels de instellingen, die door `nrfBegin()` gedaan zijn. In dit geval is het aantal *retransmits* 8 en is de tijd tussen twee pogingen 1000 μs, de versterkingsfactor -6 dBm, de datasnelheid 250 Kbps, de CRC-lengte twee bytes, is 54 het gebruikte kanaal, is de auto-acknowledge actief en zijn de payloads dynamisch.
Nadat de interruptbits laag en de fifo's leeggemaakt zijn, opent de initialisatiefunctie op regel 51 en 52 de *pipes* voor het schrijven van het datapakket en voor het ontvangen van de acknowledge. De globale variabele `pipe` bevat het adres van de *pipe*.

Meer informatie over de mogelijke instellingen van de radiomodule staat in de datasheet en de documentatie van de bibliotheek. Voor een juiste communicatie moet een aantal parameters bij de zender en de ontvanger hetzelfde zijn, zoals: het kanaalnummer, de datasnelheid, de CRC-lengte en het gebruik van de auto-acknowledge en de dynamische payload.

Code E.1 : De initialisatie van nRF24L01+ voor de zender van figuur E.7.

```
33 void init_nrf(void)
34 {
35   nrfspiInit();                                       // Initialize SPI
36   nrfBegin();                                         // Initialize radio module
37
38   nrfSetRetries(NRF_SETUP_ARD_1000US_gc,              // Auto Retransmission Delay: 1000 us
39                 NRF_SETUP_ARC_8RETRANSMIT_gc);        // Auto Retransmission Count: 8 retries
40   nrfSetPALevel(NRF_RF_SETUP_PWR_6DBM_gc);            // Power Control: -6 dBm
41   nrfSetDataRate(NRF_RF_SETUP_RF_DR_250K_gc);         // Data Rate: 250 Kbps
42   nrfSetCRCLength(NRF_CONFIG_CRC_16_gc);              // CRC Check
43   nrfSetChannel(54);                                  // Channel: 54
44   nrfSetAutoAck(1);                                   // Auto Acknowledge on
45   nrfEnableDynamicPayloads();                         // Enable Dynamic Payloads
46
47   nrfClearInterruptBits();                            // Clear interrupt bits
48   nrfFlushRx();                                       // Flush fifo's
49   nrfFlushTx();
50
51   nrfOpenWritingPipe(pipe);                           // Pipe for sending
52   nrfOpenReadingPipe(0, pipe);                        // Necessary for acknowledge
53 }
```

In code E.2 staat het hoofdprogramma van de zender. Regel 11 en 12 sluiten de headerbestanden van de bibliotheek in. Op regel 14 is het adres `"HvA01"` van de *pipe* gedeclareerd als een array van vijf `uint8_t`.

Code E.2 : Het hoofdprogramma voor de zender van figuur E.7.

```
 8 #define F_CPU 2000000UL
 9 #include <avr/io.h>
10 #include <util/delay.h>
11 #include "nrf24/nrf24spiXM2.h"
12 #include "nrf24/nrf24L01.h"
13
14 uint8_t  pipe[5] = {0x48, 0x76, 0x41, 0x30, 0x31};       // pipe address "HvA01"
15
16 void     init_nrf(void);
17 void     init_adc(void);
18 int16_t  read_adc(void);
19
20 int main(void)
21 {
22   int16_t result;
23
24   init_adc();
25   init_nrf();
26
27   while (1) {
28     result = read_adc();
29     nrfWrite( (uint8_t *) &result, sizeof(uint16_t) );   // little endian: low byte is sent first
30     _delay_ms(20);
31   }
32 }
```

De breedte van het geheugen is vaak kleiner dan de breedte van de getallen, die opgeslagen worden. Er zijn dan twee methoden om een getal dan op te slaan: bij de *big endian* wordt het meest significante deel en bij de *little endian* wordt het minst significante deel het eerst opgeslagen.

Het hoofdprogramma initialiseert de ADC en de nRF24L01+ en leest daarna iedere 20 ms de waarde van de ADC en verstuurt deze met de functie `nrfWrite`. De functie `nrfWrite` heeft twee parameters: een pointer naar een array van bytes van het type `uint8_t` en het aantal bytes dat verstuurd wordt. De variabele `result` is 16-bits en bestaat dus uit twee bytes. Deze bytes staan naast elkaar in het geheugen. Het is daardoor mogelijk aan de functie het adres van `result` mee te geven als pointer naar een array van `uint8_t`. Wel wordt dan het minst significante byte het eerste verstuurd, omdat AVR-gcc voor integers de bytevolgorde *little endian* gebruikt, waarbij het minst significante byte het eerst wordt opgeslagen.

De functie `init_adc()` komt overeen met die uit code 20.3 met nu alleen aansluiting 0 van poort A als ingang. De functie `read_adc()` is helemaal identiek met die uit code 20.3.

In code E.3 staat een initialisatiefunctie voor de nRF24L01+ van de ontvanger. Deze functie is tot regel 72 identiek met de initialisatiefunctie van de ontvanger. De ontvanger gebruikt een interrupt. De interruptpin is aansluiting 6 van poort F en reageert op de neergaande flank. Het interruptniveau is laag.
De initialisatiefunctie opent op regel 77 alleen een *pipe* voor het lezen en stelt daarna op regel 78 de radiomodule in op de RX-mode.

Code E.3 : De initialisatie van nRF24L01+ voor de ontvanger van figuur E.7.

```
55 void init_nrf(void)
56 {
57   nrfspiInit();
58   nrfBegin();
59
60   nrfSetRetries(NRF_SETUP_ARD_1000US_gc, NRF_SETUP_ARC_8RETRANSMIT_gc);
61   nrfSetPALevel(NRF_RF_SETUP_PWR_6DBM_gc);
62   nrfSetDataRate(NRF_RF_SETUP_RF_DR_250K_gc);
63   nrfSetCRCLength(NRF_CONFIG_CRC_16_gc);
64   nrfSetChannel(54);
65   nrfSetAutoAck(1);
66   nrfEnableDynamicPayloads();
67
68   nrfClearInterruptBits();
69   nrfFlushRx();
70   nrfFlushTx();
71
72   // Interrupt
73   PORTF.INT0MASK |= PIN6_bm;
74   PORTF.PIN6CTRL  = PORT_ISC_FALLING_gc;
75   PORTF.INTCTRL   = (PORTF.INTCTRL & ~PORT_INT0LVL_gm) | PORT_INT0LVL_LO_gc;
76
77   nrfOpenReadingPipe(0, pipe);
78   nrfStartListening();
79 }
```

In code E.4 staat het hoofdprogramma van de ontvanger. Regel 11 en 12 sluiten weer de headerbestanden van de bibliotheek in. Regel 14 definieert weer het *pipe address* en regel 15 declareert de array `packet` waarin de te ontvangen gegevens worden opgeslagen. Hoewel in dit voorbeeld maar twee bytes gebruikt worden, heeft deze array de maximale pakketgrootte van 32 bytes.

Code E.4 : Het hoofdprogramma voor de ontvanger van figuur E.7.

```
 8 #define F_CPU 2000000UL
 9 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include "nrf24/nrf24spiXM2.h"
12 #include "nrf24/nrf24L01.h"
13
14 uint8_t  pipe[5] = {0x48, 0x76, 0x41, 0x30, 0x31};
15 uint8_t  packet[32];
16
17 void init_pwm(void);
18 void init_nrf(void);
19
20 int main(void)
21 {
22   init_pwm();
23   init_nrf();
24
25   PMIC.CTRL |= PMIC_LOLVLEN_bm;
26   sei();
27
28   while (1) {}   // do nothing
29 }
```

Het hoofdprogramma initialiseert eerst het PWM-signaal, waarmee de led wordt aangestuurd, en initialiseert vervolgens de nRF24L01+. Nadat het interruptmechanisme is ingesteld en aangezet, komt het hoofdprogramma in een oneindige while, die verder niets doet.

Code E.5 : De initialisatie van het PWM-signaal met de timer/counter 0 van poort C.

```
44 void init_pwm(void)
45 {
46   PORTC.OUTCLR = PIN0_bm;
47   PORTC.DIRSET = PIN0_bm;
48
49   TCC0.CTRLB   = TC0_CCAEN_bm | TC_WGMODE_SINGLESLOPE_gc;
50   TCC0.CTRLA   = TC_CLKSEL_DIV1_gc;
51   TCC0.PER     = 9999;
52   TCC0.CCA     = 0;
53 }
```

De initialisatiefunctie voor het PWM-signaal staat in code E.5. De frequentie is ingesteld op 200 Hz. Dit volgt direct uit formule 22.3 met f_{clk} 2 MHz, een prescaling N van 1 en `PER` gelijk aan 9999. De duty-cycle is aanvankelijk 0%, omdat register `CCA` 0 is. Nadat de ontvanger geïnitialiseerd is en een waarde heeft ontvangen, zal de duty-cycle veranderen. De maximale waarde die de ontvanger verstuurt is 2047. Met formule 22.4 volgt daaruit dat de maximale duty-cycle ruim 20% is.

Op het moment dat de nRF24L01+ van de ontvanger een correct datapakket heeft ontvangen, stuurt het een interrupt naar de Xmega van de ontvanger. Deze interrupt triggert de interruptfunctie, die in code E.6 staat.

Code E.6 : De interruptfunctie voor de ontvanger van figuur E.7.

```
31  ISR(PORTF_INT0_vect)
32  {
33    uint8_t  tx_ds, max_rt, rx_dr;
34
35    nrfWhatHappened(&tx_ds, &max_rt, &rx_dr);
36
37    if ( rx_dr ) {
38      nrfRead(packet, 2);
39      TCC0.CCABUFL  =  packet[0];       // low byte was sent first
40      TCC0.CCABUFH  =  packet[1];
41    }
42  }
```

De nRF24L01+ kent drie redenen om een interrupt te genereren en de drie interruptbits uit het statusregister geven aan wat de reden is:

- Bit `TX_DS`, *transmit data send*, is hoog, wanneer de gegevens correct zijn verzonden.
- Bit `MAX_RT`, *maximum retries*, is hoog, wanneer de gegevens niet correct zijn verstuurd en het maximaal aantal pogingen overschreden is.
- Bit `RX_DR`, *receive data ready*, is hoog, wanneer de gegevens correct ontvangen zijn.

De functie `nrfWhatHappened()` vraagt de waarden van de drie interruptbits op. Als de variabele `rx_dr` ongelijk aan `0` is, heeft de nRF24L01+ een datapakket ontvangen. De functie `nrfRead()` haalt dit pakket dan op en plaatst het in de array `packet`. Met de twee gelezen bytes wordt de duty-cycle van het PWM-signaal aangepast door deze aan het `CCA`-register van de timer/counter toe te kennen. Het lage byte van de analoge waarde is het eerste byte dat de zender verstuurt, dus wordt `packet[0]` aan register `CCABUFL` toegekend en `packet[1]` aan register `CCABUFH`.

F

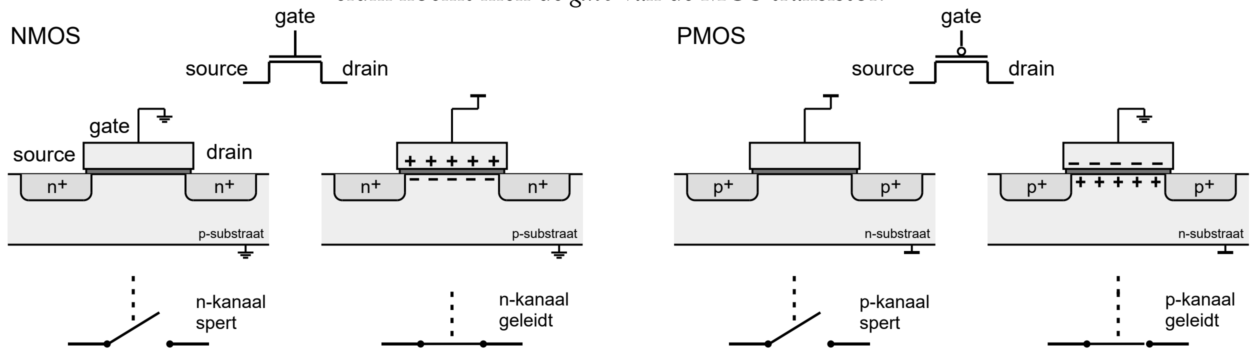
CMOS

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.

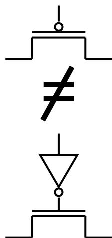
F.1 De MOS-transistor als schakelaar

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.

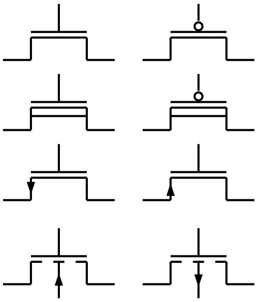


Figuur F.1 : Een NMOS- en een PMOS-transistor.
Links is het symbool en twee keer de dwarsdoorsnede van een NMOS-transistor getekend. Tussen de gate en het p-substraat bevindt zich de isolerende oxidelaag. Direct naast de gate bevinden zich de zwaar gedoteerde n-gebieden: de drain en de source. Bij de 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 F.2 : Let op! Een PMOS-transistor is geen inverter met een NMOS-transistor.

Figuur F.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.

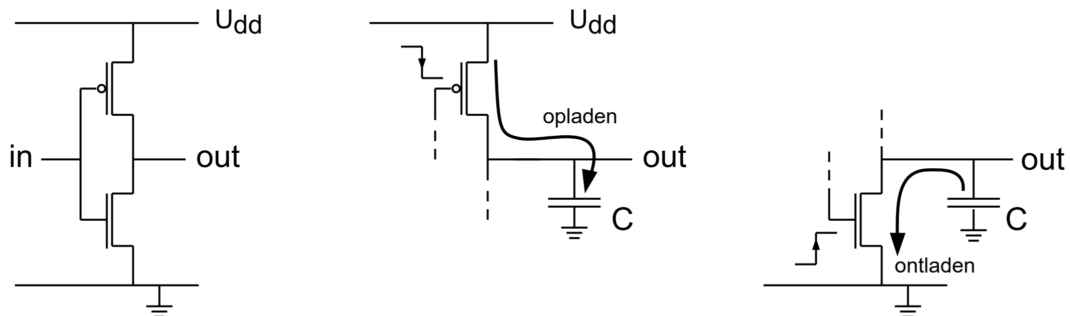


Figuur F.3 : Alternatieve symbolen voor MOS-transistoren. Links staat steeds een NMOS- en rechts het bijbehorende PMOS-symbool. De onderste symbolen hebben het substraat of back-gate als vierde aansluiting.

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.

F.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 F.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 F.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.

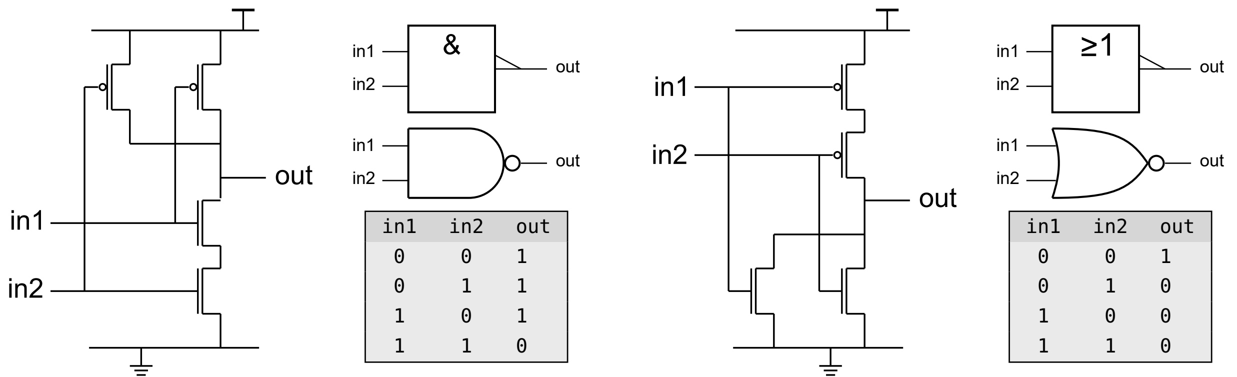
Figuur F.5 : Het Amerikaanse en Europese symbool voor een inverter. Links staat het Amerikaanse of ANSI-symbool en rechts staat het Europese of IEC-symbool.

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.

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.

F.3 CMOS-logica

In figuur F.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 de referentie 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 de referentie 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 F.6 geeft voor de beide poorten ook de Amerikaanse en de Europese symbolen.



in1	in2	out
0	0	1
0	1	1
1	0	1
1	1	0

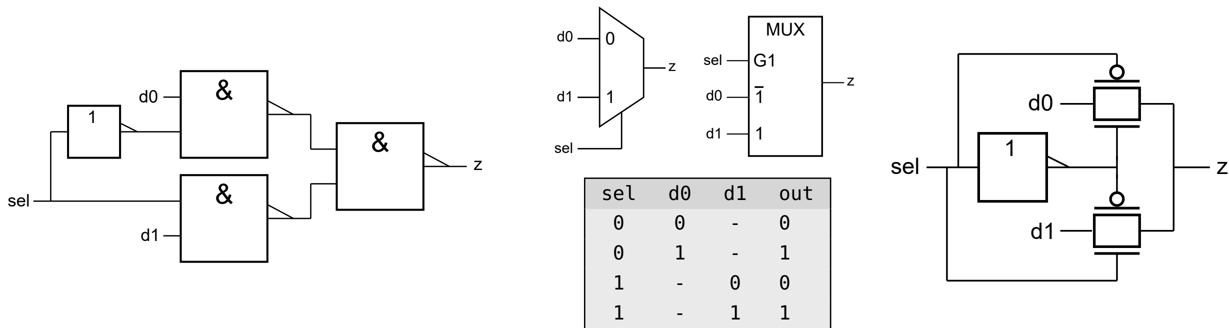
in1	in2	out
0	0	1
0	1	0
1	0	0
1	1	0

Figuur F.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 F.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 F.7 staat ook een multiplexer, die opgebouwd is uit een inverter en twee transmissiepoorten. De transmissiepoort wordt in paragraaf F.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 F.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.

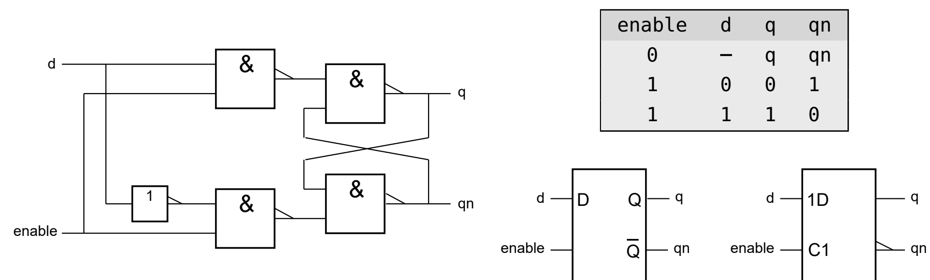
F.4 De D-latch

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 F.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 F.8.

In dit boek is een D-latch altijd niveaugevoelig (*level sensitive*) en is een D-flipflop altijd flankgevoelig (*edge triggered*).

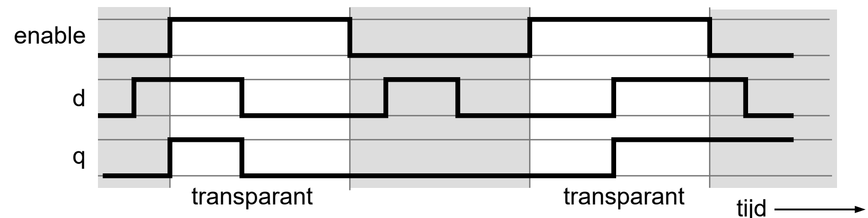


enable	d	q	qn
0	–	q	qn
1	0	0	1
1	1	1	0

Figuur F.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.

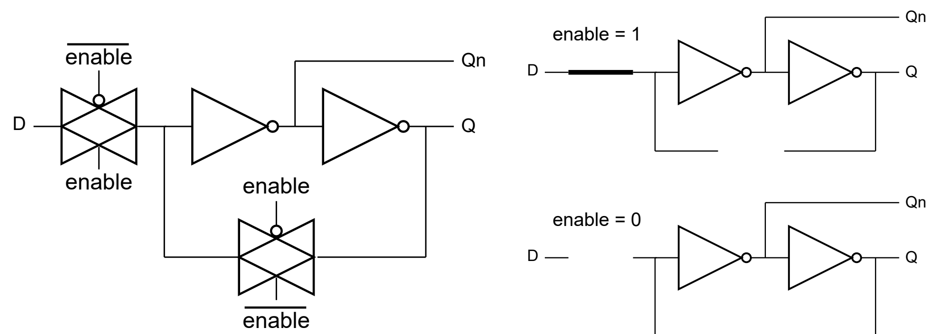
In figuur F.9 staat het tijdsdiagram van de D-latch. Duidelijk is te zien dat als het signaal `enable` hoog is — en dus actief is — uitgang `q` de data-ingang `d` volgt. De latch is dan transparant. Daarom noemt men deze component ook wel eens een transparante D-latch. Als `enable` laag is, verandert 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.

De generiek IO van de Xmega, zie figuur 15.3, heeft een latch voor de ingangsflipflop `INn`. Deze latch zorgt er dat het ingangssignaal gesynchroniseerd wordt.



Figuur F.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.

Figuur F.10 toont een D-latch die uit twee inverters en twee transmissiepoorten bestaat. De transmissiepoort komt in paragraaf F.7 aan de orde. Voor deze D-latch zijn acht transistoren nodig in plaats van de achttien die bij het schema van figuur F.8 nodig zijn. Een nadeel is dat het enable-signaal geïnverteerd beschikbaar moet zijn.



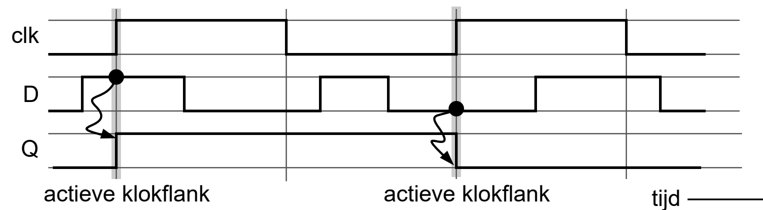
Figuur F.10 : Een D-latch opgebouwd uit transmissiepoorten.
Links staat het schema van een latch die is opgebouwd uit twee inverters en twee transmissiepoorten. Als `enable` hoog is, geleidt de eerste transmissiepoort en wordt het data-signaal doorgegeven naar de uitgangen. Als `enable` laag is, geleidt 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 F.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.

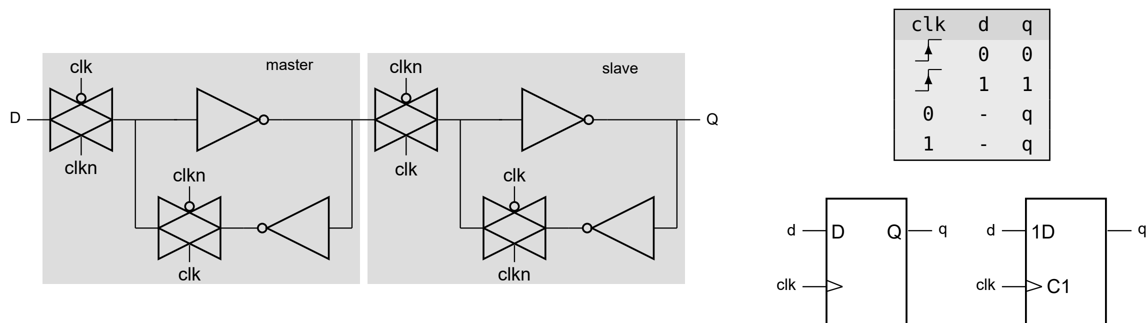
F.5 De D-flipflop

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 F.12 geeft de waarheidstabel en de symbolen van een positieve flankgevoelige D-flipflop. De waarheidstabel en het tijdsdiagram uit figuur F.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 F.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 F.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`.



clk	d	q
↑	0	0
↑	1	1
0	-	q
1	-	q

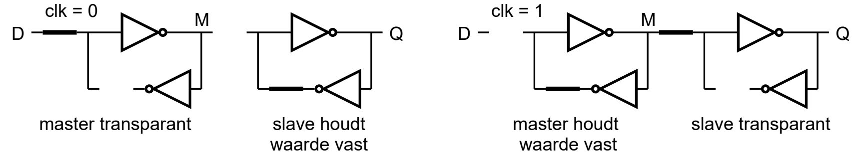
Figuur F.12 : Een D-flipflop op basis van transmissiepoorten.
Links staat het schema van een D-flipflop die is opgebouwd uit transmissiepoorten. In feite bestaat de flipflop uit twee in serie geschakelde D-latches: een *master* met een geïnverteerde klok (`clkn`) en een *slave* met een klok `clk`. Rechts bovenaan staat de waarheidstabel en daar onder staan het Amerikaanse en het IEC-symbool.

Figuur F.13 demonstreert het gedrag van de D-flipflop uit figuur F.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.

Sommige schrijvers noemen de D-latch en de D-flipflop respectievelijk D-flipflop en master-slave flipflop.

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`.

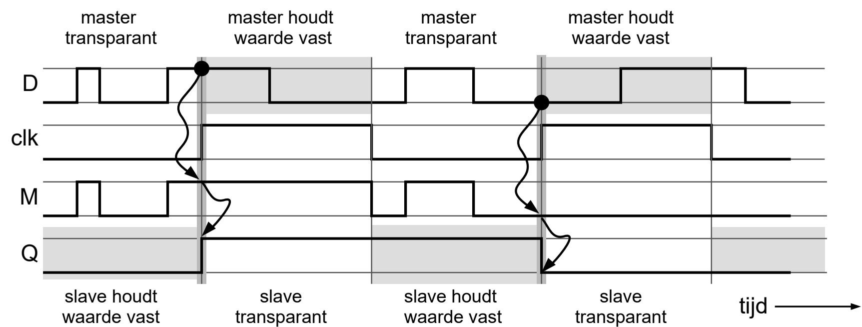
Vanwege de bouw spreekt men over een master-slave flipflop en vanwege het gedrag over een edge-triggered flipflop.



Figuur F.13 : Het gedrag van een D-flipflop uit figuur F.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 F.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 F.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_{setup} is de tijd dat het datasignaal stabiel moet zijn voor de klokflank. De holdtijd t_{hold} is de tijd dat het datasignaal stabiel moet blijven na de klokflank. In figuur F.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.

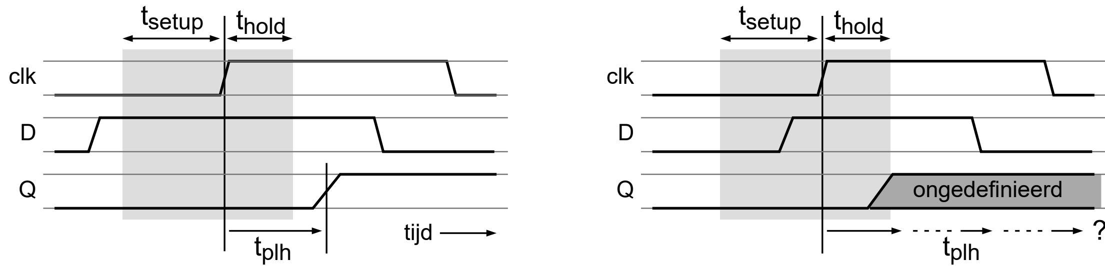
Een MTBF die heel kort is, ontdekt men — als het goed is — bij het testen.

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_p. Deze kan anders zijn bij het hoog worden (t_{plh}) en bij het laag worden (t_{phl}).

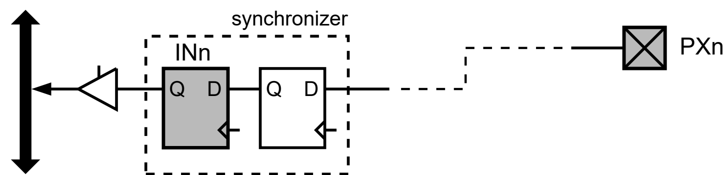
De kans dat het misgaat is te berekenen en hangt af van de bouw van de flipflop, van de klokfrequentie en van de frequentie waarmee het ingangssignaal verandert. Uit deze kans kan de tijd worden berekend tussen de twee momenten dat het misgaat. Deze tijd noemt men de MTBF (*mean time between failure*).



Figuur F.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.

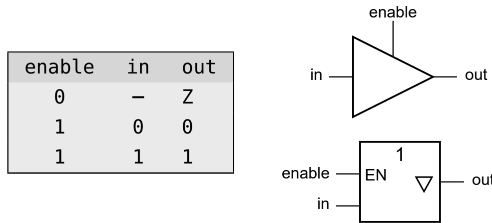
De generiek IO van de Xmega, zie figuur 15.3, gebruikt een extra D-latch voor de synchronisatie. Deze latch, zie figuur F.16, staat voor flipflop `INn`.
Als de klok laag is, is de ingang van de D-latch niet doorverbonden. De latch houdt de huidige waarde vast en de uitgang van de D-latch is stabiel. Als de klok hoog wordt, leest de D-flipflop deze stabiele waarde. De flipflop wordt zo nooit metastabiel.



Figuur F.16 : De synchronizer bij de generieke IO van de Xmega.
Tussen de aansluitpin `PXn` en de D-flipflop `PINXn` is de extra D-latch aangebracht om het ingangssignaal te synchroniseren.

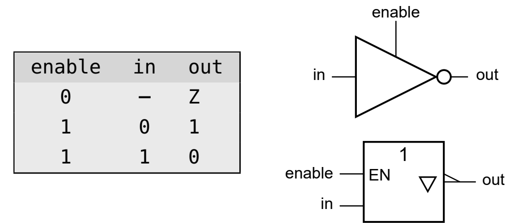
F.6 De tristatebuffer en de tristate-inverter

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 F.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 F.18 staan de symbolen en de waarheidstabel van de tristate-inverter.



enable	in	out
0	–	Z
1	0	0
1	1	1

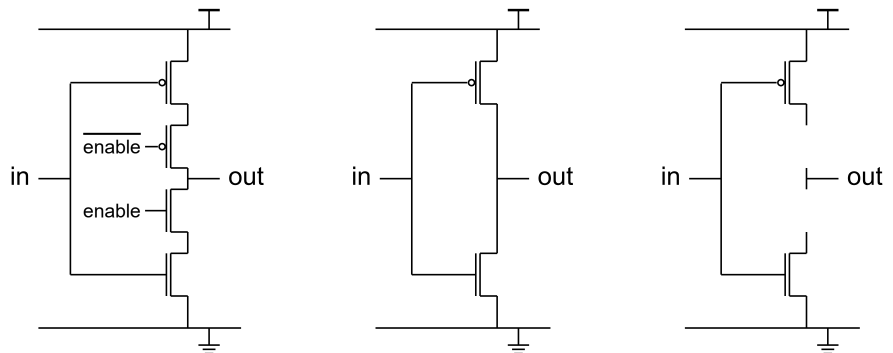
Figuur F.17 : De waarheidstabel en de symbolen voor de tristate buffer.
Links staat de waarheidstabel, rechtsboven het Amerikaanse en rechtsonder het IEC-symbool.



enable	in	out
0	–	Z
1	0	1
1	1	0

Figuur F.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 F.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.

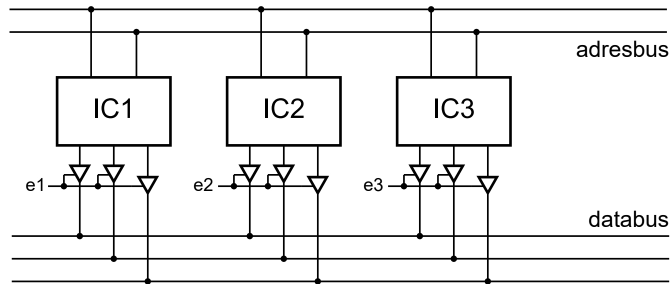


Figuur F.19 : De werking van een tristate-inverter. Links is de tristate-inverter getekend, die bestaat uit twee PMOS- en twee NMOS-transistoren.
In het midden staat de situatie als het enable-signaal hoog is. De binnenste twee transistoren zijn geleidend. De tristate-inverter functioneert dan als een 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 F.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 Xmega uit figuur 15.3 bevat bijvoorbeeld vijf tristatebuffers.

Tristatebuffers worden veel gebruikt bij busstructuren. De IC's in figuur F.20 kunnen alle drie gelijktijdig 'luisteren' naar wat er — bijvoorbeeld door de adresdecoder van de microprocessor — op de adresbus is gezet. Daarentegen kunnen de

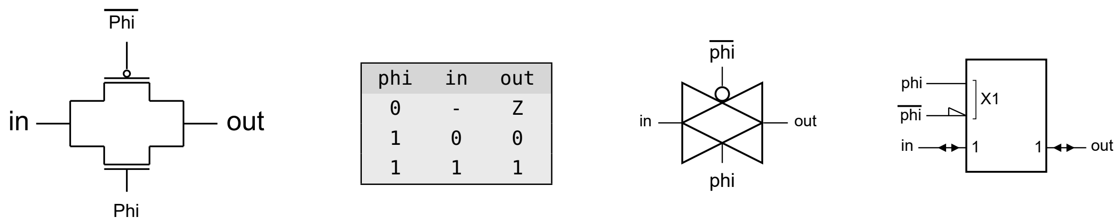


Figuur F.20 : De toepassing van een tristatebuffer. Alle drie IC's luisteren gelijktijdig naar de adresbus. De IC's zetten alleen informatie op de databus als het betreffende enable-signaal hoog is.

drie IC's niet allemaal tegelijkertijd gegevens op de databus zetten. Met de 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.

F.7 De transmissiepoort

Een transmissiepoort (*transmission gate*) bestaat uit een NMOS- en een PMOS-transistor. Figuur F.21 geeft de schakeling, de waarheidstabel en twee symbolen.



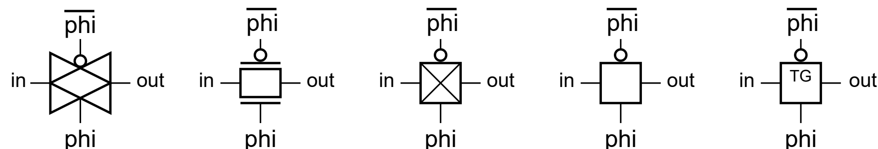
phi	in	out
0	-	Z
1	0	0
1	1	1

Figuur F.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.

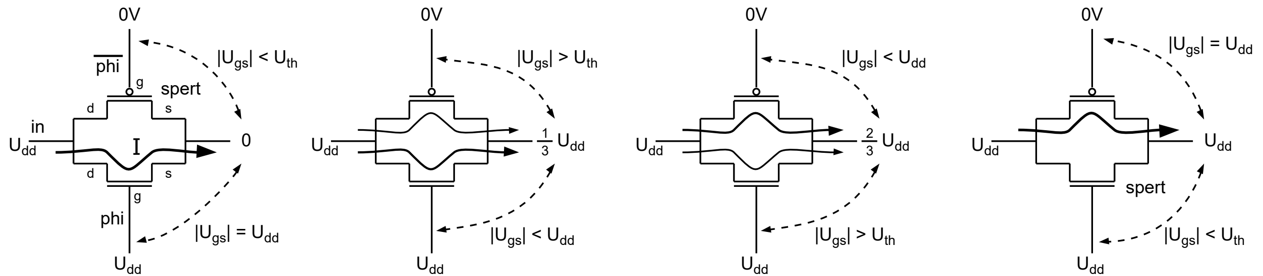
De transmissiepoort heeft geen richting. Beide zijde kunnen ingang en uitgang zijn. In de voorbeelden is de linker kant steeds hoog en loopt de stroom van links naar rechts. Als de linkerzijde laag is en de rechterzijde loopt de stroom van rechts naar links. Het IEC-symbool geeft dit bidirectionele gedrag duidelijk aan.

Er worden heel veel verschillende symbolen voor een transmissiepoort gebruikt. In figuur F.22 staan er een aantal.



Figuur F.22 : Een aantal voorbeelden van symbolen die voor een transmissiepoort worden gebruikt.

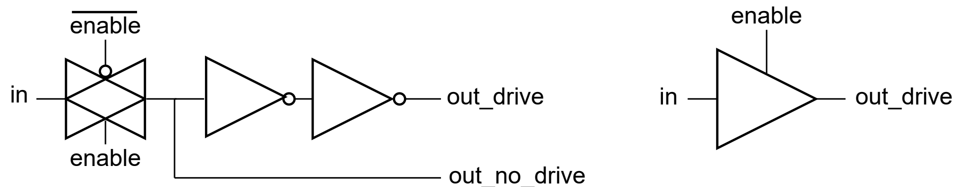
Figuur F.23 demonstreert de situatie dat de `phi` van laag naar hoog gaat terwijl de ingang hoog is. Aanvankelijk is de spanning U_{gs} tussen de gate en de source van de NMOS-transistor gelijk aan de voedingsspanning U_{dd}. De stroom door de NMOS is dan maximaal. De grootte van de U_{gs} van de PMOS 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 F.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}).

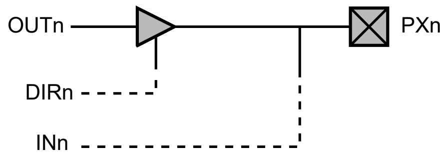
Met transmissiepoorten kunnen allerlei logische functies gerealiseerd worden. In figuur F.7 staat een multiplexer die opgebouwd is uit twee transmissiepoorten en een inverter. Figuur F.10 en figuur F.12 geven respectievelijke een D-latch en een D-flipflop met transmissiepoorten.
De kracht van componenten 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.

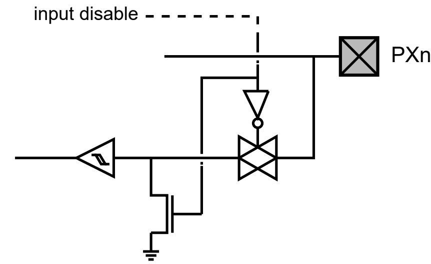


Figuur F.24 : De transmissiepoort met twee inverters en een tristatebuffer. De uitgang `out_no_drive` van de transmissiepoort heeft geen drijvende kracht. De uitgang van de tristatebuffer heeft deze kracht wel.

De twee inverters in figuur F.24 achter de transmissiepoort zorgen er voor dat de drijvende kracht (*drive*) van de uitgang `out_drive` wel goed is.
Figuur F.25 toont de tristatebuffer bij de aansluitpin van de generieke IO van de Xmega. Hier is een tristatebuffer nodig, want de stroom die geleverd moet wor-



Figuur F.25 : **De tristatebuffer bij de aansluitpin van de Xmega.**



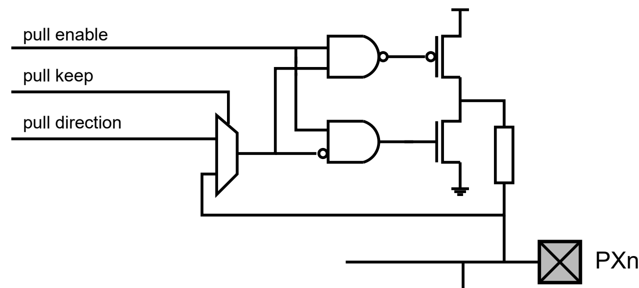
Figuur F.26 : **De transmissiepoort bij de generieke IO van de Xmega.**

den kan groot zijn. Figuur F.26 laat de ingangstak zien met de transmissiepoort, die aangesloten is op het signaal `input disable`. Achter de transmissiepoort zit een schmitttrigger, die de drijvende kracht levert. Daarom kan hier een transmissiepoort gebruikt worden.

F.8 De pulluptransistor en de pulldowntransistor

Een PMOS-transistor wordt gebruikt om een datalijn hoog te maken en een NMOS-transistor om een datalijn laag te maken. Een weerstand van enkele kΩ's maakt de drijvende kracht klein, zodat een andere component de spanning op de datalijn weer laag of hoog maken. De PMOS-transistor, die op deze manier wordt gebruikt, wordt een pulluptransistor genoemd en de NMOS-transistor, die de datalijn laag maakt, wordt een pulldowntransistor genoemd.

Figuur F.27 geeft een detail van de generieke IO bij de Xmega met de pullup- en de pulldowntransistor bij de aansluitpin. Als de signalen *pull enable*, *pull keep* en *pull direction* hoog zijn, zijn de gates van de pullup- en de pulldown-transistor laag, zodat de pulluptransistor geleidt en de pulldowntransistor spert. De aansluitpin is dan hoog, mits er geen andere signaal is die de pin omlaag trekt.



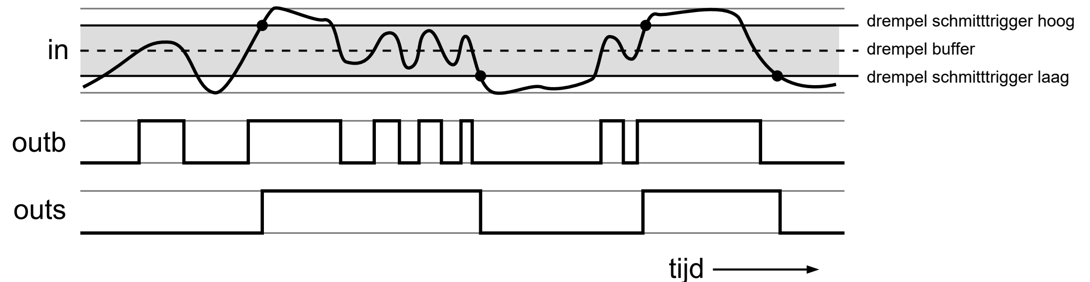
Figuur F.27 : **De pullup- en de pulldowntransistor bij de aansluitpin van de Xmega.**

Als het signaal *pull direction* laag is, zijn de gates van de pullup- en de pulldowntransistor hoog, zodat de pulldown geleidt en de pullup spert. De aansluitpin is dan laag, mits er geen andere signaal is die de pin omhoog trekt.

Het detail van de ingang van de Xmega uit figuur F.26 bevat ook een pulldowntransistor, die het signaal voor de schmitttrigger laag kan maken.

F.9 De schmitttrigger

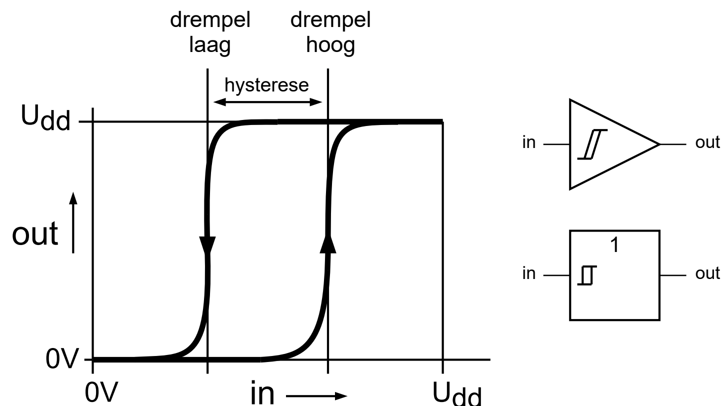
De schmitttrigger is een buffer of inverter met twee drempelwaarden. Als de ingang boven een bepaalde hoge drempelwaarde komt, wordt de uitgang hoog. Als de ingang beneden een andere lagere drempelwaarde komt, wordt de uitgang laag. Tussen de twee drempelwaarden behoudt de uitgang zijn waarde.



Figuur F.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.

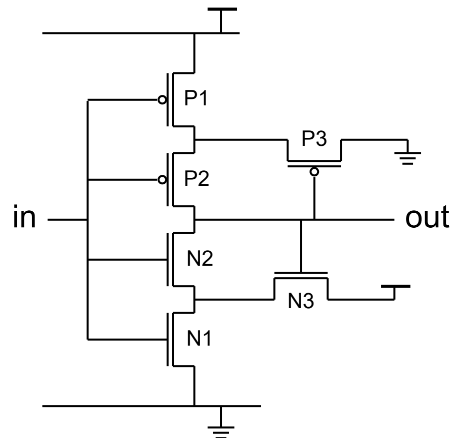
Het voordeel van de schmitttrigger ten opzichte van een gewone buffer of inverter is de grote stabiliteit en de lage ruisgevoeligheid. Een ingangssignaal met ruis veroorzaakt bij een gewone buffer meerdere overgangen. Bij een schmitttrigger is dat veel minder vaak het geval, zoals het tijdsdiagram van figuur F.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 F.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 F.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 F.30 staat een CMOS-implementatie.



Figuur F.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.

G

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 G.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 G.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 G.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`	bepaalt absolute waarde van een **`int`**
`div`	geheeltallige deling met **`int`**'s
`labs`	bepaalt 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 G.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 G.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 G.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 G.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 G.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 G.8 : De belangrijkste macro's uit floats.h.

Macro's	
`FLT_RADIX`	2, de basis radix voor alle 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 G.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 G.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 G.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 G.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 G.12 : De belangrijkste functies uit assert.h.

Functies	
`assert`	detecteert runtimefouten en geeft dan een foutmelding

Tabel G.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

H

Application notes

Naast de AU-manual en de datasheet staan op de internetsite van Atmel een groot aantal application notes. Meestal hoort daar een zip-bestand bij met drivers en voorbeelden. Tabel H.1 geeft een overzicht van de meest gebruikte documenten.

Tabel H.1 : Een overzicht van de meest gebruikte documenten en application notes.

Document	Titel
	Atmel AVR XMEGA AU Manual
	ATxmega256A3U/ATxmega192A3U/ATxmega128A3U/ATxmega64A3U Datasheet
	AVR Instruction Set
AVR1000	Getting Started Writing C-code for XMEGA
AVR1001	Getting Started With the XMEGA Event System
AVR1003	Using the XMEGA Clock System
AVR1005	Getting started with XMEGA
AVR1010	Minimizing the power consumption of XMEGA devices
AVR1300	Using the XMEGA ADC
AVR1301	Using the XMEGA DAC
AVR1302	Using the XMEGA Analog Comparator
AVR1303	Using the XMEGA IR communication module
AVR1304	Using the XMEGA DMA Controller
AVR1305	Using the XMEGA Interrupts and the Programmable Multi-level Interrupt Controller
AVR1306	Using the XMEGA Timer/Counter
AVR1307	Using the XMEGA USART
AVR1308	Using the XMEGA TWI
AVR1309	Using the XMEGA SPI
AVR1310	Using the XMEGA Watchdog Timer
AVR1311	Using the XMEGA Timer/Counter Extensions
AVR1313	Using the XMEGA IO Pins and External Interrupts
AVR1314	Using the XMEGA Real Time Counter
AVR1315	Using the XMEGA EEPROM
AVR1321	Using the Atmel AVR XMEGA 32-bit Real Time Counter and Battery Backup System
AVR1600	Using the XMEGA Quadrature Decoder
AVR1606	XMEGA Internal RC Oscillator Calibration
AVR1617	Frequency Measurement with Atmel AVR XMEGA Family Devices
AVR186	Best practices for the PCB layout of Oscillators

De AU-manual is het belangrijkste document, hierin worden alle onderdelen van Xmega's uit de AU-serie uitgebreid besproken. De datasheet bevat een korte bespreking van alle onderdelen, een samenvatting van de instructieset, de package

informatie, de elektrische en niet-elektrische eigenschappen en informatie over de pinaansluitingen. Een vereenvoudigde versie van het overzicht met de pinaansluitingen uit paragraaf 32.2 van de datasheet staat in tabel H.2.

Tabel H.2 : Een overzicht van de aansluitingen bij de Xmega256a3u.

PORTA	pin #	interrupt	ADCAPOS GAINPOS	ADCA NEG	ADCA GAINNEG	ACA POS	ACA NEG	ACA OUT	REFA
PA0	62	SYNC	ADC0	ADC0		AC0	AC0		AREF
PA1	63	SYNC	ADC1	ADC1		AC1	AC1		
PA2	64	SYNC/ASYNC	ADC2	ADC2		AC2			
PA3	1	SYNC	ADC3	ADC3		AC3	AC3		
PA4	2	SYNC	ADC4		ADC4	AC4			
PA5	3	SYNC	ADC5		ADC5	AC5	AC5		
PA6	4	SYNC	ADC6		ADC6	AC6		AC1OUT	
PA7	5	SYNC	ADC7		ADC7		AC7	AC0OUT	

PORTB	pin #	interrupt	ADCAPOS GAINPOS	ADCBPOS GAINPOS	ADCB NEG	ADCB GAINNEG	ACB POS	ACB NEG	ACBOUT	DAC	REFB	JTAG
PB0	6	SYNC	ADC8	ADC0	ADC0		AC0	AC0			AREF	
PB1	7	SYNC	ADC9	ADC1	ADC1		AC1	AC1				
PB2	8	SYNC/ASYNC	ADC10	ADC2	ADC2		AC2			DAC0		
PB3	9	SYNC	ADC11	ADC3	ADC3		AC3	AC3		DAC1		
PB4	10	SYNC	ADC12	ADC4		ADC4	AC4					TMS
PB5	11	SYNC	ADC13	ADC5		ADC5	AC5	AC5				TDI
PA6	12	SYNC	ADC14	ADC6		ADC6	AC6		AC1OUT			TCK
PB7	13	SYNC	ADC15	ADC7		ADC7		AC7	AC0OUT			TDO

PORTC	pin #	interrupt	TCC0	AWEXC	TCC1	USARTC0	USARTC1	SPIC	TWIC	TWIC*	CLOCKOUT	EVENTOUT
PC0	16	SYNC	OC0A	$\overline{\text{OC0ALS}}$					SDA	SDAIN		
PC1	17	SYNC	OC0B	OC0AHS		XCK0			SCL	SCLIN		
PC2	18	SYNC/ASYNC	OC0C	$\overline{\text{OC0BLS}}$		RXD0				SDAOUT		
PC3	19	SYNC	OC0D	OC0BHS		TXD0				SCLOUT		
PC4	20	SYNC		$\overline{\text{OC0CLS}}$	OC1A			$\overline{\text{SS}}$				
PC5	21	SYNC		OC0CHS	OC1B		XCK1	MOSI				
PC6	22	SYNC		$\overline{\text{OC0DLS}}$			RXD1	MISO			clk_{rtc}	
PC7	23	SYNC		OC0DHS			TXD1	SCK			clk_{per}	EVOUT

PORTD	pin #	interrupt	TCD0	TCD1	USB	USARTD0	USARTD1	SPID	CLOCKOUT	EVENTOUT
PD0	26	SYNC	OC0A							
PD1	27	SYNC	OC0B			XCK0				
PD2	28	SYNC/ASYNC	OC0C			RXD0				
PD3	29	SYNC	OC0D			TXD0				
PD4	30	SYNC		OC1A				$\overline{\text{SS}}$		
PD5	31	SYNC		OC1B			XCK1	MOSI		
PD6	32	SYNC			D-		RXD1	MISO		
PD7	33	SYNC			D+		TXD1	SCK	clk_{per}	EVOUT

PORTE	pin #	interrupt	TCE0	TCE1	USARTE0	USARTE1	SPIE	TWIE	TWIE*	TOSC	CLOCKOUT	EVENTOUT
PE0	36	SYNC	OC0A					SDA	SDAIN			
PE1	37	SYNC	OC0B		XCK0			SCL	SCLIN			
PE2	38	SYNC/ASYNC	OC0C		RXD0				SDAOUT			
PE3	39	SYNC	OC0D		TXD0				SCLOUT			
PE4	40	SYNC		OC1A			$\overline{\text{SS}}$					
PE5	41	SYNC		OC1B		XCK1	MOSI					
PE6	42	SYNC				RXD1	MISO			TOSC2		
PE7	43	SYNC				TXD1	SCK			TOSC1	clk_{per}	EVOUT

PORTF	pin #	interrupt	TCF0	USARTF0
PF0	46	SYNC	OC0A	
PF1	47	SYNC	OC0B	XCK0
PF2	48	SYNC/ASYNC	OC0C	RXD0
PF3	49	SYNC	OC0D	TXD0
PF4	50	SYNC		
PF5	51	SYNC		
PF6	54	SYNC		
PF7	55	SYNC		

PORTR	pin #	interrupt	PDI	XTAL
PDI	56		PDI_DATA	
$\overline{\text{reset}}$	57		PDI_CLOCK	
PR0	58	SYNC		XTAL2
PR1	59	SYNC		XTAL1

Bij TC0 kunnen alle pinnen worden verplaatst naar het hoge nibble.

Als TC0 is geconfigureerd als TC2 zijn alle acht pinnen beschikbaar als PWM-uitgang.

TWIC* is een alternatieve configuratie voor het gebruik bij externe drivers.

Bij USART0 kunnen alle pinnen worden verplaatst naar het hoge nibble.

Bij de SPI kunnen de MOSI en SCK worden verwisseld.

clk_{per} en EVENTOUT kunnen worden verplaatst tussen de poorten C, D en E en tussen de pinnen 4 en 7.

I

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 I.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.

Tabel I.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*.

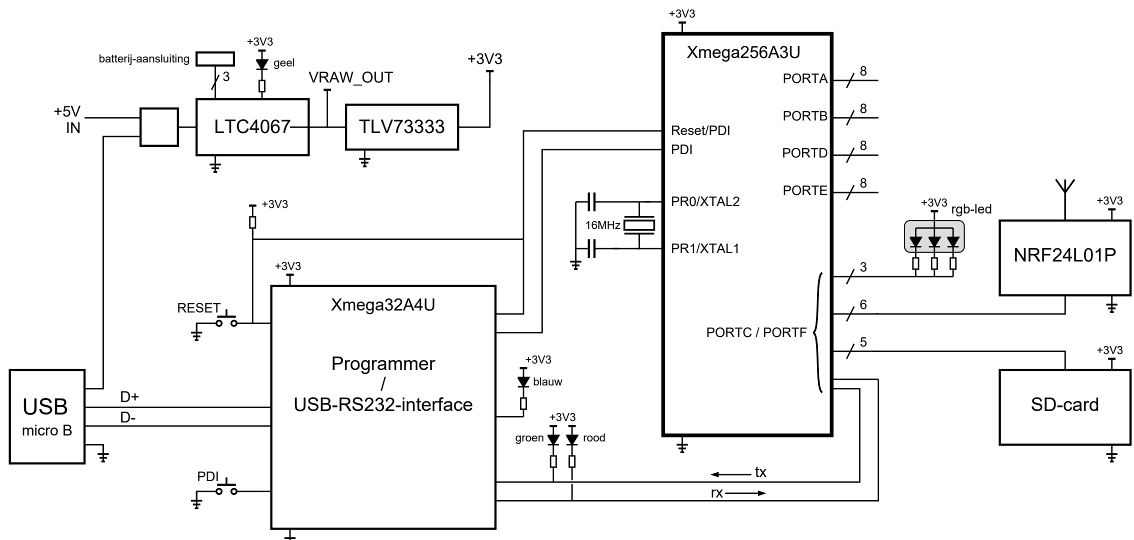
Dec	Oct	Hex	Bin	Symbool
0	000	00	0000 0000	NUL, Null, ’\0’
1	001	01	0000 0001	SOH
2	002	02	0000 0010	STX
3	003	03	0000 0011	ETX
4	004	04	0000 0100	EOT
5	005	05	0000 0101	ENQ
6	006	06	0000 0110	ACK
7	007	07	0000 0111	BEL, Bell, ’\a’
8	010	08	0000 1000	BS, Backspace, ’\b’
9	011	09	0000 1001	HT, Horizontal 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	?
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

Xmega-bord

Bij het samenstellen van dit boek is een speciaal Xmega-bord gebruikt, dat ontworpen is door J.D. Bakker van de Hogeschool van Amsterdam. In figuur J.1 staat het blokschema. Het Xmega-bord bevat onder meer:

- De Xmega256a3u, die geprogrammeerd kan worden.
- Een Xmega32a4u, die gebruikt wordt als programmer of als USB-RS232-interface.
- Een USB-powermanagementsysteem, LTC4067, die de USB-spanning, een externe spanning of een batterij gebruikt en de batterij oplaadt.
- Een lineaire spanningsregulator, TLV73333, voor een stabiele en nauwkeurige 3,3 V spanning.
- Een 2,4 GHz draadloze module op basis van een Nordic nRF24L01+.
- Een SD-kaarthouder.

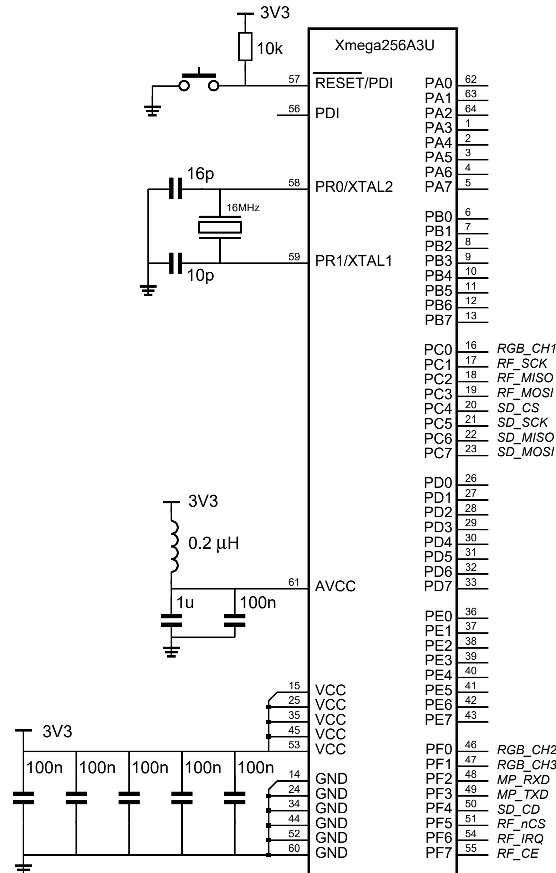


Figuur J.1 : Een blokschema van het Xmega-bord met de belangrijkste componenten.

Er zijn vier indicatieleds. De gele led brandt continu bij de aanwezigheid van een batterij en knippert als het Xmega-bord gevoed wordt via USB. De blauwe led brandt als de Xmega32a4u in de programmeermodus is en is uit als deze in communicatiemodus is. De groene led geeft aan dat er activiteit is op de `tx` van de Xmega256a3u en de rode led is dat er activiteit is op de `rx`.

Er zijn twee drukknoppen. De knop `RESET` herstart de beide Xmega's. Bij een reset met de `PDI`-knop ingedrukt komt de Xmega32a4u in de programmeermodus.

De twee aansluitingen van poort R van de Xmega256a3u zijn verbonden met een 16 MHz kristal. Verder zijn er zes poorten met acht aansluitingen. De poorten A, B, D en E zijn vrij te gebruiken. De zestien aansluitingen van de poorten C en F zijn verbonden met respectievelijk: de nRF24L01+, de SD-kaarthouder, de rgb-led en de `tx` en de `rx` van UART-verbinding tussen de twee Xmega's.



In dit boek is het fragment uit figuur J.2 in alle schema's met de Xmega256a3u gebruikt. Van ongebruikte poorten zijn dan niet alle aansluitingen getekend.

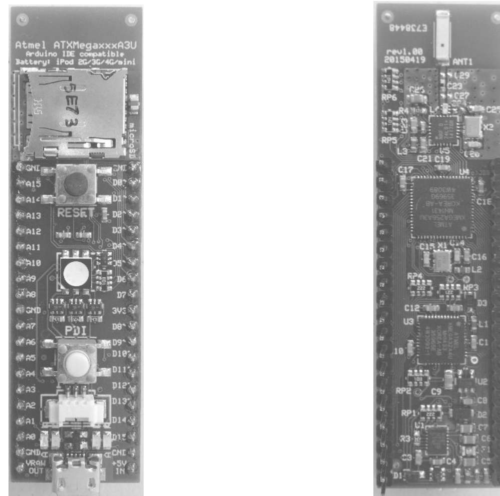
Figuur J.2 : Een fragment met de Xmega256a3u uit het schema van het Xmega-bord.

In figuur J.2 staat een fragment uit het schema met de Xmega256a3u. Alle aansluitingen van de Xmega zijn getekend. Bij de aansluitingen van poort C en F is aangegeven met welk onderdeel deze verbonden zijn op het Xmega-bord. Tabel J.3 geeft een compleet overzicht.

Het Xmega-bord wordt gevoed via USB, een externe voeding via `+5V_IN` of met een externe batterij. De batterij laadt automatisch op als ook de USB of de externe voedingsbron is aangesloten. Aansluiting `VRAW_OUT` van het Xmega-bord geeft de ruwe voedingsspanning. De spanningsregulator zet de voedingsspanning om naar een nauwkeurige spanning van +3,3 V. Bij aansluiting `+3V3` van het Xmega-bord is deze spanning beschikbaar. De maximale stroom, die door de TLV73333 geleverd kan worden, is slechts 360 mA. De vijf aansluitingen `GND` zijn verbonden met de referentie (*ground*).

Figuur J.3 : De aansluitingen van poort C en F van de Xmega op het Xmega-bord.

Signaalnaam	Aansluiting	Omschrijving
`MP_RXD`	`PF2`	de `rx` van de UART-verbinding met de Xmega32a4u
`MP_TXD`	`PF3`	de `tx` van de UART-verbinding met de Xmega32a4u
`RGB_CH1`	`PC0`	de blauwe rgb-led
`RGB_CH2`	`PF0`	de groene rgb-led
`RGB_CH3`	`PF1`	de rode rgb-led
`RF_SCK`	`PC1`	het kloksignaal van de SPI-verbinding met de nRF24L01+
`RF_MISO`	`PC2`	de MISO van de SPI, data van de nRF24L01+ naar de Xmega
`RF_MOSI`	`PC3`	de MOSI van de SPI, data van de Xmega naar de nRF24L01+
`RF_nCS`	`PF5`	het chipselectsignaal van de nRF24L01+
`RF_IRQ`	`PF6`	het interruptsignaal van de nRF24L01+
`RF_CE`	`PF7`	het chipenablesignaal van de nRF24L01+, dat het zenden/ontvangen activeert
`SD_CS`	`PC4`	het chipselectsignaal van de SD-kaart
`SD_SCK`	`PC5`	het kloksignaal van de SPI-verbinding met SD-kaart
`SD_MISO`	`PC6`	de MISO van de SPI, data van de SD-kaart naar de Xmega
`SD_MOSI`	`PC7`	de MOSI van de SPI, data van de Xmega naar de SD-kaart
`SD_CD`	`PF4`	het kaartdetectie-signaal van de SD-kaart



Figuur J.4 : Het bovenaanzicht en het onderaanzicht van het Xmega-bord.

Van het Xmega-bord bestaat ook een eenvoudige versie zonder de SD-kaarthouder en de draadloze module.
De *pinout* van dit bord is identiek met het bord van de volledige versie uit figuur J.5.

In figuur J.4 staat het bovenaanzicht en het onderaanzicht van het Xmega-bord. Het bord is ongeveer 20 bij 75 mm en heeft twee rijen met twintig pinnen. De pinout komt overeen met die van een 40-pins DIP-behuizing. Het Xmega-bord past in een *breadboard*. De afstand tussen de pinnen is 2,54 mm (100 mil) en de afstand tussen de rijen is 17,8 mm (700 mil).

Hoewel de afmetingen anders zijn, zijn de aansluitingen van het Xmega-bord compatibel met de Arduino. De naamgeving op het Xmega-bord past bij Arduino. Er zijn 16 analoge in- en uitgangen genaamd A0 tot en met A15 en 16 digitale in- en uitgangen genaamd D0 tot en met D15. Figuur J.5 geeft de gebruikelijke naamgeving van de aansluitingen.

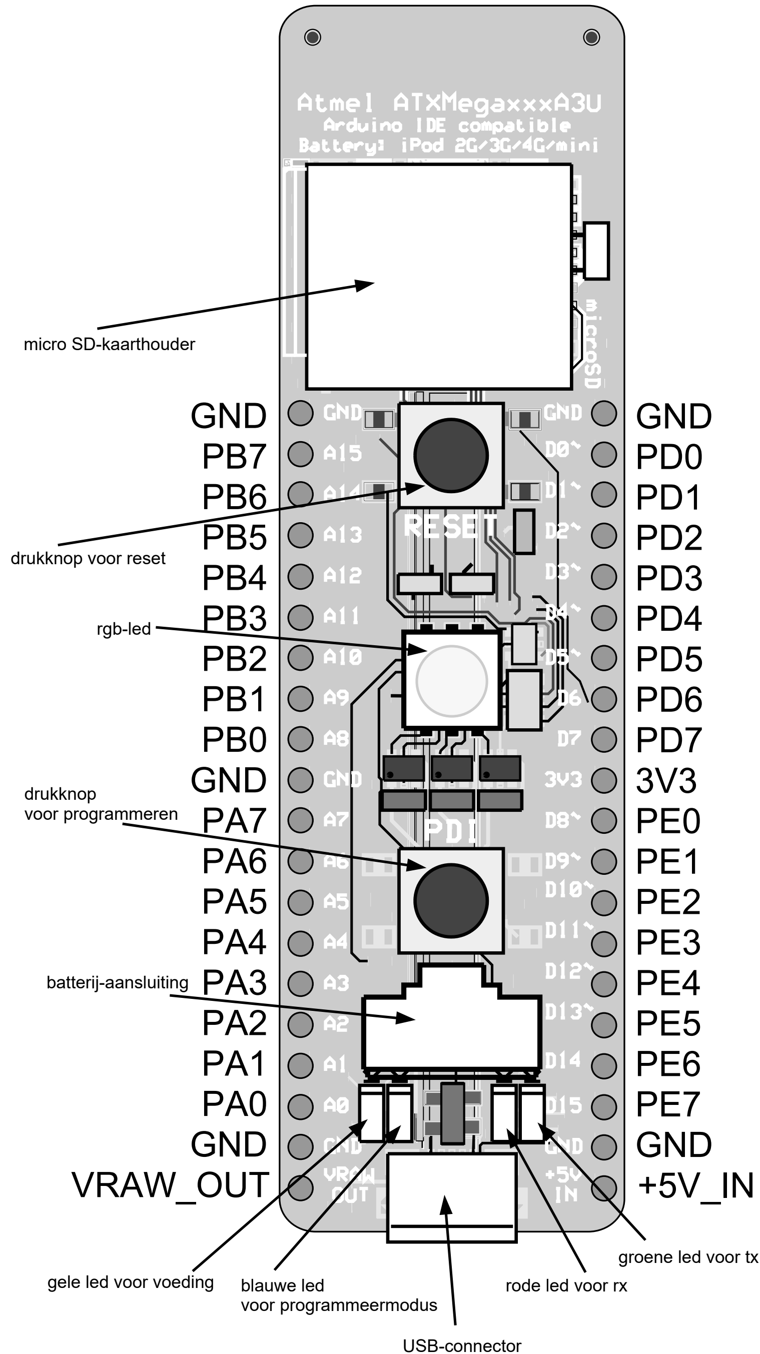
Naast de 32 aansluitingen voor de poorten A, B, D en E heeft het Xmega-bord nog acht aansluitingen.

Het bord heeft vijf `GND`-pinnen.

De `3V3` is een nauwkeurige 3,3 V afkomstig van de spanningsregulator, waarmee de Xmega's gevoed worden.

De `+5V_IN` is de aansluiting voor een externe voedingsspanning.

`VRAW_OUT` is de ruwe voedingsspanning afkomstig van de USB, de batterij of de `+5V_IN`-pin.



Figuur J.5 : Een realistische tekening van het Xmega-bord met de pinnamen van de Xmega256a3u.

Index

Symbolen

!, 108
!=, 26, 103, 107
", 13, 37, 151
" versus ´, 152
#, flag, 51, 105, 288, *zie ook* format specifier
#, operator, 187
##, operator, 187
%, format specifier, *zie* format specifier
%, modulus, 24, 28, 261, *zie ook* rekenkundige bewerking
%=, 110
&, adres, 41, 54, 131, 132, 315, 316, 318, 396
&, bitsgewijze EN, 41, 108, 113, 241
&&, 108
&=, 110
´, 151
´ versus ", 152
*, dereferentie pointer, 41, 133–135, 137, 138, 208
*, plaatsvervanger in format string, 51
*, typedeclaratie pointer, 41, 133
*, vermenigvuldigen, 24, 41
*/, einde commentaarblok, 90
*=, 110, 170
+, flag, 51
+, optellen, 20, 24
++, 93, 109
+=, 84, 110
ff_conf.h, 476
,, *zie* operator, komma-
-, aftrekken, 24
-, flag, 51
--, 109, 170
-=, 110
->, veld bij pointer naar structuur, 182, 183, 220
., veld bij structuur, 177, 183
..., 185
/, delen, 24
/*, start commentaarblok, 90
//, commentaarregel, 90
/=, 110
:, *zie* ?:
;, 13, 35
<, 26, 107
<<, 109, 113
<<=, 110, 113, 254
<=, 26, 107
=, 20, 47, 63
==, 25, 26, 103, 107
== versus =, 26
>, 26, 107
>=, 26, 107
>>, 109
>>=, 110
?:, 80, 87, 106
[], 122, 152, 157, 254, 256, 258
[][], 149
_BV(), *zie* bitbewerking
\, 50, 79, 106, 360
\", 106
\', 106
\0, 63, 76, 93, 106, 138, 151, 152, 155, 163, *zie ook* end-of-string
\\, 106
\f, 106
\n, 21, 106, 162, 163, 165, *zie ook* end-of-line
\r, 106
\t, 106
^, bitsgewijze XOR, 108
^=, 110
|, bitsgewijze OF, 108, 241
|=, 110
||, 108
_BV, bitvalue, 208
__, 188
__CYGWIN__, 465
__DATE__, 188
__FILE__, 188
__GNUC__, 188
__LINE__, 188
__STDC__, 188
__TIME__, 188
__USE_MINGW_ANSI_STDIO, 52
__flash, 258, 259, 445, 446
__progmem__, 259, 445
_bp, bitmasker, 210, 239, 240
_bp, bitpositie, 210, 239
_gc, groepsconfiguratie, 239, 240
_gm, groepsmasker, 239, 240
_gp, groepspositie, 239, 242
~, bitsgewijs inverteren, 108, 209, 241, 254
0, flag, 51
0, prefix octaal, 104
0X, prefix hexadecimaal, 104
0x, prefix hexadecimaal, 104
5x7 dotmatrix, *zie* dotmatrix
7-segmentdisplay, 243, 260–263
74HC595, schuifregister, 355
74hc595, 359

A

aansturing DC-motoren, 397–401
aansturing servomotor, 401–403
abs(), *zie* stdlib-bibliotheek
AC-motor, 401
achtergrondverlichting, 270
acos(), *zie* math-bibliotheek
actuator, 323
ADC, *zie* Analog-to-Digital Converter, *en ook* Xmega ADC
  conversietijd, 332
  opbouw pipelined, 327
  principe pipelined, 326–327
  successieve approximatie, 325–326
adres, 41, 54, 135, 137, 206, 208, *zie ook* geheugenadres
adresbus, 7, 497
adresoperator, 41, 54, 131, 132
Advanced Encryption Standard, 192
Advanced RISC Machine, 16
afdrukken
  conditioneel, 80, 87
  geformatteerd, 52, 286–290
afrondingsfout
  berekening baud rate, 294
afsluitteken, 30
alfanumerieke string
  omzetten in hexadecimaal getal, 314
  omzetten in integer, 60, 286
  omzetten in long, 286

omzetten in unsigned integer, 283, 286, 289
omzetten in unsigned long, 286, 289
algoritme, 41–43
antidender, 217, 237
initialisatie LCD, 283
quicksort, 171
successieve approximatie, 325
voor 7-segmentsdisplay, 261
voor dotmatrix, 253
voor het afspelen van beltonen, 405
voor sorteren, 171
ALU, *zie* Arithmetic Logic Unit
Analog-to-Digital Converter, 5, 192, 323–352, 357, 364, *zie* Xmega ADC *en ook* analoog-digitaalconversie
analoge comparator, 420–426, *zie ook* Xmega analoge comparator
blokschema, 420
principe, 417, 420
analoog-digitaalconversie, 324–326
comparator, 325, 332
conversietijd, 326, 332
DAC, 325
differentieel, 329
handmatig, single-ended, 337
referentiespanning, 325, 331–332
sample-and-hold, 332
signed single-ended, 329
unsigned single-ended, 329
analoog-digitaalconverter
conversiemethoden, 333
anode, 252, 270
ANSI, 12, 490
ANSI C, 12
GNU89, 14, 83, 90
GNU99, 83
ISO C90, 12–14
`appendStud()`, 183
application notes, 511–512
Application Specific Integrated Circuit, 3
architectuur
AVR-microcontroller, 7
Harvard-, 7, 189
microcontroller, 5
microprocessor, 4
Princeton-, 7
von Neumann-, 7
Xmega, 191
`argc`, *zie* hoofdroutine
argument, 25, 68, 77, 78, 157, 161, 177, 185
argumentenlijst, 51, 157, 176, 185
variabele, 185–186
`argv`, *zie* hoofdroutine
`argv[0]`, *zie* hoofdroutine
Arithmetic Logic Unit, 5
ARM, *zie* Advanced RISC Machine
array, 23–24, 28–30, 66, 119–131, 151
`[][]`, 149
`[]`, 122, 152, 157, 254, 256, 258
declaratie, 23, 122
dynamisch, 136
dynamische declaratie, 141–150
gebruik pointers bij, 138
index, 123, 125, 126, 129, 138
indices bij meerdimensionaal, 125, 126
initialisatie, 122
lezen buiten bereik van, 123–124
meer dimensionale, 124–130
schrijven buiten bereik van, 124
toewijzing, 23, 123
tweedimensionaal, 124, 126, 128, 144, 146–148, 253
tweedimensionaal array op basis van een eendimensionaal array, 146–147
tweedimensionaal met pointers, 146–147
tweedimensionaal met VLA, 148–150
van pointers, 157
van strings, 124, 157, 171
variable declaratie, 141–150
ASCII, 513
ASCII-tabel, 514
ASCII-waarde, 22, 87, 96, 124, 275, 314, 513
ASIC, *zie* Application Specific Integrated Circuit
`asin()`, *zie* math-bibliotheek
`asm`, 224–226, 264
assembler, 3
`asm`, 224–226, 264
`nop`, 224–226, 264, 274
assembly, 3, 228, 229
`assert.h`, *zie* standaardbibliotheek
associativiteit, *zie* voorrangsregels
asynchroon, 292, 463, 496
AT25128, serieel EEPROM, 357, 358
Atmel AVR, 17, 189
Atmel Studio, 9, 17, 198
`atoi()`, *zie* stdlib-bibliotheek
atomic block, 455–456
atomic-bibliotheek
`ATOMIC_FORCEON`, 456
`ATOMIC_RESTORESTATE`, 456
`ATOMIC_BLOCK()`, 456
`NONATOMIC_FORCEOFF`, 456
`NONATOMIC_RESTORESTATE`, 456
`NONATOMIC_BLOCK()`, 456
atomische bewerking, 455
attribuut, 259, 445
`auto`, 118
`register`, 118
`AutoCalibration2M()`, 299, 437
`AutoCalibration32M()`, 437
`AutoCalibrationTosc32M()`, 437
`average(()`, 185
AVR, *zie* Atmel AVR
AVR GNU C-Compiler, 9
AVR-bibliotheek
`avr/interrupt.h`, 224, 257
`avr/io.h`, 114, 205, 239
`avr/iox256a3u.h`, 179, 205, 226, 239
`avr/pgmspace.h`, 351, 413, 445
`avr/sleep.h`, 449
`avr/wdt.h`, 453
`util/atomic.h`, 455, 456
`wdt.h`, 452
AVR-gcc, 104, 198
`avr-gcc`, *zie* GNU C-Compiler voor AVR
`avr-libc` bibliotheek, 287, *zie ook* AVR-bibliotheek

B

bandgap-referentie, 331, 412, 420
basisweerstand, 250
baud, 293
baud rate, *zie* RS232
Baudot, Emile, 464
baudsnelheid, 301, 362
BCD, *zie* Binary Coded Decimal
beeldscherm, 2, 49, 199, 243, 244
behuizing, 193
Thin profile plastic Quad Flat Package, 193
Very thin Quad Flat No-lead, 193
beltoon, 404
berekenen faculteit met recursie, 169
bestand
einde van, 38, 161, 166–168
lezen uit en schrijven naar, 159–168
bestandsgrootte bepalen, 166
bestandssysteem, 474–477
besturingsopdracht, 67
bewerking, 95–110
logische, 5, *zie ook* logische bewerking
rekenkundige, 5, *zie ook* rekenkundige bewerking
relationele, *zie* relationele bewerking
Binary Coded Decimal, 370
binomium van Newton, 127
bipolaire transistor, 251
bit banging, 355, 476
bit clear, *zie* bitbewerking
bit set, *zie* bitbewerking
bit test, *zie* bitbewerking
bit toggle, *zie* bitbewerking
`bit_is_clear`, *zie* bitbewerking
`bit_is_set`, *zie* bitbewerking
bitbewerking, 108–110, 208–211, 254
`_BV()`, 237
bit clear, 205
bit set, 205
bit toggle, 205
`bit_is_clear()`, 218, 237, 394, 396
`bit_is_set()`, 218, 423, 453
bitsgewijs inverteren, 108
bitsgewijze EN, 41, 108, 113
bitsgewijze OF, 108
bitsgewijze XOR, 108
`loop_until_bit_is_clear()`, 220

loop_until_bit_is_set(), 220, 394, 396
meerdere bits wijzigen, 210
naar links schuiven, 109, 113
naar rechts schuiven, 109
bitmanipulatie, *zie* bitbewerking
bitmasker, 210, 239, 240
bitmaskeren, 113, 241, 254
bitnotatie, *zie ook* bitbewerking
bitoperator, *zie* bitbewerking
bitpositie, 210, 239
bitsgewijs inverteren, *zie* bitbewerking
bitsgewijze EN, *zie* bitbewerking
bitsgewijze OF, *zie* bitbewerking
bitsgewijze XOR, *zie* bitbewerking
bitwise, *zie* bitbewerking
blok, 39, 69
blokschema, 41–43
bloktoewijzing, 69, 86
bloktransfer, 415
Bogen, Alf-Egil, 9, 189
boolean, 107
FALSE, 107
TRUE, 107
boom, 34, 138
gebruik pointers bij, 138
boot-loader, 194, 198
bootsector, 194
bouncing, *zie* dender
boundary scan, *zie* test, boundary scan
bounded-buffer problem, 304
broncode, 14, 94
brownout, 192
brushless DC, 403
buffer, 160, 163, 287
circulaire, 303–307
fifo-, 304, 479
tristate, *zie* tristatebuffer
build_array(), 144, 146
bursttransfer, 415
busy flag, 284
button_pressed(), 217, 219, 220, 263, 266
met parameters, 219
buzzer, 404
magnetische, 404
piezo-elektrische, 404
byte, 22, 272, 297, 304

C

calc_bscale(), 312
calc_bsel(), 312
call by reference, 40–41, 54, 131
callbackfunctie, 176
calloc(), *zie* geheugenfunctie
capaciteit voor onderdrukken stoorsignalen, 201
car_backward(), 400
car_forward(), 400
car_left(), 400
car_left_curve(), 400
car_stop(), 400
case, *zie* voorwaardelijke opdracht
cat, *zie* Unix-commando
CCP, *zie* configuration change protection
ceil(), *zie* math-bibliotheek
Chan, 475
change_case(), 300
char, *zie* datatype
circulaire buffer, *zie* buffer, circulaire
CISC, *zie* Complex Instruction Set Computer
classificatie, 118
clear_screen(), 319, 373
CloseComm(), 466
CMOS, 489–502
D-flipflop, 492, 494–496, 499
D-latch, 492–496, 499
inverter, 490–491
logica, 491–492
NAND, 491–492
NOR, 491–492
pulldowntransistor, 500
pulluptransistor, 500
schmitttrigger, 423, 501–502
transmissiepoort, 493–494, 498–500
tristate-inverter, 496–498
tristatebuffer, 496–499
CMOS-technologie, 489, 491, 502
commentaar, 89–90, 94
commentaarblok, 90
einde */, 90
start /*, 90
commentaarregel, //, 90
communicatiefunctie
CreateFile(), 464
GetCommState(), 465
ReadFile(), 467
SetCommState(), 465
SetupComm(), 464
WriteFile(), 465
comparator, 420
compare/capture-blok, 385
compilatie, 14
compilatietraject, 14–15
compiler, 14, 198
cross-, 17, 106, 200
native compiler, 16
compiler directive, 186–188, *zie ook* preprocessoropdracht
compiler-optie
-D, 52
-DF_CPU, 212
-DF_CPU, 436
-Wall, 14, 29, 48, 103
-Wl,--stack, 150
-Wl,-u,vprintf, 289, 415
-c, 15, 37
-lm, 106
-lprintf_flt, 289, 415
-lprintf_min, 289
-o, 12, 15, 28, 29
-std, 83
Complementair Metal Oxide Semiconductor, 489–502, *zie ook* CMOS
Complex Instruction Set Computer, 8
Complex Programmable Logical Device, 3
conditionele toewijzing, *zie* voorwaardelijke opdracht
Config1kHzToscRTC(), 440
Config32kHzRTC(), 439, 440
Config32MHzClock(), 440
Config32MHzClock_Ext16M(), 438
configuration change protection, 435
const, 114, 173, 254, 259
constante, 93, 96, 101, 103–104, 107, 128, 133
FLT_MAX, 101
FLT_MIN, 101
RAND_MAX, 262
RANDOM_MAX, 286
UINT_MAX, 96, 283
UINT_MIN, 96
contactdender, 215–216, *zie ook* dender
contrastspanning, 271
control statements, *zie* besturingsopdracht
conversiefunctie
atoi(), 59–61, 102, 141
dtostre(), 286, 287
dtostrf(), 286, 287, 289
itoa(), 286, 314
ltoa(), 286
tolower(), 69, 300
toupper(), 69, 165, 300
uit ctype.h, 69
ultoa(), 286, 289
utoa(), 286, 289
conversietijd
ADC, *zie* ADC, conversietijd
cos(), *zie* math-bibliotheek
cosh(), *zie* math-bibliotheek
counter, 192, 229, *zie* teller *en zie ook* Xmega timer/counter
CPLD, *zie* Complex Programmable Logical Device
crosscompiler, *zie* compiler, cross-
ctype.h, *zie* standaardbibliotheek
Cyclic Redundancy Check, 192
Cygwin, 16, 97, 100, 149, 150, 162

D

D-flipflop, 492, 494–496, 499, *zie ook* CMOS
D-latch, 492–496, 499, *zie ook* CMOS
DAC, *zie* Digital-to-Analog Converter, *en ook* Xmega DAC
darlingtontransistor, 251
Data Encryption Standard, 192
databus, 7, 189, 498

datageheugen, 5, 6, 195, 416, 417
dataregister, 5
datastructuur, 138, 181–184, 370, 465, 466
  gebruik pointers bij, 138
  struct, 138, 177, 182, 183, 371
datatype, 22–23, 95–110
  char, 20, 96, 97
  double, 22, 100, 101, 103, 106
  FILE *, *zie* in- en uitvoer
  float, 22, 100–103
  float bij kleinere microcontroller, 103
  float versus **double**, 104
  int, 13, 14, 20, 55, 96, 97
    bij ATmega 32, 22, 96
  int16_t, 114
  int32_t, 114
  int8_t, 114
  long, 23, 96, 97
  long double, 100
  long long, 113
  register16_t, 180
  register32_t, 180
  register8_t, 179, 180
  representatie gebroken getallen, 100
  representatie gehele getallen, 97
  short, 23, 96
  signed, 96
  size_t, 156
  uint16_t, 114, 283
  uint32_t, 114
  uint64_t, 289
  uint8_t, 114, 220, 283, 284
  unsigned, 96
  unsigned int, 97, 286
  unsigned long, 97, 156, 286, 466
  unsigned long long, 97, 169
DB9-connector, *zie* RS232
DC-motor, 383, 397–401
debouncing, *zie* dender, anti-
debugger, 198, 200
decimaal, 104
declaratie, 14, 20, 65–66
  blok-, 40
  globale, 38, 89
  lokaal in **for**-lus, 83
  lokale, 40, 83
decrement, 27, 109
default, *zie* voorwaardelijke opdracht
#define, 77–80, 89, 91, 93, 106, 107, 186
defined, *zie* voorwaardelijke preprocessoropdracht
delay, *zie* tijdvertraging
demo_ff(), 476–478
dender, 215–216
  antidenderalgoritme, 217–220, 237–239
  antidenderschakeling, 216
  oorzaken, 215
dereferentie-operator, 133, *zie* *, dereferentie pointer
DFLL, *zie* digital frequency locked loop
digitaal-analoogconverter, 410–415
  blokschema, 410, 411
digital frequency locked loop, 436
Digital Signal Processor, 3
Digital-to-Analog Converter, 324, 325, 357, 364
DIR, *zie* Xmega ports, DIR
Direct Memory Access, 192, 415–419
disassembler, 229
diskio.c, 476
display
  grafisch, 243
  karaktergeorienteerd, 243
display_level(), 246, 248, 249, 338, 340
dissipatie, 9, 448
DMA, *zie* Direct Memory Access
do while, *zie* herhalingsopdracht
dotmatrix, 251–259
double, *zie* datatype
draadloze module, 479–488
drain, 201, 489
driehoek van Pascal, 127–130
driver
  eeprom_driver.c, 441
  eeprom_driver.h, 442
  twi_master_driver.c, 375
  twi_master_driver.h, 375
  twi_slave_driver.c, 378
  twi_slave_driver.h, 378
  usart_driver.c, 308
  usart_driver.h, 308
drukknop, 216, 237, 263
DS1307, real time clock, 370
DS3232, real time clock, 353, 370–374
  instellen van tijd, 371
  uitlezen van tijd, 371
DSP, *zie* Digital Signal Processor
dtostre(), *zie* stdlib-bibliotheek
dtostrf(), *zie* stdlib-bibliotheek
duty-cycle, 384, 386, 399, 428
  bij dual-slope-modus, 393
  bij single-slope-modus, 391
dynamisch gedissipeerde vermogen, 451
dynamische geheugenallocatie, 137, 141–150

E

echo, *zie* Unix-commando
edge triggered, *zie* flankgevoelig
edge-triggered flipflop, 494
'e'enkanaalsmethode, 411
EEPROM, *zie* Electrical Erasable Programmable Read Only Memory
  IO-mapped, 442–443
  memory-mapped, 443–445
EEPROM(), 443
eeprom-bibliotheek, 441
EEPROM_DisableMapping(), 444
eeprom_driver.c, 441
eeprom_driver.h, 442
EEPROM_EnableMapping(), 443, 444
EEPROM_FlushBuffer(), 442
EEPROM_ReadByte(), 441, 442
EEPROM_WaitForNVM(), 442–444
EEPROM_WriteByte(), 441, 442
eepromReadBuffer(), 444
eepromReadByte(), 442, 443
eepromWriteByte(), 442, 443
eindconditie
  do while, 86
  for, 82
  while, 84
Electrical Erasable Programmable Read Only Memory, 6, 194, 196, 357, 364, 441
Electro Magnetic Compatibility, 201
elektromagnetische interferentie, 201
#elif, *zie* voorwaardelijke preprocessoropdracht
else, *zie* voorwaardelijke opdracht
#else, *zie* voorwaardelijke preprocessoropdracht
embedded software, 2
embedded systeem, 1–3, 9
EMC, *zie* Electro Magnetic Compatibility
EMI, *zie* Electro Magnetic Interference
emptyBuffer(), 46
end-of-line, 161, 163, 317, *zie ook* \n
  <CR>, carriage return, 161
  <LF>, linefeed, 161
  Unix, 161
  verwijderen, 165
  Windows, 161
end-of-string, 23, 55, 63, 151, 152, 154, 156, 283, *zie ook* \0
#endif, *zie* voorwaardelijke preprocessoropdracht
Enhanced Shockburst, 481
enum, 107, 114, 116
enumeratie, 114–115
EOF, 163, 165
EPROM, *zie* Erasable Programmable Read Only Memory
Erasable Programmable Read Only Memory, 4, 6
errno.h, *zie* standaardbibliotheek
#error, 187
escape sequence, 21, 107, 514
  \', 106
  \0, nul, 106
  \\, backslash, 106
  \", 106
  \f, formfeed, 106
  \n, newline, 106
  \r, carriage return, 106
  \t, tab, 106
event-system, 192
executable, *zie* programma, uitvoerbaar
exFAT, 474
exit_with_message, 186
exp(), *zie* math-bibliotheek

exponent, 100
extern, 117, 320
externe interrupt, 237
externe klok, 192, 197
externe variabele, 117

F

F_CPU, 212, 284, 436
fabs(), *zie* math-bibliotheek
fac(), 169, 170
faculteit, 169
fade(), 396
fast-PWM, 390, 392
FAT, *zie* File Allocation Table
FAT32, 474
FatFs, 474–478
  diskio.c, 476
  f_gets(), 477
  f_write(), 477
  f_mount(), 477
  f_open(), 477
  f_printf(), 477
  FatFs, 477
  ff.c, 475
  ff.h, 477, 478
  ff_conf.h, 476
  Fil, 477
  mmc_avr.c, 476, 478
  sdmm.c, 476, 478
fclose(), *zie* in- en uitvoerfunctie
feof(), *zie* in- en uitvoerfunctie
FET, *zie* Field Effect Transistor
ff.c, 475
ff.h, 477, 478
fflush(), *zie* in- en uitvoerfunctie
fgetc(), *zie* in- en uitvoerfunctie
fgets(), *zie* in- en uitvoerfunctie
fib(), 169
Fibonacci, 119, 135
  berekenen getallen met recursie, 169
  berekenen getallen van, 121
  berekenen met pointers, 135–137
  getallen afbeelden op LCD, 281, 283
  getallen van, 119–121, 127
  reeks van, 119, 127
Field Effect Transistor, 398
  N-channel, 398
  P-channel, 398
Field Programmable Gate Array, 3
fieldwidth, format specifier, 51
fifo, 304, 479
fifo-buffer, *zie* buffer, fifo-
FILE *, *zie* in- en uitvoer
File Allocation Table, 474
fill_array(), 148
flag, format specifier, 51
flankgevoelig, 492, 494, 495
flash, 4, 6, 190, 194, 258, 259, 357, 364, 441, 445–448
  __flash, 258, 259, 445, 446
flipflop, 202, 469, *zie ook* D-flipflop
float, *zie* datatype
FLoating Point Operations Per Seconde, 4
floats.h, *zie* standaardbibliotheek
floor(), *zie* math-bibliotheek
FLOPS, *zie* FLoating Point Operations Per Seconde
flowchart, 42, 458
fopen(), *zie* in- en uitvoerfunctie
for, *zie* herhalingsopdracht
format specifier, 20, 25, 54, 286, 288
  %S, 447
  %c, 50, 104
  %d, 50, 103, 104
  %e, 50, 286, 288
  %f, 50, 103, 286, 288
  %g, 50, 286, 288
  %o, 50, 104
  %s, 50, 54
  %x, 50, 104
  bij microcontrollers, 51, 286
  optie
    fieldwidth, 51
    flag, 51
    modifier, 51
    precision, 51
format string, 20, 21
fouten
  afvangen, 61
  compilatie-, 17, 63, 83
  linker-, 17
  runtime-, 17, 61, 63–66
FPGA, *zie* Field Programmable Gate Array
fprintf(), *zie* in- en uitvoerfunctie
fputc(), *zie* in- en uitvoerfunctie
fputs(), *zie* in- en uitvoerfunctie
fread(), *zie* in- en uitvoerfunctie
free(), *zie* geheugenfunctie
freeStuds(), 183
frequentie, 404
  bij dual-slope-modus, 393
  bij frequentiemodus, 390
  bij normale modus, *zie* periodetijd bij normale modus
  bij single-slope-modus, 391
frequentiemeting, 427
fscanf(), *zie* in- en uitvoerfunctie
fseek(), *zie* in- en uitvoerfunctie
FSM, finite state machine, *zie* toestandsmachine
ftell(), *zie* in- en uitvoerfunctie
fullduplex, 463
functie, 31–37
  aanroep, 35
  body, 35, 38
  definitie, 35, 37
  gebruik pointers voor uitvoer, 138
  header, 35, 37–39
  met variabele argumentenlijst, 185–186
  naam, 35
  parameter, 35, 37
  prototype, 35, 37, 38, 60, 68, 132, 173
  returntype, 35
Fur Elise, 404
fwrite(), *zie* in- en uitvoerfunctie

G

gate, 489–490, 499
gcc, *zie* GNU C-Compiler
gedeelde klok, *zie* klokdeler
gedissipeerd vermogen, 451
geheeltallig delen, 103
geheugen
  alloceren, 136
geheugenadres, 5, 7
geheugenfunctie
  calloc(), 136
  free(), 136, 143
  malloc(), 116, 132, 136, 141, 144, 146, 148, 154, 167
  realloc(), 136
  sizeof(), *zie* operator
geheugengebruik, 66
geheugenruimte
  alloceren, 152, 154, 163, 167, 183
gemiddelde stroom, 253
gereserveerde namen, 94
get_age1(), 41
get_age2(), 41
getallen
  binaire, 96
  drijvende komma, 100, 101, 286
  gebroken, 100–101, 103–104, 286
  gebroken bij LCD, 287–290
  gehele, 96–97, *zie ook* datatype **char**, **int**, **long**, **signed**, **unsigned**
  integer, 96
  integer bij kleine microcontroller, 22, 96
  L, suffix **long**, 168
  two's complement representatie, 96
  UL, suffix **unsigned long**, 212, 431
  ULL, suffix **unsigned long long**, 113
  zwevende komma, 100
getalrepresentatie, *zie* getallen
getc(), *zie* in- en uitvoerfunctie
getchar(), *zie* in- en uitvoerfunctie
getline(), *zie* in- en uitvoerfunctie
getScore(), 44–47, 54, 58
GNU, 16
GNU C-Compiler, 14, 16
GNU C-Compiler voor AVR, 17, 198
  avr-gcc, 198
GNU-stijl, 92
groepsconfiguratie, 239, 240
groepsmasker, 239, 240
groepspositie, 239, 242

- Gulden Snede, 119–121, 135, 287

H

- H-brug, 383, 397–398
- halfduplex, 463
- handshaking, 292
- Harvard, *zie* architectuur, Harvard-
- HD44780, 270–290
  - 4-bit modus, 269, 272, 274, 280–284
  - 8-bit modus, 269, 272, 277–280, 284
  - aansluiting, 268–270
  - aansluitingen, 270, 271
  - achtergrondverlichting, 270
  - adressering geheugen, 276
  - bewegende tekst, 279–280
  - busy flag, 269, 275, 280–283
  - CGRAM, 275
  - CGROM, 275
  - clear display, 273
  - communicatie met, 271–273
  - contrast, 271
  - datalijnen, 272, 273
  - DDRAM, 275
  - E-signaal, 273–274, 280
  - enable display/cursor, 273
  - function set, 273
  - geheugens van, 275–276
  - initialisatie 4-bit modus, 279
  - initialisatie 8-bit modus, 277, 279
  - instructieset, 273
  - karakterset, 272, 275, 276
  - move cursor, 273
  - oscillatorfrequentie, 274
  - R/W-signaal, 271, 273, 280, 283
  - read busy flag, 273
  - RS-signaal, 271, 273, 279, 280, 283
  - setuptijd, 273
  - shift display/cursor, 273
  - signaalniveaus, 268
  - tijdskarakteristieken, 273, 274
  - timing bij, 273–275
  - VEE, contrastspanning, 271
  - write character, 273
- headerbestand, 13, 36, 183, 503–509
  - eigen bestand, 13, 183
  - systeembestand, 13
- heap, 65, 142, 147, 150
- Hello World, 12–14
- hergebruik van code, 33
- herhalingsopdracht, 27, 42, 81–86
  - **`do while`**, 81, 85–86, 460
  - **`for`**, 27–29, 81–84, 86, 461
  - **`while`**, 27, 30, 81, 84–85, 460
- hex-bestand, 198
- hex-code, 198, 303
- hexadecimaal, 104, 314, 315
- hierarchie, 34
- holdtijd, 495, 496
- hoofdprogramma, 221, 228, *zie* main
- hoofdroutine, 11, 13, 31, 38, 59, 61, 89, *zie ook* `main`
  - `argc`, 59, 60
  - `argv`, 59, 60, 68, 157
  - `argv[0]`, 60, 61, 157
  - **`char`**`**`, 158
  - `main`, 13, 38, 59, 61, 89
  - **`return`**, 13, 61
- hyperterminal, 299, 317
- hysterese, 423–426
- hystereselus, 424
- hysteresespanning, 424

I

- I^{2}C-bibliotheek met TWI-drivers Atmel, 375–382
- I^{2}C-bibliotheek op basis van `TWI_t`, 367–374
- I^{2}C-interface, 192, 353, 354, 364–382
  - `ACK`-bit, 365
  - bij DS3232 real time clock, 370–374
  - bit rate, 369
  - identificatiecode, 364
  - levelshifting, 374–375
  - master, 364
  - protocol ontvangen data, 365–366
  - protocol versturen data, 365
  - schrijf/leesbit, 365
  - SCL, kloklijn, 364
  - SDA, datalijn, 364
  - slave, 364
  - slave-adres, 364
  - startconditie, 365
  - stopconditie, 365
  - verschil met SPI, 364
  - versturen `0`, 365
  - versturen `1`, 365
- `i2c.c`, 368
- `i2c.h`, 368, 369
- `i2c_init()`, 368, 369, 373
- `i2c_read()`, 368, 369, 371
- `i2c_restart()`, 367–369, 371
- `i2c_start()`, 368, 369, 371
- `i2c_stop()`, 367–369, 371
- `i2c_write()`, 368, 369, 371
- IEC, 490, *zie* International Electrotechnical Commission
  - 617-12: 1991, 490
- IEEE, 100, 469, 490
  - 754 SinglePrecision Format, 100
  - JTAG 1149.1, 469
  - Std 91-1984, 490
  - Std 91a-1991, 490
- **`if`**, *zie* voorwaardelijke opdracht
- `#if`, *zie* voorwaardelijke preprocessoropdracht
- **`if-else-if`**, *zie* voorwaardelijke opdracht
- `OUT`, *zie* Xmega ports, `IN`
- in- en uitvoer
  - binary, 161
  - `EOF`, 163, 165
  - `FILE *`, 160, 315, 318
  - `FILE`-structuur, 315, 317
  - filepointer, 161, 162, 165
  - lezen en schrijven bij bestanden, 162
  - lezen uit bestand, 163
  - mode, 161
  - `stderr`, 141, 144, 186
  - `stdin`, 116, 162, 165, 286, 316, 317, 322
  - `stdout`, 162, 166, 286, 315–317, 322
  - text, 161
- in- en uitvoerfunctie, 162–163
  - `f_gets()`, 477
  - `f_write()`, 477
  - `f_mount()`, 477
  - `f_open()`, 477
  - `f_printf()`, 477
  - `fclose()`, 160, 162, 465
  - `FDEV_SETUP_STREAM()`, 315, 316, 318, 321
  - `feof()`, 166
  - `flush()`, 57, 467
  - `fgetc()`, 55, 162, 163, 165–166, 315
  - `fputc()`, 162
  - `fgets()`, 58, 116, 162–165
  - `fputs()`, 162
  - `fopen()`, 138, 160, 161, 464
  - `fprintf()`, 162, 317
  - `fputc()`, 56, 166, 315
  - `fread()`, 162, 163, 166–168
  - `fwrite()`, 162
  - `fscanf()`, 160–163, 317
  - `fseek()`, 166, 167
  - `ftell()`, 166, 168
  - `getc()`, 55, 166
  - `getchar()`, 55, 56, 97, 166
  - `getline()`, 319
  - `getline()`, 163
  - `pgm_read_byte()`, 446, 448
  - `pgm_read_byte_far()`, 448
  - `pgm_read_byte_near()`, 448
  - `pgm_read_dword()`, 446
  - `pgm_read_word()`, 446
  - `printf()`, 13, 14, 20, 55, 162, 286, 288, 314
  - `printf`, 315, 316
  - `printf_P()`, 446
  - `putc()`, 56, 166
  - `putchar()`, 55, 56, 166
  - `puts()`, 55, 56
  - `puts_P()`, 446
  - `rewind()`, 166, 168
  - `scanf()`, 54, 138, 162, 314
  - `scanf`, 315, 316
  - `setbuf()`, 468
  - `sprintf()`, 286, 287
  - `sscanf()`, 58
  - `ungetc()`, 166
  - `vfprintf()`, 186, 289, 415
- in-system programming, 197

#include, 13, 37, 89, 91, 186
increment, 27, 109
indirectie-operator, 133, *zie* *, dereferentie pointer
Industrial, Scientific and Medical, 479
info, *zie* Unix-commando
infrarood communicatie, 192
inhoud van pointer, 133, 134, 137, 208, *zie ook* *, dereferentie pointer
init_motor(), 400
init_ac(), 422, 424, 426
init_adc(), 338, 340, 342, 346, 348, 350, 414, 485
init_clock(), 438, 476, 478
init_dac(), 419
init_dma(), 419
init_inputcapture(), 428, 431, 432, 434
init_ledbar(), 338, 340
init_nrf(), 484, 485
init_pwm(), 428, 486, 487
init_rtc(), 440
init_stream(), 319–321, 373, 476, 478
init_dac(), 419
init_timer(), 348, 350, 455
init_timer(), 478
init_uart(), 312, 315, 316, 320, 345, 346, 414
init_uart_bscale_bsel(), 298–300, 302, 306
initlcd(), 279, 283
inline, 361
Input/Output, 4, 5
inputcapture-modus, 385, 427–434
inspringen, 69, 71, 91
instructie, 5, 8
instructieregister, 5
int, *zie* datatype
integer, 13, *zie* datatype **char**, **int**, **long**, **signed**, **signed**
intern EEPROM Xmega, 441–445
International Electrotechnical Commission, 491
interrupt, 5, 221–239, 255
  acties bij aanroep ISR, 228
  analoge comparator, 423
  cli(), 226, 456
  externe, 192, 222
  inputcapture, 428
  interne, 222
  Interrupt Service Routine, 222, 228, 237, 255–257, 259, 266, 302, 305, 307
  interruptvector, 222, 228
  overzicht interruptvectoren, 227
  resetvector, 228
  sei(), 226, 257, 456
  timer overflow, 407
  watchdog reset, 453
Interrupt Service Routine, 222
interruptfunctie, 256, 257, 259, 305, 307, *zie ook* interrupt, Interrupt Service Routine
interruptmechanisme, 222–223
interrupts
  asynchrone externe interrupt, 450
inverter, 490–491, *zie ook* CMOS
invoer
  geformatteerde, 52–54
  ongeformatteerde, 55–58
IO, *zie* Input/Output
IO-mapped, 442–443
IO-poort, 202, 203, *zie ook* Xmega ports
IO-register, 195, *zie ook* Xmega ports
isalnum(), *zie* testfunctie
isblank(), *zie* testfunctie
iscntrl(), *zie* testfunctie
isdigit(), *zie* testfunctie
isgraph(), *zie* testfunctie
islower(), *zie* testfunctie
ISM, *zie* Industrial, Scientific and Medical
isprint(), *zie* testfunctie
ispunct(), *zie* testfunctie
ISR, *zie* Interrupt Service Routine en *zie ook* interrupt
isspace(), *zie* testfunctie
isupper(), *zie* testfunctie
isxdigit(), *zie* testfunctie
iteratie, 137
  do while, 86
  for, 81, 82, 87
  while, 84
iteratieve functie, 169, 170
iteratieve opdracht, *zie* herhalingsopdracht
itoa(), *zie* stdlib-bibliotheek

J

JFET, *zie* Junction Field Effect Transistor
Joint Test Action Group, 197, 469–471
JTAG, 469, *zie ook* Joint Test Action Group
  debuggen via, 471
  extest, 471
  intest, 471
  programmeren via, 471
JTAG-interface, 192, 197, 200
  TAP-controller, 470
  TCK, test clock, 470
  TDI, test data in, 470
  TDO, test data out, 470
  TMS, test select mode, 470
  TRST, test reset, 470
JTAG-programmer, 198
Junction Field Effect Transistor, 251, 398, 489

K

kalibratie
  ADC, 351–352
  DAC, 413
  oscillator, 436–437
kathode, 252, 270
Kernighan, Brian, 12
keywords, *zie* gereserveerde namen
klokdeling, 255
klokflank, 494, 495
klokfrequentie, 229, 255, 274, 294, 332, 495
kloksysteem, 435, 438
koffiezetten, 42
komma-operator, *zie* operator, komma-
KS0066, 268
kwadraat, 79, 106

L

L, *zie* getallen
latch, *zie ook* D-latch
latch_data(), 360, 361
LCD, 357, 364, *zie* Liquid Crystal Display
LCD gebroken getallen, 287–290
LCD-bibliotheek, 283–290
  lcd.c, 284
  lcd.h, 284
  lcd_clear(), 284, 285, 287, 288
  lcd_cmd(), 284
  lcd_data(), 284
  lcd_gotoxy(), 284, 285, 287, 288
  lcd_home(), 284
  lcd_init(), 284, 285
  lcd_putc(), 284
  lcd_puts(), 284, 285, 287, 288
lcd4write(), 283
lcd_cmd(), 279, 281
lcd_fputc(), 318
lcd_putc(), 279, 281
lcd_puts(), 283
lcd_write(), 277
led, 200, 201, 216, 243, 270
  aansturing, 201, 250–251
  intensiteitsregeling, 394–395
Led Blink, 201
  met externe interrupt 0, 225
ledarray, *zie* dotmatrix
ledmatrix, 243, *zie ook* dotmatrix
ledspanning, 250, 252
ledstroom, 250, 252
leesbaarheid, 77, 84, 86, 90–92, 107
level sensitive, *zie* niveaugevoelig
levelshifting, 269, 374–375
  bij LCD, 269
lijst, 138, 183
  afdrukken, 184
  gebruik pointers bij, 138
  object, 139
  record, 183
  toevoegen aan, 183
  verwijderen, 184
limits.h, *zie* standaardbibliotheek
#line, 187

linker, 14, 198
linking, 14
Liquid Crystal Display, *zie ook* HD44780267, *zie ook* HD44780290
LM74, temperatuursensor, 354
`locale.h`, *zie* standaardbibliotheek
`log()`, *zie* math-bibliotheek
`log10()`, *zie* math-bibliotheek
logische bewerking, 92, 108
  EN, 41, 108
  NIET, 108
  OF, 108
`long`, *zie* datatype
look-up table, *zie ook* opzoektabel
`lookup[]`, opzoektabel, 258, 262, 264
`lookup[][]`, opzoektabel, 254, 256
loop assignment, *zie* herhalingsopdracht
`loop_until_bit_is_clear`, *zie* bitbewerking
`loop_until_bit_is_set`, *zie* bitbewerking
`ls`, *zie* Unix-commando
`ltoa()`, *zie* stdlib-bibliotheek
Lucebert, 160
luidspreker, 404

M

machinecode, 228, 229
macro, 13, 68, 77, 107, 208, 212
  `__CYGWIN__`, 465
  `max`, 80, 106
  `min`, 80
macrodefinitie, 208, 212, 360
`main`, *zie ook* hoofdroutine
`malloc()`, *zie* geheugenfunctie
`man`, *zie* Unix-commando
mantisse, 100
`MAPPED_EEPROM_START`, 445
marking, *zie* RS232
master, 354, 364, 494
master-slave flipflop, 494
math-bibliotheek
  `acos()`, 105
  `asin()`, 105
  `ceil()`, 105, 144, 146, 148
  `atan()`, 105
  `cos()`, 105
  `cosh()`, 105
  `exp()`, 105
  `fabs()`, 105
  `floor()`, 105
  `log()`, 105
  `log10()`, 105
  `pow()`, 105
  `round()`, 105, 312
  `sin()`, 105
  `sinh()`, 105
  `tan()`, 105
  `tanh()`, 105
`math.h`, *zie* standaardbibliotheek
MAX232, 292, *zie* RS232
MCU, *zie* microcontroller unit
Mega Instruction Per Seconde, 4
`memcpy()`, *zie* stringfunctie
memory-mapped, 443–445
menselijk oog, 252
Metal Oxide Semiconductor, 489
Metal Oxide Semiconductor Field Effect Transistor, 251, 398, 489
metastabiel, 495, 496
metastabiliteit, 496
microcode, 8
microcontroller, 2–9, 11, 51, 85, 100, 101, 106, 108, 131, 189–199, 221, 226, 243, 244, 263, 272, 277, 323, 325, 463, 489
  architectuur, 5
  keuze, 9
  omzet, 2
  verschil met microprocessor, 4–5
microprocessor, 3–5, 7, 11, 323
  architectuur, 4
  omzet, 2
  verschil met microcontroller, 4–5
microSD-kaart, 473–478
MinGW, 16, 52, 65, 150, 162
miniSD-kaart, 473
MIPS, *zie* Mega Instruction Per Seconde
MMC-kaart, 473
`mmc_avr.c`, 476, 478
MML, *zie* Music Markup Language
modifier, format specifier, 51
MOS, *zie* Metal Oxide Semiconductor
MOSFET, *zie* Metal Oxide Semiconductor Field Effect Transistor, *zie* Metal Oxide Semiconductor Field Effect Transistor
`motor_off()`, 400
`motor_on()`, 399, 400
MPU, *zie* microprocessor unit
Music Markup Language, 404
muziek, 404
muziek afspelen, 404–408

N

naamgeving, 93–94
naar links schuiven, *zie* bitbewerking
naar rechts schuiven, *zie* bitbewerking
NAND, *zie ook* CMOS
Nassi-Shneiderman diagram, 43
nauwkeurigheid
  ADC, 324
  bij format specifier, 51
  `double`, 100
  `float`, 100
  `long double`, 100
neveneffect, 60, 79, 92, 111, 254
New Technology File System, 474
`newStud()`, 183
NFET, *zie* Field Effect Transistor
nibble, 272
niet-atomische bewerking, 455
niveaugevoelig, 492–494
NMOS-transistor, 375, 489–491, 497–499
Nokia beltoon, 404
`nop`, 224–226, 264
Nordic, 479
NPN-transistor, 250, 251
nRF24L01
  channel, 483
  data pipe, 482–483
  Enhanced Shockburst, 481
  toestandsdiagram, 480
nRF24L01+, 479–488, 516
  register map, 479
  SPI, 479
  SPI-opdrachten, 479
NTFS, *zie* New Technology File System
`NULL`, 137, 139, 154, 156, 161, 163, 183
nullpointer, 156, 161, *zie ook* `NULL`
nulmodemverbinding, *zie* RS232
nulstand, 401–403
`NVM_EXEC()`, 442

O

object, 138
objectcode, 14, 37
octaaf, 404
octaal, 104
`offset_adc()`, 343
offsetcompensatie, 336
`offsetof()`, 351, 413
omgevingslicht, 270
onderhoudbaarheid, 77
oneindige lus, 83, 85, 279, 315
ongeformatteerde invoer, 55–58
ongeformatteerde uitvoer, 55–58
ontvanger, 292, 293, 297, 304, 484–488
`OpenComm()`, 466
operand, 24, 80, 108
operator, 95–110
  bit-, *zie* bitbewerking
  conditionele, *zie* voorwaardelijke opdracht, `?:`
  decrement, 27, 109
  increment, 27, 109
  komma-, 84
  logische, *zie* logische bewerking
  relationele, *zie* relationele bewerking
  schuif-, *zie* bitbewerking
  `sizeof``()`, 113, 116, 132, 136, 137, 178, 394, 396
opmaak, 89–94
opzoektabel, 253, 258, 261, 263, 265, 418
opzoektabel, sinus, 418
oscillator, 192, 197, 435
oscillatorfrequentie, 274
`OUT`, *zie* Xmega ports, `OUT`
`output_enable_off()`, 361
`output_enable_on()`, 361

overdraagbaarheid, 162, 164

P

package, *zie* behuizing
PAL, *zie* Programmable Array Logic
parameter, 13, 35, 37, 38
  actuele, 38–40
  formele, 38–40
  ingangs-, 13, 35
parameterlijst, 13, 38, 39, 138
pariteit, 296
pariteitsbit, *zie* RS232
parser, 165
Pascal, Blaise, 127
PDI-programmer, 198
PDIP, *zie* behuizing, Plastic Dual-In-line Package
periodetijd, 384, 385, 427
periodetijd bij normale modus, 232
periodetijdmeting, 427
PFET, *zie* Field Effect Transistor
`pgm_read_byte()`, *zie* in- en uitvoerfunctie
`pgm_read_byte_far()`, *zie* in- en uitvoerfunctie
`pgm_read_byte_near()`, *zie* in- en uitvoerfunctie
`pgm_read_dword()`, *zie* in- en uitvoerfunctie
`pgm_read_word()`, *zie* in- en uitvoerfunctie
pgmspace-bibliotheek
  `pgm_read_byte()`, 448
  `pgm_read_byte_far()`, 448
  `pgm_read_byte_near()`, 448
  `PROGMEM`, 259, 445, 446
  `PSTR()`, 446
Phase Locked Loop, 192, 435, 438
pinout, 193
pipe, 482–483
pipelined, 326
pipelining, 7, 189
plaatsvervanger, 20
`PlayRTTTL()`, 406, 407
PLC, *zie* Programmable Logic Circuit
PLD, *zie* Programmable Logical Device
PLL, *zie* phase locked loop
PMOS-transistor, 489–491, 497–499
pn-overgang, 375, 490
PNP-transistor, 251, 252
pointer, 41, 60, 63, 131–139, 151, 206, 208, 283
  bij arrays, 138
  bij bomen, 138
  bij datastructuur, 138
  bij lijsten, 138
  bij string, 138
  declaratie, 132
  fouten met, 134–135
  reken met, 133–134
  toepassingen, 138–139
  toewijzing, 132–133
pointer naar functie, 176
  declaratie, 176, 177
polling, 217–222, 263–265
`OUT`, *zie* Xmega ports
potmeter, 271
`pow()`, *zie* math-bibliotheek
power-on-reset, 227, 451
`#pragma`, 187
precision, format specifier, 51
preprocessing, 14
preprocessor, 14, 198
preprocessoropdracht, 13, 14, 77–80, 186–188, 465
prescaled clock, *zie* gedeelde klok
prescaler, *zie* klokdeler
prescaling, 255
priemgetal, 87
Princeton, *zie* architectuur, Princeton-
principe analoge comparator, 417, 420
`print_age()`, 34, 36, 37
`print_array()`, 144, 146, 148
`print_ctype()`, 72, 88
`print_digit()`, 72, 73
`printb()`, 99, 105, 113
Printed Circuit Board, 291, 353, 469
`EndOfLine()`, 78
`printf()`, *zie* in- en uitvoerfunctie
`printf_P()`, *zie* in- en uitvoerfunctie
`printNumber()`, 78
`printSpace`, 78
`printStuds()`, 183
`printText()`, 78
prioriteit, *zie* voorrangsregels
producer-consumer problem, 304
Program Debug Interface, 192, 193, 197
program structure diagram, 43
programcounter, 5
programma
  argumenten doorgeven, 59–61
  naam van het, 60, 157
  neveneffect van een, 60, 79, 92, 111
programma, uitvoerbaar, 14
Programmable Array Logic, 3
Programmable Logic Circuit, 3
Programmable Logical Device, 3
Programmable Read Only Memory, 6
programmabus, 7, 189
programmacode pc
  afdrukken Qu'etelet-index, 102
  afdrukken tweedimensionaal array, 126
  berekening getallen van Fibonacci met array, 121
  berekening getallen van Fibonacci met pointers, 136
  cijfer als tekst afdrukken, 73
  COM-poort met RS232-bibliotheek, 468
  datastructuur afdrukken, 178
  `double` en **`float`**, 104
  driehoek van Pascal, 128
  eendimensionaal array declareren met `malloc`, 141
  eendimensionaal array declareren met variable length array, 142
  eigenschappen cijfer afdrukken, 75
  eigenschappen karakter afdrukken, 72
  gehele getallen binair afdrukken, 112
  Hello World, 12
  Hello World met strings, 24
  Hello World niet-ANSI, 13
  hexadecimale en octale getallen, 104
  invoer met argumenten, 59
  iteratieve berekening faculteit, 170
  iteratieve berekening Fibonacci, 170
  leeftijd afdrukken met functie `age`, 35
  lezen en afdrukken naam en leeftijd, 52
  lezen uit bestand met fgetc, 165
  lezen uit bestand met fgets, 163
  lezen uit bestand met fread, 167
  lezen uit bestand met fscanf, 161
  lijst afdrukken, 182
  naam en leeftijd afdrukken, 62
  omzetten jaar, maand en dag, 153
  ongeformatteerd lezen en afdrukken, 55, 56
  ontvangen via de COM-poort, 467
  recursieve berekening faculteit, 169
  recursieve berekening Fibonacci, 169
  som en gemiddelde van array met afsluitteken, 30
  som en gemiddelde van array met getallen, 29
  som van even getallen, 28
  som van twee getallen, 20
  sorteren met `qsort`, 173
  sorteren met quicksort, 171
  toestandsmachine, 76
  twee keer de som van twee getallen, 21
  tweedimensionaal array declareren met apart pointerarray, 144
  tweedimensionaal array gebaseerd op eendimensionaal array, 146
  tweedimensionaal array gebaseerd op variable length array, 148
  verschil tussen == en =, 47
  versturen via de COM-poort, 465, 466
  voorbeeld met macrodefinities, 78
  voorbeeld met `strncpy` en `strlcpy`, 155
  vullen en afdrukken meerdimensionaal array, 126
programmacode Xmega
  aansturen 4-digit 7-segmentdisplay, 262
  aansturen dotmatrix, 254
  aansturen dotmatrix met opzoektabel in flash, 258
  aansturen dotmatrix met timer/counter, 256
  aansturen ledarray, 254
  aansturen ledbar, 246
  aansturen servomotor, 402

ADC differentieel freerunningmodus, 350
ADC differentieel met timer, 348
ADC handmatig, differentieel, 345
ADC handmatig, differentieel met interrupt, 346
ADC handmatig, signed single-ended, 344
ADC handmatig, unsigned single-ended, 338, 340
afspelen RTTTL-beltoon, 406
analoge comparator, 422, 424
analoge comparator met interrupt en scaler, 423
analoge comparator met windowmodus, 426
antidenderalgoritme, 237
benaderen DS3232 via I^{2}C, 373, 377
besturen robotwagen, 400
demo microSD-kaart met delay, 477
demo microSD-kaart met timer, 478
DMA, 416
driehoekvormig signaal met DAC, 412
drukknop met polling, 217, 219
extern EEPROM benaderen via SPI, 358
LCD gebroken getallen met `dtostrf`, 288, 289
LCD gebroken getallen met `sprintf`, 288, 289
generatie sinus, 418, 419
I^{2}C-bibliotheek, 369
initialisatie van nRF24L01+, 484
knipperende led met RTC, 440
LCD met acht datalijnen, 277
LCD met acht datalijnen en bewegende tekst, 280
LCD met LCD-bibliotheek, 285
LCD met vier datalijnen, 282
ledblink, met `TCE0` zonder ISR, 234
ledblink, met externe interrupt `0`, 225
ledblink, met frequentiemodus, 235
ledblink, met timer/counter `0`, 232–233
met SPI naar 74595 schrijven, 361
ontvangen met nRF24L01+, 486
periodetijd en pulsbreedte meten met inputcapture, 428
programma met DAC, ADC en UART, 414
PWM-signaal met frequentiemodus, 389
PWM-signaal met single-slope, 391
regeling lichtintensiteit led, 394
regeling voor rgb-led, 396
rtc-bibliotheek voor DS3232, 371
schrijven naar en lezen uit EEPROM, 441
schrijven naar en lezen uit EEPROM met memory-mapping, 443
slaapstand idle met `TCC0` als wekker, 450
slaapstand idle met timer/counter als wekker, 449
TWI master ontvangt gegevens, 381
TWI master stuurt gegevens, 379
TWI slave ontvangt gegevens, 379
TWI slave stuurt gegevens, 381
UART met circulaire buffer, 306
UART met wrapper, 308
UART, met printf, 315
UART, met printf en scanf, 316
UART, versturen en ontvangen, 300
UART, versturen en ontvangen met een interrupt, 302
UART, versturen getallen met wrapper, 314
UART, versturen van gegevens, 299
uitlezen drukknop+, 217, 220
vier PWM-signalen met frequentiemodus, 389
vier PWM-signalen met single-slope, 392, 394
watchdog principe, 453
watchdog voorbeeld, 453
zenden met nRF24L01+, 485
zes drukknoppen met interrupt, 264
zes drukknoppen met polling, 264
programmageheugen, 5, 194, 445–448
programmateller, *zie* programcounter
programmer, 197, 198
PROM, *zie* Programmable Read Only Memory
prototype, 35, 37, 38, 64, 68, 89, 107, 132, 173, 183, *zie ook* functie, prototype
PSD, *zie* program structure diagram
pseudocode, 41–46
aansturen 7-segmentdisplay, 261
aansturen dotmatrix, 253
aansturen ledbar, 245
pulldown, 203
pulldowntransistor, *zie ook* CMOS
pullup, 203
pulluptransistor, *zie ook* CMOS
pullupweerstand, 216
pulsbreedte, 384, 402
bij servomotor, 402
pulsbreedtemodulatie, 383–408
aansturing DC-motoren, 397–401
aansturing servomotor, 401–403
bij een pulsvormig signaal, 384
bij een sinusvormig signaal, 384
dual-slope, 385, 393, 397–403
duty-cycle, 384
frequentiemodus, 385, 388–390, 404–408
het aansturen van een led, 384
muziek afspelen, 404–408
normale modus, 385, 387–388
regeling intensiteit led, 394–395
regeling intensiteit rgb-led, 395–397
relatieve pulsduur, 384
single-slope, 385, 390–392, 394–397
pulsduur, 384, 385, 401
bij servomotor, 401
Pulse Width Modulation, 192, 383–408, *zie ook* pulsbreedtemodulatie
`putc()`, *zie* in- en uitvoerfunctie
`putchar()`, *zie* in- en uitvoerfunctie
`puts()`, *zie* in- en uitvoerfunctie
`puts_P()`, *zie* in- en uitvoerfunctie
Putty, 299, 341, 343, 373
PWM, 324, 383–408, *zie ook* Pulse Width Modulation

Q

`qsort()`, *zie* stdlib-bibliotheek
Qu'etelet, Adolphe, 101
qualifier, 259, 360, 445
`quicksort()`, 171

R

RAM, 258, *zie* Random Access Memory
`rand()`, *zie* stdlib-bibliotheek
`random()`, *zie* stdlib-bibliotheek
Random Access Memory, 4–6, 194, 304, 441
`random_r()`, *zie* stdlib-bibliotheek
Read Only Memory, 6
read-modify-write, 207
`read_adc()`, 338, 340, 342–345, 414, 485
`readCalibrationByte()`, 413
`readCalibrationWord()`, 351
`ReadCommByte()`, 467
`readRTTTLdefaults()`, 406
`readRTTTLnote()`, 405, 406
real time clock, 353, 357, 364
met DS3232, 370–374, 376–378
`realloc()`, *zie* geheugenfunctie
realtime-systeem, 3
recept, 42
recursie, 168–175
recursie versus iteratieve oplossingen, 169–170
Reduced Instruction Set Computer, 8
referentie, 325, 327, 331
referentiespanning, 331, 420, 424
ADC, 331–332, 336, 337, 341, 345
analoge comparator, 420
regeling intensiteit led, 394–395
regeling intensiteit rgb-led, 395–397
rekeneenheid, centrale, 4
rekenkundige bewerking, 92
aftrekken, `-`, 24, 105, 133
bij microcontroller, 106
delen, `/`, 24, 92, 105
`floor()`, 106
machtverheffen, 106
modulus, `%`, 24, 28, 105, 261

optellen, +, 20, 24, 92, 105, 133
pow(), 106
remainder, 105
uit math.h, 105
uit stdlib.h, 106
vermenigvuldigen, *, 24, 92, 105
relatieve pulsduur, 384
relationele bewerking, 107
!=, 26, 107
<=, 26, 107
<, 26, 107
==, 26, 107
>=, 26, 107
>, 26, 107
representatie, 95, 97, 100, *zie ook* getallen
resetvector, *zie* interrupt, resetvector
return, 35, 37, 38, 40, 88, *zie ook* hoofdroutine
rewind(), *zie* in- en uitvoerfunctie
rgb-led, 383, 395–397, 516
intensiteitsregeling, 395–397
Ring Tone Text Transfer Language, 404–408
ringbuffer, *zie* buffer, circulaire
RISC, *zie* Reduced Instruction Set Computer
Ritchie, Dennis, 12
ROM, *zie* Read Only Memory
round(), *zie* math-bibliotheek
RS232, 192, 291, 463
baud rate, 293–295, 464
databits, 296–298, 464, 465
DB9-connector, 298, 463
marking, 463
MAX232, 292
nulmodemverbinding, 292, 463
pariteitsbit, 296–298, 464, 465
protocol, 296, 463–464
RX, 292, 298, 463
spacing, 463
startbit, 296, 464
stopbit, 296–298, 464, 465
TX, 292, 298, 463
rtc.c, 371
rtc.h, 371
rtc_get_date(), 371, 376, 377
rtc_get_time(), 372
rtc_set_date(), 372
rtc_set_time(), 371, 376, 377
rtc_time_to_string(), 372, 373
RTTTL, *zie* Ring Tone Text Transfer Language
ruis, 423, 501
runtime errors, *zie* fouten, runtime-

S

safety loop, 454
samengesteld datatype, 23–24, 177–180
sample&hold, 326, 410
scan path, *zie* test, scanpad
scanf(), *zie* in- en uitvoerfunctie
scheduling, 221, *zie* tijdplanning
schema
aansturing DC-motor met TB6552, 399
aansturing fade led, 394
aansturing LCD met acht datalijnen, 277
aansturing LCD met vier datalijnen, 281
aansturing luidspreker, 404
aansturing magnetische buzzer, 404
aansturing rgb-led, 395
analoge comparator met gedeelde spanning als referentie, 423
analoge comparator met hysterese, 423
demonstratie analoge comparator, 421
met led en drukknop, 213
meting met ADC, 337
piezo-elektrische buzzer, 404
seriele verbinding met de UART, 298
voor aansturen ledbar, 245
voor inputcapture, 428
voor knipperen led, 200
zes drukknoppen en een 7-segmentdisplay, 263
schmitttrigger, 203, 216, 423, 501–502
schuifoperator, *zie* bitbewerking
scope, 38, 69
block, 38, 39
file, 38
function, 38
function prototype, 38, 39
SD-kaart, 473–478
SD-kaarthouder, 516
sdmm.c, 476, 478
segmentation fault, 61, 65
sei(), 257
Serial Peripheral Interface, 5, 192, 291, 353–358, 364
communicatie met nRF24L01+, 479
communicatie met SD-kaart, 474
master mode, 354
MISO, Master In Slave Out, 354
MOSI, Master Out Slave In, 354
SCK, Spi Clock, 354
slave mode, 354
SS, Slave Select, 354
verschil met I^2C, 364
serialF0, 320–322
init_stream(), 321
uartF0_getc(), 321
uartF0_putc(), 321
serialF0.c, 320
serialF0.h, 320
serieel, 463
seriele communicatie, 292, 353–382, 463
servomotor, 383, 401–403
set_adcch_input(), 350
set_level_array, 396
set_usart_txrx_direction(), 312, 313
set_usartctrl(), 295, 299, 300
setbuf(), *zie* in- en uitvoerfunctie
setuptijd, 273, 495, 496
short, *zie* datatype
show_fibonacci(), 285, 287–289
showResult(), 44
signed, *zie* datatype
simplex, 463
simulator, 200
sin(), *zie* math-bibliotheek
sinh(), *zie* math-bibliotheek
sink, 201
sinus, 418
sinus[], opzoektabel, 418
size_t, *zie* datatype
sizeof(), *zie* operator
slaapstand, 448–451, *zie ook* Xmega slaapstanden
extended standby, 449
idle, 449–450
power-down, 449–451
power-save, 449
standby, 449
slave, 354, 364, 494
SlaveReceiveData(), 379, 381
sleep mode, 192, 448–451
sleep-bibliotheek
extended standby, 449
idle, 449–450
power-down, 449–451
power-save, 449
set_sleep_mode(), 449
sleep.h, 449
sleep_mode(), 449
SLEEP_MODE_IDLE, 450
standby, 449
sleutel, 253
SMD, *zie* Surface Mounted Device
sorteren, 171–175
qsort, 171–175
quicksort, 171
source, 201, 489, 499
spacing, *zie* RS232
SPI, *zie* Serial Peripheral Interface
spi_eeprom.h, 357
spi_eeprom_read_byte(), 357
spi_eeprom_write_byte(), 357, 358
spi_init(), 356, 358, 361, 363
spi_read(), 356
spi_transfer(), 356, 357, 361, 363
spi_write(), 356
sprintf(), *zie* in- en uitvoerfunctie
sprongopdracht, 86–88
break, 73, 74, 86–88, 171
continue, 86–88, 308
square(), 79, 176
srand(), *zie* stdlib-bibliotheek
srandom(), *zie* stdlib-bibliotheek
sscanf(), *zie* in- en uitvoerfunctie
stack, 65, 142, 149–150, 222, 223
Stallman, Richard, 16

standaard stream, 317, 322
standaard Unix-bibliotheek
  unistd.h, 465
standaard Windows-bibliotheek
  windows.h, 466
standaardbibliotheek
  assert.h, 509
  ctype.h, 68, 107, 507
  errno.h, 509
  floats.h, 101, 507–508
  limits.h, 96, 507
  locale.h, 508
  math.h, 106, 509
  stdarg.h, 185, 186, 505
  stdbool.h, 107
  stddef.h, 351, 413, 506
  stdint.h, 114, 180
  stdio.h, 13, 503–504
  stdlib.h, 60, 106, 261, 505
  string.h, 64, 139, 506
  time.h, 508
standaardinvoer, 55, 162
standaarduitvoer, 56
standard library, *zie* standaardbibliotheek
starcmp(), 174
starcmp_size(), 175
starcmp_reverse(), 174
start_freq_timer(), 406, 407
start_ms_timer(), 406, 407
startbit, 296, *zie* RS232
startconditie
  do while, 86
  for, 82
  for, zonder start- en eindconditie, 83
  for, zonder startconditie, 170
  while, 84
state machine, *zie* toestandsmachine
static, 117, 256, 412
Static Random Access Memory, 190, 195
statische functie, 117–118
statische variabele, 117–118
status
  van het programma, 13, 61
statusregister, 5
stdarg-bibliotheek
  ..., 185
  va_arg(), 185
  va_end(), 186
  va_list, 185, 186
  va_start(), 185, 186
stdarg.h, *zie* standaardbibliotheek
stdbool.h, *zie* standaardbibliotheek
stddef.h, *zie* standaardbibliotheek
stderr, *zie* in- en uitvoer
stdin, *zie* in- en uitvoer
stdint.h, *zie* standaardbibliotheek
stdio.h, *zie* standaardbibliotheeken zie ook in- en uitvoer(functies)519
stdlib-bibliotheek
  abs(), 106
  atoi(), 59–61, 102, 141
  calloc(), 136
  dtostre(), 286, 287
  dtostrf(), 286–289
  free(), 136, 143
  itoa(), 286, 314
  ltoa(), 286
  malloc(), 132, 136, 141, 144, 146, 148, 154, 167
  qsort(), 173
  rand(), 106, 261
  random(), 286
  random_r(), 286
  realloc(), 136
  srand(), 106
  srandom(), 286
  ultoa(), 286
  utoa(), 286
stdlib.h, *zie* standaardbibliotheek
stdout, *zie* in- en uitvoer
stoorsignalen onderdrukken, 201
stop_freq_timer(), 406, 407
stop_ms_timer(), 406, 407
stopbit, 296, *zie* RS232
strcat(), *zie* stringfunctie
strchr(), *zie* stringfunctie
strcmp(), *zie* stringfunctie
strcpy(), *zie* stringfunctie
stream, 314–317
stream.c, 319
stream.h, 319
string, 13, 23, 24, 66, 131, 151–158
  einde van, *zie* \n *en zie ook* end-of-string
  format, 54
  gebruik pointers bij, 138
  toekennen aan een string, 63
string.h, *zie* standaardbibliotheek
string_to_rtc_time(), 372, 373, 377
stringfunctie, 64
  memcpy(), 444
  strcat(), 64, 156
  strchr(), 156
  strcmp(), 64, 140, 152, 156, 174
  strcpy(), 63, 64, 139–140, 154, 156, 177, 182
  strlcat(), 156
  strlcpy(), 154, 156
  strlen(), 64, 154, 156, 164, 175
  strlwr(), 156
  strncat(), 156
  strncmp(), 156
  strncpy(), 154, 156, 157
  strpbrk(), 165
  strrchr(), 156
  strstr(), 156
  strtok(), 156
  strupr(), 156
strlcat(), *zie* stringfunctie
strlcpy(), *zie* stringfunctie
strlen(), *zie* stringfunctie
strlwr(), *zie* stringfunctie
strncat(), *zie* stringfunctie
strncmp(), *zie* stringfunctie
strncpy(), *zie* stringfunctie
stroom afvoeren, 201
stroom leveren, 201
stroomdiagram, 41–43, 457–462
  actiesymbool, 458
  beslissingssymbool, 458
  connectiesymbool, 459
  eindsymbool, 459
  inout-symbool, 459
  offpage-symbool, 459
  pijl, 458
  processymbool, 458, 459
  startsymbool, 459
stroomverbruik, 4, 352, 451
strpbrk(), *zie* stringfunctie
strrchr(), *zie* stringfunctie
strstr(), *zie* stringfunctie
strtok(), *zie* stringfunctie
struct, 116, *zie* datastructuur
structuur, 89–94, 469
strupr(), *zie* stringfunctie
successieve approximatie, 325
suffix, 114–115
swap(), 171
switch, *zie* voorwaardelijke opdracht
synchronizer, 496
synchroon, 292, 293, 452
systeemfunctie
  exit(), 464
  sleep(), 465

T

tan(), *zie* math-bibliotheek
tanh(), *zie* math-bibliotheek
TB6552, dual H-bridge, 399
tekenbit, 100
teller, 5, 229, 255, *zie ook* timer
temperatuursensor, 354, 357, 364
Tera Term, 299, 373
test
  bed of needles, 469
  boundary scan, 469–471
  boundary scan flipflop, 470
  functionele, 469
  productie-, 469
  scanpad, 470
  structurele, 469
  testvector, 470
testfunctie
  isalnum(), 69, 71
  isblank(), 69
  iscntrl(), 69, 87
  isdigit(), 68, 69, 71
  isgraph(), 69
  islower(), 69, 300
  isprint(), 69
  ispunct(), 69, 71
  isspace(), 69, 165

isupper(), 69, 71, 300
isxdigit(), 69
uit ctype.h, 69
Thomson, Kennith, 12
tijdplanning, 257
tijdsduur, 385, 404
tijdvertraging
met _delay_ms(), 217, 219, 254, 262, 264, 277, 280, 282, 299
met _delay_us(), 282
time.h, *zie* standaardbibliotheek
timer, 192, 229, 255–257, 383–408
timer/counter, 229–239, *zie ook* Xmega timer/counter
timer/counter capture interrupt, 428
timer/counter overflowinterrupt, 237
toekenning, 20
toestandsdiagram, 480
toestandsmachine, 76
diagram, 76
Mealy, 76
Moore, 76
toestand, 76
toestandsovergang, 76
toetsenbord, 2, 199, 243, 314
toewijzing, 20
toggelen, 109, 214
TQFP, *zie* behuizing, Thin Quad Flat Pack
transactie, 415
transducer, 323
transistor, 270
transmissiepoort, 203, 493–494, 498–500, *zie ook* CMOS
triangle(), 412
tristate-inverter, 496–498, *zie ook* CMOS
tristatebuffer, 203, 496–499, *zie ook* CMOS
TWI, *zie* Two-Wire serial Interface, *en ook* Xmega TWI
twi_master_driver.c, 375
twi_master_driver.h, 375
TWI_MasterInit(), 377, 379, 381
TWI_MasterInterruptHandler(), 377, 379, 381
TWI_MasterRead(), 381
TWI_MasterWrite(), 379
TWI_MasterWriteRead(), 376
twi_slave_driver.c, 378
twi_slave_driver.h, 378
TWI_SlaveInitializeDriver(), 379, 381
TWI_SlaveInitializeModule(), 379, 381
TWI_SlaveInterruptHandler(), 379, 381
Two Wire Interface, 291, 353
two’s complement, *zie* representatie
Two-Wire serial Interface, 192, 353, 354, 364–382
type checking, 12
typecasting, 97–99, 101–102, 290
typedef, 107, 116, 138, 156, 177–179
typedefinitie, 89, 116–117, 178

U

UART, *zie* Universal Asynchronous Receiver and Transmitter
databits, 296
init_stream(), 319, 320, 373
met driver Atmel, 307–317
ontvangen gegevens, 300
pariteit, 296
startbit, 296
stopbit, 296
versturen en ontvangen met circulaire buffer, 303–307
versturen en ontvangen met interrupt, 301
versturen gegevens, 299, 300
wrapper, 307–317, 320
uart.c, 319
uart.h, 319
uart_fgetc(), 315–318
uart_fputc(), 315–318
uart_getc(), 306, 307, 316, 320
uart_init(), 306–308, 315, 316
uart_init_bscale_bsel(), 300
uart_putc(), 306–308, 316, 320
uart_puts(), 306–308
uartF0_getc(), 321
uartF0_putc(), 321
uint8_t, *zie* datatype
uitvoer
geformatteerde, 50–52
ongeformatteerde, 55–58
UL, *zie* getallen
ULL, *zie* getallen
ultoa(), *zie* stdlib-bibliotheek
#undef, 187
ungetc(), *zie* in- en uitvoerfunctie
struct, 177
union, 177–180
Universal Asynchronous Receiver and Transmitter, 5, 292
Universal Serial Bus, 192
Universal Synchronous and Asynchronous Receiver and Transmitter, 192, 291, 373
Unix, 49, 161
end-of-line, 161
Unix-commando
avr-gcc, *zie ook* GNU C-Compiler voor AVR
cat, 59
echo, 61
gcc, *zie ook* GNU C-Compiler
info, 157
ls, 15
man, 157
unsigned, *zie* datatype
unsigned long long, *zie* datatype
USART, *zie* Universal Synchronous and Asynchronous Receiver and Transmitter *en ook* Xmega UART 519
USART_DataRegEmpty(), 311
usart_driver.c, 308
usart_driver.h, 308
USART_Format_Set(), 312
USART_GetChar(), 310
USART_InterruptDriver_Initialize(), 310, 312
USART_PutChar(), 310
USART_RXComplete(), 311
USB, 197
USB-interface, 198
utoa(), *zie* stdlib-bibliotheek

V

va_arg(), *zie* stdarg-bibliotheek
va_end(), *zie* stdarg-bibliotheek
va_list, *zie* stdarg-bibliotheek
va_start(), *zie* stdarg-bibliotheek
variabele, 38
globale, 38, 65, 259, 303
lokale, 38, 65, 83
variabele argumentenlijst, 185–186
variable length array, 141, 148
bij eendimensionaal array, 142
bij tweedimensionaal array, 148
zinvol voorbeeld, 143
VCC, digitale voedingsspanning, 201, 271, 298
verdeel-en-heers, 32–33
verdeel-en-heersstrategie, 34
vergelijkingsoperator, *zie* relationele bewerking
verkorte schrijfwijze, 84, 109–110, 113
vermogen, 3, 448
vermogensbesparing, 221
vermogensverbruik, 448
verversingsfrequentie, 252
verversingstijd, 252, 254
verwijderen end-of-line, 165
Visser van Ma Yuan, 160
VLA, *zie* variable length array
vluchtig, 6, 118, 194
void, 13, 35, 40
volatile, 118, *zie* vluchtig
volatile, 208, 224–226, 255, 274, 309, 379, 430, 455
volatile, 118
volatile pointer, 118, 255
volume(), 38
von Neumann, John, 7
voorrangsregels, 92, 108, 110, 111
voorwaardelijke opdracht, 25–26, 67–80
?:, 80, 87, 106
case, 73
default, 73

else, 26, 69–70, 460
if, 26, 68–70, 72, 76, 91, 460
if-else-if, 26, 72, 460
if-else-if versus **switch**, 72, 75
nesten van **if**'s, 70–71
switch, 72–77, 460
voorwaardelijke preprocessoropdracht
defined, 188, 465
#elif, 188
#else, 187, 188, 465
#endif, 187, 188, 465
#if, 187, 188, 465
#ifdef, 188
#ifndef, 188

W

watchdog, 227, 452–454, *zie ook* Xmega watchdog
watchdog-bibliotheek, 452–454
wdt.h, 452, 453
wdt_disable(), 452, 453
wdt_enable(), 452, 453
wdt_reset(), 452, 453
watchdogmechanisme, 452
watchdogtimer, 192, 452
while, *zie* herhalingsopdracht
white space, 69, 163, 165
wifi, 483
Windows, 49, 161, 464
end-of-line, 161
windows.h, *zie* standaard Windows-bibliotheek
witte regels, 91
Wollan, Vegard, 9, 189
wrapper
uart_getc(), 315, 316
uart_init(), 315, 316
uart_putc(), 315, 316
WriteCommByte(), 466

X

Xmega, 9, 104, 189–198
AVCC, analoge voeding, 201
C voor AVR, 273–274, 277–290
CCP, configuration change protection, 436–440
CCP_IOREG_gc, 436–438
EEPROM, 196
External Bus Interface, 195
externe klok, 197
fusebit, 196
general purpose register, 195
generieke IO, 203
in- en uitgangsregister, 195
indeling datageheugen, 195
indeling programmageheugen, 194
interruptniveaus, 180, 223
JTAG-interface, 471
kristaloscillator, 197
lockbit, 196
ontwikkeltraject, 198
pinout, 193
PMIC, 224–226
programmeren via JTAG, 471
RAMPZ, 447
Z, 447
SRAM, 195
systeemklok, 196
toelaatbare stroom, 250
VCC, digitale voeding, 201
VREFA, 331, 412
VREFB, 331, 412
Xmega ADC
analoge referentie, 331–332
automatisch, 347–351
automatisch converteren, 333–334
CH*n*, channel n, 328, 334
converteren met event trigger, 333–334
converteren met freerunningmodus, 333–334
differential mode, 328
differentieel, 329, 344–351
differentieel met freerunningmodus, 351
differentieel met timer, 347–349
handmatig converteren, 333–334
ingangsselectie, 328–329
interne signaal als ingang, 413–415
offsetfout, 335
opbouw, 328
overzicht instellingen, 352
prescaler, 327, 332
referentiespanning, 327, 331–332
resultaat, differentieel, 331
resultaat, signed single-ended, 331, 343
resultaat, unsigned single-ended, 330, 340
signed single-ended, 329, 341–344
single-ended mode, 328
uitgangsregisters, 329–331
unsigned single-ended, 329, 330, 337–341
versterkingsfout, 335
Xmega ADC channel
ADC_CH_CHIF_bm, 334
ADC_CH_INPUTMODE_DIFF_gc, 346, 348, 350
ADC_CH_INPUTMODE_INTERNAL_gc, 414
ADC_CH_INPUTMODE_SINGLEENDED_gc, 338, 340
ADC_CH_INTLVL_LO_gc, 346
ADC_CH_MUXINT_DAC_gc, 414
ADC_CH_MUXNEG_GND_gc, 342
ADC_CH_MUXPOS_gm, 343
ADC_CH_START_bm, 334, 346, 348, 350
ADCA_CH0_vect, 346, 348, 350
CTRL, 328, 334
INTCTRL, 346, 348, 350
INTFLAGS, 334
MUXCTRL, 328, 338, 340, 343, 345
RES, resultaat register, 328, 330, 334
RESH, resultaat hoge byte, 330
RESL, resultaat lage byte, 330
Xmega ADC registers
ADC_CH*n*IF_bm, 334
ADC_CH*n*START_bm, 334
ADC_CONMODE_bm, 339
ADC_EVACT_CH0_gc, 348
ADC_EVACT_NONE_gc, 350
ADC_EVSEL_0123_gc, 348, 350
ADC_FREERUN_bm, 339, 350
ADC_REFSEL_INTVCC_gc, 338, 340
ADC_RESOLUTION_12BIT_gc, 338–340
ADC_SWEEP_0123_gc, 350
ADC_SWEEP_0_gc, 348
ADCB, 339
CAL, 351
CH*n*RES, 329, 334
CONVMODE-bit, 330
CTRLA, 334
CTRLB, 330, 334
EVACT, 349
EVCTRL, 348–350
EVSEL-bits, 349
FREERUN, 339
FREERUN-bit, 334
INTFLAGS, 334
PRESCALER, 332, 338, 340
REFCTRL, 338, 340
RESOLUTION-bit, 330
SWEEP-bits, 349
Xmega analoge comparator, 420–426
AC0OUT, 424
AC_AC0OUT_bm, 422
AC_AC0STATE_bm, 423
AC_ENABLE_bm, 422, 426
AC_HYSMODE_LARGE_gc, 424
AC_HYSMODE_NO_gc, 424
AC_HYSMODE_SMALL_gc, 424
AC_INTLVL_LO_gc, 422
AC_INTMODE_BOTHEDGES_gc, 422
AC_MUXNEG_DAC_gc, 421
AC_MUXNEG_PIN*n*_gc, 421, 422
AC_MUXNEG_SCALER_gc, 421, 422, 424
AC_MUXPOS_DAC_gc, 421
AC_MUXPOS_PIN*n*_gc, 421, 422
AC_WEN_bm, 426
AC_WINTMODE_INSIDE_gc, 426
AC_WINTMODE_OUTSIDE_gc, 426
AC_WSTATE_INSIDE_gc, 426
ACA_AC0_vect, 423
ACA_ACW_vect, 426
AC*n*CTRL, 422, 426
AC*n*MUXCTRL, 420, 422
blokschema, 420
CTRLB, scaler, 422, 424
HYSMODE-bits, 424
hysterese, 424–425

SCALER, 422, 424
schaalfactor, 422
WINCTRL, 426
windowmodus, 425–426
WINTMODE-bits, 426
WSTATE-bits, 426
Xmega DAC, 410–415
blokschema, 410, 411
CH0GAINCAL, 413
CH0OFFSETCAL, 413
CH*n*DATA, 412
CHSEL-bits, 411
IDOEN-bits, 411
CTRLA, 411, 412
CTRLB, 411, 412
CTRLC, 412
DAC_CH*n*DRE_bm, 412
DAC_CH*n*EN_bm, 412, 414, 419
DAC_CHSEL_SINGLE_gc, 412–414
DAC_ENABLE_bm, 412, 419
DAC_IDOEN_bm, 414
DAC_REFSEL_AVCC_gc, 412
DACB0GAINCAL, 413
DACB0OFFCAL, 413
dual channel, 410
single channel, 410
STATUS, 412
uitgangsspanning DAC, 412
Xmega DFLLRC2M
CTRL, 437
DFLL_ENABLE_bm, 437
Xmega DFLLRC32M
CTRL, 437
DFLL_ENABLE_bm, 437
Xmega DMA
ADDRCTRL, 416, 419
block transfer, 415
burst transfer, 415
CTRL, 416, 419
DESTADDRn, 416, 419
DMA_CH_BURSTLEN_8BYTE_gc, 416, 419
DMA_CH_ENABLE_bm, 416, 419
DMA_CH_TRFREQ_bm, 416, 419
DMA_CH_TRNIF_bm, 416, 419
DMA_ENABLE_bm, 416, 419
SRCADDRn, 416, 419
transaction, 415
TRIGSRC, 416, 419
Xmega EEPROM, 441–445
adressering, 441
EEPROM-drivers van Atmel, 441
lezen uit, 441–445
schrijven naar, 441–445
Xmega event system
CHnMUX, 348, 428, 432
EVSYS, 348, 428, 431, 434
EVSYS_CHMUX_PORTE_PIN1_gc, 428, 431, 432, 434
EVSYS_CHMUX_TCD0_OVF_gc, 419
EVSYS_CHMUX_TCE0_OVF_gc, 348, 432
Xmega flash
FSTR, 447
pgm_read_byte(), 413, 445
pgm_read_dword(), 446
pgm_read_float(), 446
pgm_read_word(), 351, 446
printf_P(), 446
PSTR(), 446
Xmega interruptvector, 180, 226, 227
overzicht, 227
PORT*x*_INT0_vect, 224, 237, 264, 450, 487
Xmega klok, 435–437, *zie ook* Xmega oscillator
32 MHz klok met extern 16 MHz kristal, 438
CLK_RTCEN_bm, 439, 440
CLK_RTCSRC_RCOSC32_gc, 439
CLK_RTCSRC_TOSC_g, 440
CLK_SCLKSEL_PLL_gc, 438
CLK_SCLKSEL_RC32M_gc, 436
Config32MHzClock, 435
CTRL, 436, 438
externe oscillator, 437
overzicht, 435, 438
RTCCTRL, 439, 440
verbeteren 2 MHz klok, 437
verbeteren 32 MHz klok, 437
verbeteren 32 MHz klok met extern kristal, 437
Xmega NVM
ADDR*n*, 442
CMD, 442
NVM_CMD, 351, 413
NVM_CMD_ERASE_WRITE_EEPROM_PAGE_gc, 442
NVM_CMD_LOAD_EEPROM_BUFFER_gc, 442
NVM_CMD_NO_OPERATION_gc, 351, 413
NVM_CMD_READ_CALIB_ROW_gc, 351, 413
NVM_CMD_READ_EEPROM_gc, 442
NVM_PROD_SIGNATURES_t, 351, 413
Xmega oscillator, 435–437, 439–441, *zie ook* Xmega klok
CTRL, 436
DFFL, *zie* Xmega DFLLRC2M en Xmega DFLLRC32M
DFLLCTRL, 437, 439
externe oscillator, 437
OSC_FRQRANGE_12TO16_gc, 438
OSC_PLLEN_bm, 438
OSC_PLLFAC_gm, 438
OSC_PLLSRC_XOSC_gc, 438
OSC_RC2MCREF_RC32K_gc, 437
OSC_RC32KEN_bm, 437, 439
OSC_RC32MCREF_RC32K_gc, 437
OSC_RC32MCREF_XOSC32K_gc, 437, 439
OSC_RC32MEN_bm, 436
OSC_RC32MRDY_bm, 436
OSC_XOSCEN_bm, 437, 438, 440
OSC_XOSCSEL_32KHz_gc, 437, 440
OSC_XOSCSEL_XTAL_16KCLK_gcc, 438
STATUS, 436
Xmega ports, 202–208
asynchrone interrupt, 450
DIR, direction register, 203, 205
DIRCLR, 205, 213
DIRSET, 205, 283
externe interrupt 0, 224, 487
IN, ingangsregister, 204, 496
INT0MASK, 224–226
INTCTRL, 207, 224–226
INTFLAGS, 207
INT*n*MASK, 207
ISC-bits, 204, 225
MPCMASK, 264, 265, 396
OPC-bits, 204, 218, 219
OUT, uitgangsregister, 203, 205, 208
OUTCLR, 205
OUTSET, 205
OUTTGL, 205, 224–226
PIN*n*CTRL, 204, 218, 220, 264, 265, 373, 377, 394, 396, 402, 428, 450
PORT_INT0LVL_LO_gc, 224–226
PORT_INT0LVL_OFF_gc, 237
PORT_ISC_BOTHEDGES_gc, 428
PORT_ISC_FALLING_gc, 224–226
PORT_OPC_PULLUP_gc, 218, 220, 264, 265, 396, 450
PORT_OPC_WIREDANDPULL_gc, 373, 377, 394, 402
PORT_SPI_bm, 362
PORT_t, datastructuur, 179, 205–207, 220
PORTCFG, 264, 265, 396
PORTD_INT0_vect, 224–226, 237, 266
PORT*x*, 204–206
pullup, 217, 500
REMAP, 179, 207, 362
Xmega reset, 451
brownout-reset, 452
externe reset, 201, 228, 451
PDI-reset, 452
power-on-reset, 227, 451
reset aansluiting, 201
resetvector, 227
RST, 453
RST_WDRF_bm-bits, 453
software reset, 452
watchdog-reset, 452–454
Xmega RTC, 439–441
CNT, 439, 440
CTRL, 439, 440
instellen realtime counter, 439
instellen realtime counter met extern kristal, 440
opbouw, 439
PER, 439, 440
RTC_OVF_vect, 440
RTC_PRESCALER_DIV*n*_gc, 439, 440
RTC_SYNCBUSY_bm, 439, 440
STATUS, 439
Xmega sleep, 448–451, *zie ook* sleep-bibliotheek

Xmega SPI, 354–358
communicatie met extern EEPROM, 357–358
communicatie met nRF24L01+, 479
CPHA, fasebit, 355
CPOL, polariteitsbit, 355
CTRL, 355
DATA, 355
IF, interrupt flag, 355
INTCTRL, 355
SPI_CLK2X_bm, 356
SPI_DORD_bm, 356
SPI_ENABLE_bm, 356
SPI_IF_bm, 356
SPI_MASTER_bm, 356
SPI_MODE_0_gc, 356
PRESCALER_DIVx_gc, 356
SPIx, 356
STATUS, 355
USART als SPI, 362–363
Xmega timer/counter, 255–257, 383–408
AWeX, advanced waveform extension, 403
HiRES, high resolution, 403
32-bits inputcapture, 432–433
aansluiting inputcapture, 429
capture-modus, 427–434
CCx, 385, 386, 389, 399, 428
CCxBUF, 386, 394, 399, 400
CCxBUFH, 487
CCxBUFL, 487
CLKSEL-bits, 230
CNT, 234, 388
configuratie voor 32-bits inputcapture, 432
CTRLA, prescaling, 232–233, 256
CTRLB, 387
CTRLB, CCx, 235
CTRLB, modus, 232–233, 256
CTRLD, 432, 434
CTRLFSET, 428, 431, 432
dual-slope PWM, 385, 393, 397–403
duty-cycle bij dual-slope-modus, 393
duty-cycle bij single-slope-modus, 391
EVACT-bits, 434
frequentie bij dual-slope-modus, 393
frequentie bij frequentiemodus, 390
frequentie bij single-slope-modus, 391
frequentie-capture, 433–434
frequentiemodus, 385, 388–390, 404–408
inputcapture-modus, 427–434
INTCTRLA, 232–233, 256, 348, 350, 407
INTCTRLB, 431
normale modus, 385, 387–388
overzicht PWM, 385–386
PER, 232–233, 256, 386, 388, 428
PERBUF, 386
periodetijd bij normale modus, 230, 232
prescaling selectiebits, 230
single-slope PWM, 385, 390–392, 394–397
TC0_CCAEN_bm, 235
TC0_CCxEN_bm, 389, 394, 396, 400, 402
TC1_EVDLY_bm, time delay, 432
TC_CCAINTLVL_LO_gc, 431
TC_CLKSEL_DIVn_gc, 389, 394, 407
TC_CLKSEL_EVCHn_gc, 432
TC_CLKSEL_OFF_gc, 406
TC_CMD_RESTART_gc, 428, 431, 432
TC_EVACT_CAPT_gc, 428, 432
TC_EVACT_FRQ_gc, 434
TC_EVSEL_CH0_gc, 428, 432, 434
TC_OVFINTLVL_LO_gc, 232–233
TC_OVFINTLVL_OFF_gc, 237, 407
TC_WGMODE_DSBOTH_gc, 387, 394, 400, 402
TC_WGMODE_FRQ_gc, 235, 387, 389, 406
TC_WGMODE_NORMAL_gc, 232–233, 256, 348, 350, 387, 406, 428, 450, 478
TC_WGMODE_SINGLESLOPE_gc, 387, 391, 392, 394, 396, 428
TC_WGMODE_SS_gc, 387
TC_WGMODE_t, 387
TCC0_OVF_vect, 450
TCC1_OVF_vect, 478
TCD0_OVF_vect, 256
TCE0_CCA_vect, 428, 431, 432
TCE0_OVF_vect, 232–233, 237, 387, 431, 432, 455
TCE1_OVF_vect, 406
WGM-bits, 387
Xmega TWI, 366–382
ADDR, master, 368
BAUD, master, 368
CTRL, master, 368
CTRLA, master, 368
DATA, master, 368
I2C-bibliotheek op basis van TWI_t, 367–374
I2C-bibliotheek, toepassing met DS3232, 370–374
levelshifting, 374–375
masterdriver+, 375–378
slavedriver+, 378–382
TWI_BAUD(), 369, 373, 377
TWI_MASTER_BUSSTATE_gm, 368
TWI_MASTER_BUSSTATE_IDLE_gc, 368
TWI_MASTER_CMD_RECVTRANS_gc, 368
TWI_MASTER_CMD_STOP_gc, 368
TWI_MASTER_ENABLE_bm, 368
TWI_MASTER_INTLVL_LO_gc, 379, 381
TWI_MASTER_RIF_bm, 368
TWI_MASTER_RXACK_bm, 368
TWI_MASTER_t, 366, 367
TWI_Master_t, 376
TWI_MASTER_WIF_bm, 368
TWI_SLAVE_INTLVL_LO_gc, 379, 381
TWI_SLAVE_t, 366, 367
TWI_Slave_t, 378
TWI_t, 366, 367, 376, 378
TWIx_TWIM_vect, 377, 379, 381
TWIx_TWIS_vect, 379, 381
Xmega USART
BAUDCTRLx, 293–295, 299, 363
BSEL, 295
SCALE, 295
CHSIZE, databits, 296
CMODE communication modus, 296
CTRL, 296
CTRLB, 299
CTRLC, 299, 363
data register empty, 303, 305, 306
DATA, data register, 297, 300, 303
DATA, receive, 293, 299
DATA, transmit, 293, 299, 302
DREIF, data register empty flag, 297, 307
MSPI-bit, 363
PMODE pariteitsmodus, 296
receive complete, 302, 303, 305, 306
RXCIF, receive complete flag, 297, 307
RXEN-bit, 299
SBMODE, stopbit, 296
SCALE, 293–295
BSEL, 293–295, 362
STATUS, 297
TXCIF, transmit complete flag, 297, 363
TXEN-bit, 299
uart-driver Atmel, 309
USART als SPI, 362–363
USART_BSCALE0_bp, 295, 299
USART_BSCALE_gm, 295, 299
USART_BSEL_gm, 295, 299
USART_CHSIZE_nBIT_gc, 299
USART_CMODE_ASYNCHRONOUS_gc, 299
USART_CMODE_MSPI_gc, 363
USART_DREIF_bm, 299, 300
USART_PMODE_DISABLED_gc, 299
USART_RXCIF_bm, 300
USART_RXCINTLVL_LO_gc, 302
USART_RXEN_bm, 300
USART_TXEN_bm, 299, 300
USARTxn_DRE_vect, 303, 305, 307
USARTxn_RXC_vect, 302, 303, 305, 307
Xmega watchdog, 452–454
CEN-bit, 452
CTRL, 452
ENABLE-bit, 452
PER-bits, watchdog timer prescaler, 452, 453
WDRF-bit, 454
WDT_PER_nCLK_gc-bits, 453
WINCTRL, 452
Xmega-bord, 515–518
Xmega256a3u, 516
Xmega32a4u, 516

Z

zender, 292, 293, 297, 304, 484–488
zonnebloem, 120
zwevende ingang, 451